# RapidMiner 8

## Operator Reference Manual

rapidminer

# RapidMiner 8
## Operator Reference Manual

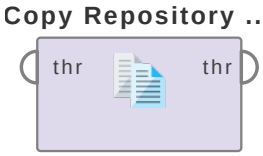
November 23, 2018

RapidMiner GmbH
www.rapidminer.com

# Contents

# 1Data Access

## Copy Repository Entry




An operator to copy a repository entry to another repository location.

### Description

Copies an entry to a new parent folder. If destination references a folder, the source entry is copied to that folder. If it references an existing entry and overwriting is not enabled (default case), an exception is raised. If overwriting is enabled the existing entry will be overwritten. If it references a location which does not exist, say, “/root/folder/leaf”, but the parent exists (in this case “/root/folder”), a new entry named by the last path component (in this case “leaf”) is created.

### Input Ports

**input** ***(inp)***

### Output Ports

**output** ***(out)***

### Parameters

**source entry** Entry that should be copied

**destination** Copy destination

**overwrite** Overwrite elements at copy destination?

# Delete Repository Entry


An operator to delete a repository entry within a process.

## Description

An operator to delete a repository entry within a process.

## Input Ports

**input** ***(inp)***

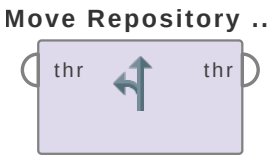
## Output Ports

**output** ***(out)***

## Parameters

**entry to delete** Entry that should be deleted

## Move Repository Entry




An operator to move a repository entry to another repository location.

### Description

Moves an entry to a new parent folder. If destination references a folder, the source entry is moved to that folder. If it references an existing entry and overwriting is not enabled (default case), an exception is raised. If overwriting is enabled the existing entry will be overwritten. If it references a location which does not exist, say, "/root/folder/leaf", but the parent exists (in this case "/root/folder"), a new entry named by the last path component (in this case "leaf") is created.

### Input Ports

**input** ***(inp)***

### Output Ports

**output** ***(out)***

### Parameters

**source entry** Entry that should be moved

**destination** Destination for move action

**overwrite** Overwrite elements at move destination?

# Rename Repository Entry


An operator to rename repository a entry within a process.

## Description

An operator to rename a repository entry. The user can select the entry that should be renamed, a new name and if an already existing entry should be overwritten or not. If overwriting is not allowed (default case) a user error is thrown if there already exists another element with the new name.

## Input Ports

**input *(inp)***

## Output Ports

**output *(out)***

## Parameters

**entry to rename** Entry that should be renamed

**new name** New entry name

**overwrite** Overwrite already existing entry with same name?

# Retrieve


This Operator can access stored information in the Repository and load them into the Process.

## Description

The Retrieve Operator loads a RapidMiner Object into the Process. This Object is often an ExampleSet but it can also be a Collection or a Model. Retrieving data this way also provides the meta data of the RapidMiner Object.

## Differentiation

This Operator is like the different *Read <source>* Operators in the *Data Access* group. Storing the data inside a repository gives one the advantage that meta data properties are stored as well. Meta data gives you additional information about the RapidMiner Object you retrieve. For an ExampleSet this is e.g. the names and types of Attributes, their range and how many missing values there are. Meta data allows you to easily configure parameters of other Operators, for example you can select Attributes from a list of available Attributes.

The data stored in the Repository can only be changed within a RapidMiner Process. Data stored on disk or within database can be changed by other means.

## Output Ports

**output *(out)*** It returns the RapidMiner Object whose path was specified in *repository entry* parameter.

## Parameters

**repository entry** The path to the RapidMiner Object which should be loaded. This parameter references an entry in the repository, which will be returned as output of this Operator.

Repository locations are resolved relative to the Repository folder containing the current Process. Folders in the Repository are separated by a forward slash ('/'). A '..' references the parent folder. A leading forward slash references the root folder of the Repository containing the current Process. A leading double forward slash ('//') is interpreted as an absolute path starting with the name of a Repository. The list below shows the different methods:

- 'MyData' looks up an entry 'MyData' in the same folder as the current Process
- '../Input/MyData' looks up an entry 'MyData' located in a folder 'Input' next to the folder containing the current Process
- '/data/Model' looks up an entry 'Model' in a top-level folder 'data' in the Repository holding the current Process
- '//Samples/data/Golf' looks up the Iris data set in the 'Samples' Repository.

When using the "Select the repository location" button, it is possible to check if the path should be resolved relative. This is useful when sharing Processes with others.

## Tutorial Processes

### Load Example Data using the Retrieve Operator

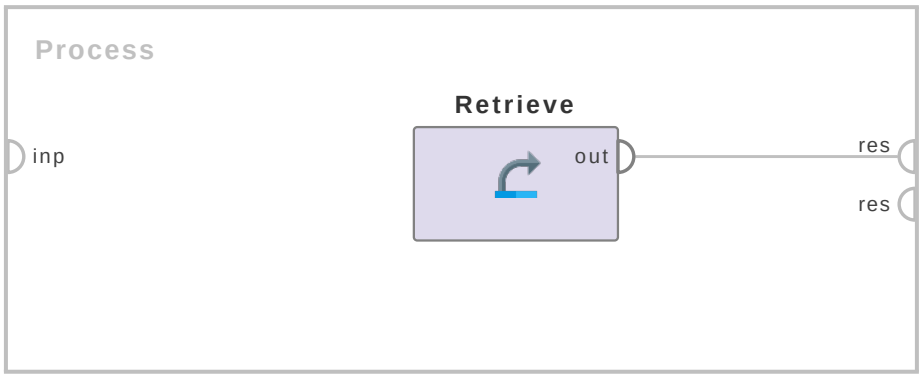
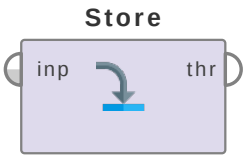



Figure 1.1: Tutorial process ‘Load Example Data using the Retrieve Operator’.

This Process loads the Golf data set from repository. The repository entry parameter is provided as an absolute path ‘//Samples/data/Golf’. Thus the Golf data set is returned from the Samples repository and the sub-folder data.

# Store




This operator stores an IO Object in the data repository.

## Description

This operator stores an IO Object at a location in the data repository. The location of the object to be stored is specified through the *repository entry* parameter. The stored object can be used by other processes by using the Retrieve operator. Please see the attached Example Processes to understand the basic working of this operator. The Store operator is used to store an ExampleSet and a model in the Example Processes.

## Input Ports

**input *(inp)*** This port expects an IO Object. In the attached Example Processes an ExampleSet and a model are provided as input.

## Output Ports

**through *(thr)*** The IO Object provided at the input port is delivered through this output port without any modifications.This is usually used to reuse the same IO Object in further operators of the process.

## Parameters

**repository entry *(string)*** This parameter is used to specify the location where the input IO Object is to be stored.

## Tutorial Processes

### Storing an ExampleSet using the Store operator

This Process shows how the Store operator can be used to store an ExampleSet. The 'Golf' data set and the 'Golf-Testset' data set are loaded using the Retrieve operator. These ExampleSets are merged using the Append operator. The resultant ExampleSet is named 'Golf-Complete' and stored using the Store operator. The stored ExampleSet is used in the third Example Process.

### Storing a model using the Store operator

This Process shows how the Store operator can be used to store a model. The 'Golf' data set is loaded using the Retrieve operator. The Naive Bayes operator is applied on it and the resultant model is stored in the repository using the Store operator. The model is stored with the name 'Golf-Naive-Model'. The stored model is used in the third Example Process.

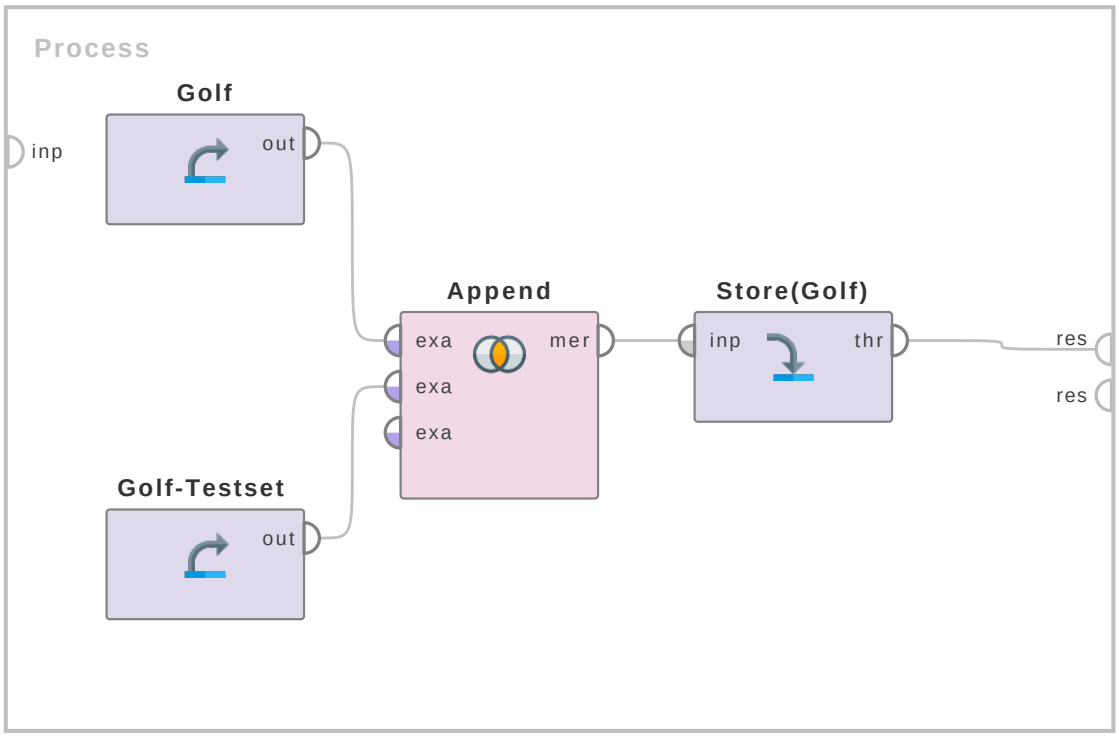



Figure 1.2: Tutorial process 'Storing an ExampleSet using the Store operator'.

### Using the objects stored by the Store operator

This Process shows how a stored IO Object can be used. The 'Golf-Complete' data set stored in the first Example Process and the 'Golf-Naive-Model' stored in the second Example Process is loaded using the Retrieve operator. The Apply Model operator is used to apply the 'Golf-Naive-Model' on the 'Golf-Complete' data set. The resultant labeled ExampleSet can be viewed in the Results Workspace.

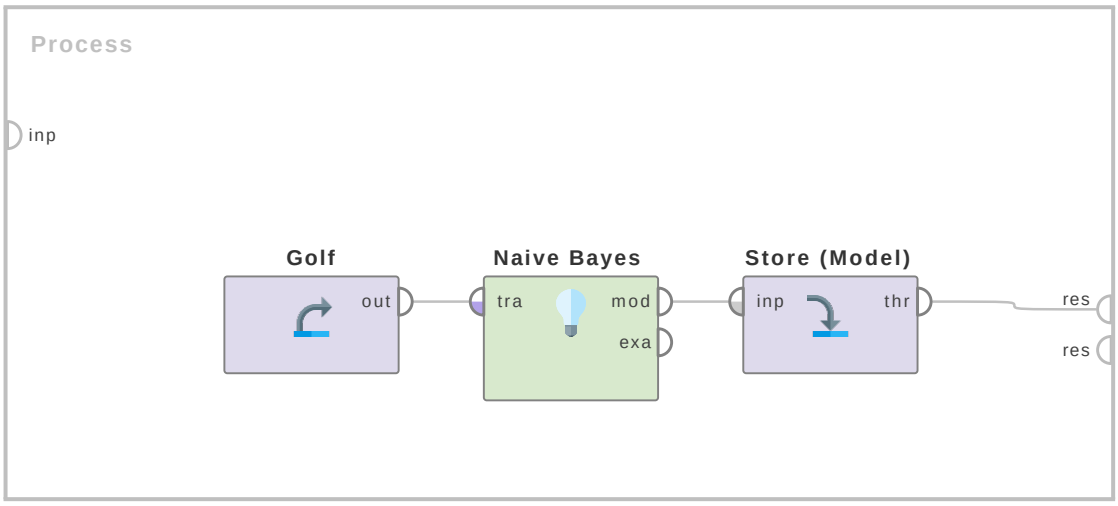



Figure 1.3: Tutorial process 'Storing a model using the Store operator'.

# 1.1 Files

## 1.1.1 Read

### Read ARFF


This operator is used for reading an ARFF file.

#### Description

This operator can read ARFF (Attribute-Relation File Format) files known from the machine learning library Weka. An ARFF file is an ASCII text file that describes a list of instances sharing a set of attributes. ARFF files were developed by the Machine Learning Project at the Department of Computer Science of The University of Waikato for use with the Weka machine learning software. Please study the attached Example Process for understanding the basics and structure of the ARFF file format. Please note that when an ARFF file is written, the roles of the attributes are not stored. Similarly when an ARFF file is read, the roles of all the attributes are set to regular.

#### Input Ports

**file *(fil)*** An ARFF file is expected as a file object which can be created with other operators with file output ports like the Read File operator.

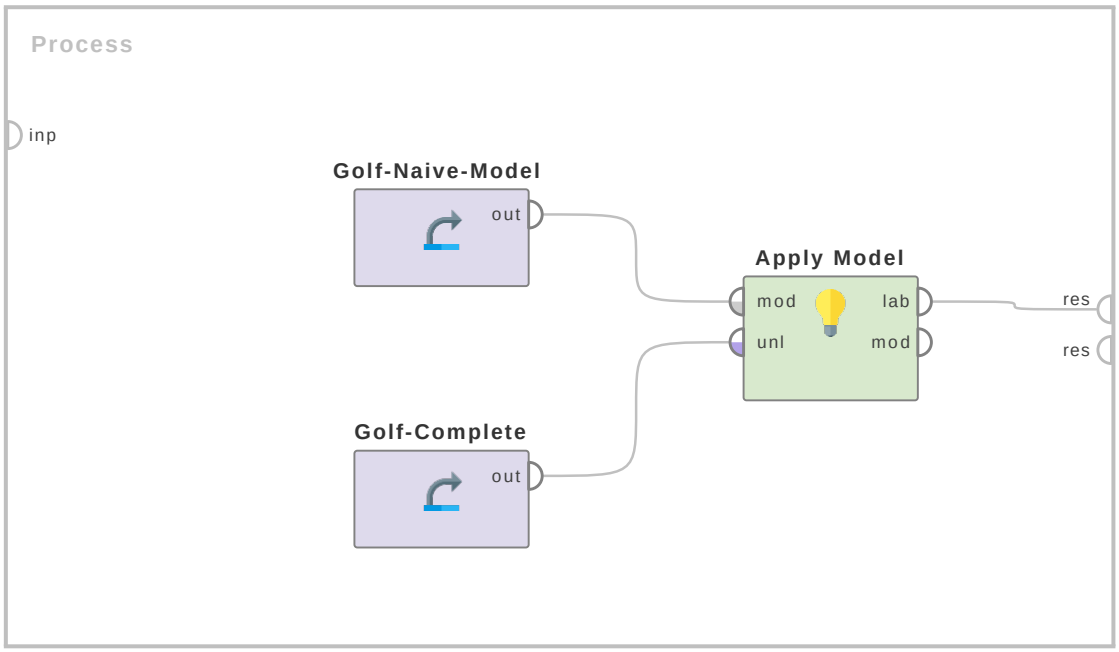



Figure 1.4: Tutorial process 'Using the objects stored by the Store operator'.

## Output Ports

**output *(out)*** This port delivers the ARFF file in tabular form along with the meta data. This output is similar to the output of the Retrieve operator.

## Parameters

**data file *(filename)*** The path of the ARFF file is specified here. It can be selected using the *choose a file* button.

**encoding *(selection)*** This is an expert parameter. A long list of encoding is provided; users can select any of them.

**read not matching values as missings *(boolean)*** This is an expert parameter. If this parameter is set to true, values that do not match with the expected value type are considered as missing values and are replaced by '?'. For example if 'abc' is written in an integer column, it will be treated as a missing value. Question mark (?) in ARFF file is also read as missing value.

**decimal character *(char)*** This character is used as the decimal character.

**grouped digits *(boolean)*** This parameter decides whether grouped digits should be parsed or not. If this parameter is set to true, the *grouping character* parameter should be specified.

**grouping character *(char)*** This parameter is available only when the *grouped digits* parameter is set to true.This character is used as the grouping character. If it is found between numbers, the numbers are combined and this character is ignored. For example if "22-14" is present in the ARFF file and "-" is set as *grouping character*, then "2214" will be stored.

**infinity string *(string)*** This parameter can be set to parse a specific infinity representation (e.g. “Infinity”). If it is not set, the local specific infinity representation will be used.

## Tutorial Processes

### The basics of the ARFF

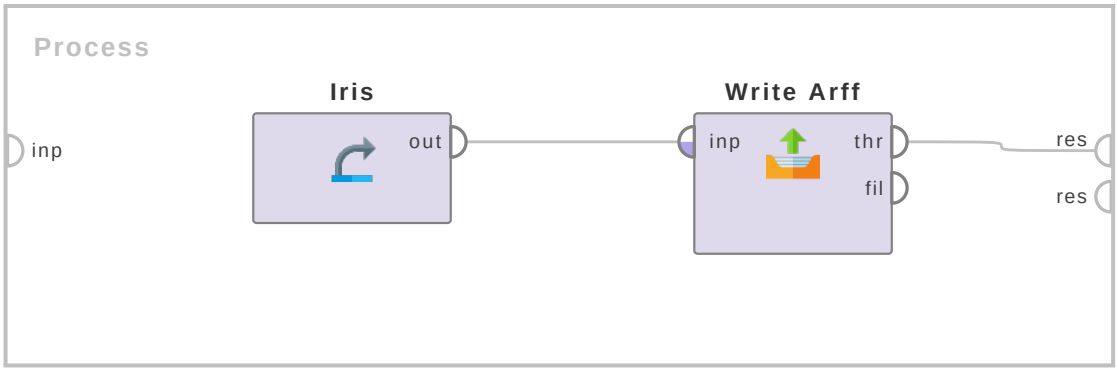



Figure 1.5: Tutorial process ‘The basics of the ARFF’.

The ‘Iris’ data set is loaded using the Retrieve operator. The Write ARFF operator is applied on it to write the ‘Iris’ data set into an ARFF file. The example set file parameter is set to ‘D:\Iris’. Thus an ARFF file is created in the ‘D’ drive of your computer with the name ‘Iris’. Open this file to see the structure of an ARFF file.

ARFF files have two distinct sections. The first section is the Header information, which is followed by the Data information. The Header of the ARFF file contains the name of the Relation and a list of the attributes. The name of the Relation is specified after the @RELATION statement. The Relation is ignored by RapidMiner. Each attribute definition starts with the @ATTRIBUTE statement followed by the attribute name and its type. The resultant ARFF file of this Example Process starts with the Header. The name of the relation is ‘RapidMinerData’. After the name of the Relation, six attributes are defined.

Attribute declarations take the form of an ordered sequence of @ATTRIBUTE statements. Each attribute in the data set has its own @ATTRIBUTE statement which uniquely defines the name of that attribute and its data type. The order of declaration of the attributes indicates the column position in the data section of the file. For example, in the resultant ARFF file of this Example Process the ‘label’ attribute is declared at the end of all other attribute declarations. Therefore values of the ‘label’ attribute are in the last column of the Data section.

The possible attribute types in ARFF are: numeric integer real {nominalValue1,nominalValue2,...} for nominal attributes string for nominal attributes without distinct nominal values (it is however recommended to use the nominal definition above as often as possible) date [date-format] (currently not supported by RapidMiner)

You can see in the resultant ARFF file of this Example Process that the attributes ‘a1’, ‘a2’, ‘a3’ and ‘a4’ are of real type. The attributes ‘id’ and ‘label’ are of nominal type. The distinct nominal values are also specified with these nominal attributes.

The ARFF Data section of the file contains the data declaration line @DATA followed by the actual example data lines. Each example is represented on a single line, with carriage returns denoting the end of the example. Attribute values for each example are delimited by commas.

They must appear in the order that they were declared in the Header section (i.e. the data corresponding to the n-th @ATTRIBUTE declaration is always the n-th field of the example line). Missing values are represented by a single question mark (?).

A percent sign (%) introduces a comment and will be ignored during reading. Attribute names or example values containing spaces must be quoted with single quotes (’). Please note that in RapidMiner the sparse ARFF format is currently only supported for numerical attributes. Please use one of the other options for sparse data files provided by RapidMiner if you also need sparse data files for nominal attributes.

### Reading an ARFF file using the Read ARFF operator

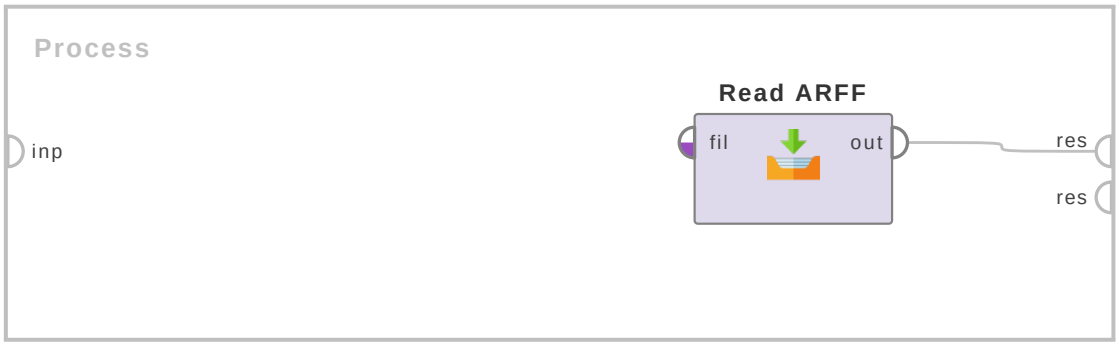



Figure 1.6: Tutorial process ‘Reading an ARFF file using the Read ARFF operator’.

The ARFF file that was written in the first Example Process using the Write ARFF operator is retrieved in this Example Process using the Read ARFF operator. The data file parameter is set to ‘D:\Iris’. Please make sure that you specify the correct path. All other parameters are used with default values. Run the process. You will see that the results are very similar to the original Iris data set of RapidMiner repository. Please note that the role of all the attributes is regular in the results of the Read ARFF operator. Even the roles of ‘id’ and ‘label’ attributes are set to regular. This is so because the ARFF files do not store information about the roles of the attributes.

# Read Access


This operator reads an ExampleSet from a Microsoft Access database.

## Description

The Read Access operator is used for reading an ExampleSet from the specified Microsoft Access database (.mdb or .accdb extension). You need to have at least basic understanding of databases, database connections and queries in order to use this operator properly. Go through the parameters and Example Process to understand the flow of this operator.

## Output Ports

**output *(out)*** This port delivers the result of the query on database in tabular form along with the meta data. This output is similar to the output of the Retrieve operator.

## Parameters

**username *(string)*** This parameter is used to specify the username of the database (if any).

**password *(string)*** This parameter is used to specify the password of the database (if any).

**define query *(selection)*** Query is a statement that is used to select required data from the database. This parameter specifies whether the database query should be defined directly, through a file or implicitly by a given table name. The SQL query can be auto generated giving a table name, passed to RapidMiner via a parameter or, in case of long SQL statements, in a separate file. The desired behavior can be chosen using the *define query* parameter. Please note that column names are often case sensitive and might need quoting.

**query *(string)*** This parameter is only available when the *define query* parameter is set to 'query'. This parameter is used to define the SQL query to select desired data from the specified database.

**query file *(filename)*** This parameter is only available when the *define query* parameter is set to 'query file'. This parameter is used to select a file that contains the SQL query to select desired data from the specified database. Long queries are usually stored in files. Storing queries in files can also enhance reusability.

**table name *(string)*** This parameter is only available when the *define query* parameter is set to 'table name'. This parameter is used to select the required table from the specified database.

**database file *(filename)*** This parameter specifies the path of the Access database i.e. the mdb or accdb file.

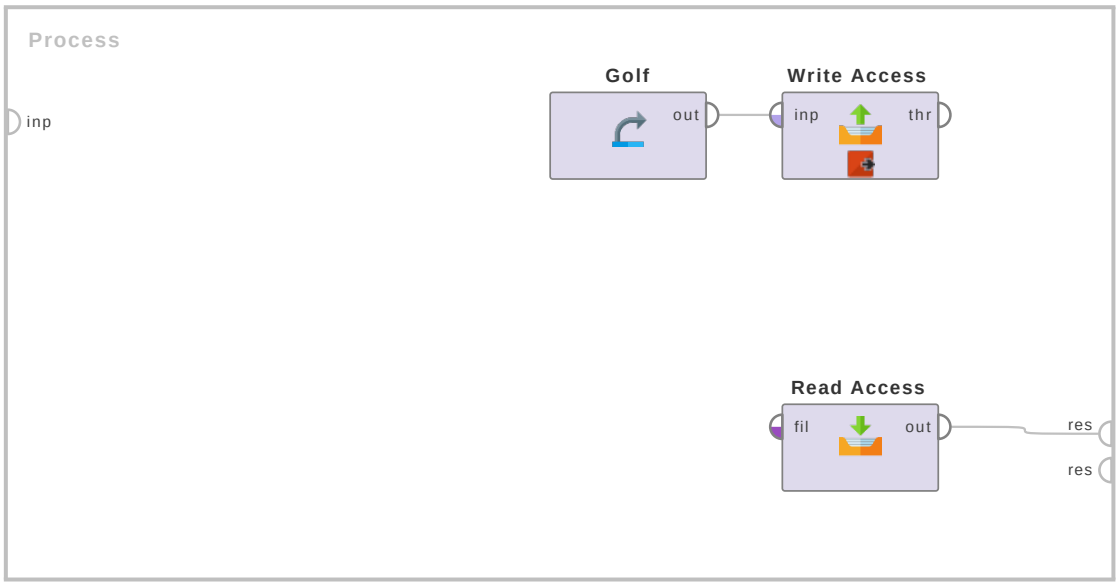



Figure 1.7: Tutorial process 'Writing and then reading data from an Access database'.

## Tutorial Processes

### Writing and then reading data from an Access database

The 'Golf' data set is loaded using the Retrieve operator. The Write Access operator is used for writing this ExampleSet into the golf table of the 'golf_db.mdb' database. The database file parameter is provided with the path of the database file 'golf_db.mdb' and the name of the desired table is specified in the table name parameter ( i.e. it is set to 'golf'). A breakpoint is inserted here. No results are visible in RapidMiner at this stage but you can see that at this point of the execution the database has been created and the golf table has been filled with the examples of the 'Golf' data set.

Now the Read Access operator is used for reading the golf table from the 'golf_db.mdb' database. The database file parameter is provided with the path of the database file 'golf_db.mdb'. The define query parameter is set to 'table name'. The table name parameter is set to 'golf' which is the name of the required table. Continue the process, you will see the entire golf table in the Results Workspace. The define query parameter is set to 'table name' if you want to read an entire table from the database. You can also read a selected portion of the database by using queries. Set the define query parameter to 'query' and specify a query in the query parameter.

# Read BibTeX


This operator can read BibTeX files.

## Description

This operator can read BibTeX files. It uses Stefan Haustein's kdb tools.

## Input Ports

**file *(fil)*** An BibTeX file is expected as a file object which can be created with other operators with file output ports like the Read File operator.

## Output Ports

**output *(out)*** This port delivers the BibTeX file in tabular form along with the meta data. This output is similar to the output of the Retrieve operator.

## Parameters

**label attribute *(string)*** The (case sensitive) name of the label attribute

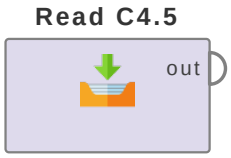
**id attribute *(string)*** The (case sensitive) name of the id attribute

**weight attribute *(string)*** The (case sensitive) name of the weight attribute

**datamanagement *(selection)*** Determines, how the data is represented internally

**data file *(filename)*** The file containing the data

# Read C4.5




This operator can read data and meta given in C4.5 format.

## Description

Loads data given in C4.5 format (names and data file). Both files must be in the same directory. You can specify one of the C4.5 files (either the data or the names file) or only the filestem.

For a dataset named "foo", you will have two files: foo.data and foo.names. The .names file describes the dataset, while the .data file contains the examples which make up the dataset.

The files contain series of identifiers and numbers with some surrounding syntax. A | (vertical bar) means that the remainder of the line should be ignored as a comment. Each identifier consists of a string of characters that does not include comma, question mark or colon. Embedded whitespce is also permitted but multiple whitespace is replaced by a single space.

The .names file contains a series of entries that describe the classes, attributes and values of the dataset. Each entry can be terminated with a period, but the period can be omited if it would have been the last thing on a line. The first entry in the file lists the names of the classes, separated by commas. Each successive line then defines an attribute, in the order in which they will appear in the .data file, with the following format:

attribute-name : attribute-type

The attribute-name is an identifier as above, followed by a colon, then the attribute type which must be one of

- *continuous:* If the attribute has a continuous value.
- *discrete [n]:* The word 'discrete' followed by an integer which indicates how many values the attribute can take (not recommended, please use the method depicted below for defining nominal attributes).
- *[list of identifiers]:* This is a discrete, i.e. nominal, attribute with the values enumerated (this is the prefered method for discrete attributes). The identifiers should be separated by commas.
- *ignore:* This means that the attribute should be ignored - it won't be used. This is not supported by RapidMiner, please use one of the attribute selection operators after loading if you want to ignore attributes and remove them from the loaded example set.

Here is an example .names file:

good, bad. dur: continuous. wage1: continuous. wage2: continuous. wage3: continuous. cola: tc, none, tcf. hours: continuous. pension: empl_contr, ret_allw, none. stby_pay: continuous. shift_diff: continuous. educ_allw: yes, no. ...

Foo.data contains the training examples in the following format: one example per line, attribute values separated by commas, class last, missing values represented by "?". For example:

2,5.0,4.0,?,none,37,?,?,5,no,11,below_average,yes,full,yes,full,good 3,2.0,2.5,?,?,35,none,?,?,?,10,average,?,?,
3,4.5,4.5,5.0,none,40,?,?,?,no,11,average,?,half,?,?,good 3,3.0,2.0,2.5,tc,40,none,?,5,no,10,below_average,yes,half,yes,full,bad ...

## Output Ports

**output *(out)*** This port delivers the C4.5 file in tabular form along with the meta data. This output is similar to the output of the Retrieve operator.

## Parameters

**c45 filestem *(filename)*** The path to either the C4.5 names file, the data file, or the filestem (without extensions). Both files must be in the same directory.

**datamanagement *(selection)*** Determines, how the data is represented internally.

**decimal point character *(char)*** Character that is used as decimal point.

**encoding *(selection)*** The encoding used for reading or writing files.

# Read CSV


This Operator reads an ExampleSet from the specified CSV file.

## Description

CSV is an abbreviation for Comma-Separated Values. The CSV files store data (both numerical and text) in plain-text form. All values corresponding to an Example are stored as one line in the CSV file. Values for different Attributes are separated by a separator character. The separator remains constant. Each row in the file uses the constant separator for separating Attribute values. The term 'CSV' suggests that the Attribute values would be separated by commas, but other separators can also be used.

The easiest way to import a CSV file is to use the *Import Configuration Wizard* from the Parameters panel. All parameters can also directly be set in the Parameters panel. For more details about the Operator, see the description of the parameters.

Please make sure that the CSV file is read correctly as an ExampleSet before building a Process that uses it.

## Differentiation

There are many *Read <source>* Operators in the *Data Access* group and *Files/Read* sub-group. For example, there is *Read Excel*, *Read URL*, *Read SPSS*, *Read XML* and other Operators, which can read ExampleSet from different file formats.

## Input Ports

**file *(fil)*** A CSV file can be optionally passed in as a file object. This can be created with Operators having file output ports such as the *Read File* Operator.

## Output Ports

**output *(out)*** This port delivers the ExampleSet created from the CSV file provided at the input port, imported through the *Import Configuration Wizard* or loaded from the path given to the *csv file* parameter.

## Parameters

**Import Configuration Wizard** This user-friendly wizard guides you to easily configure this Operator to import the CSV file.

**csv file** The path of the CSV file is specified here. It can also be selected using the 'Choose a file' button.

**column separators** Column separators for CSV files can be specified here. It can also be provided as a regular expression. A good understanding of regular expressions can be developed by studying the description of Select Attributes Operator and its tutorial Processes.

**trim lines** This parameter indicates if lines should be trimmed (removal of empty spaces at the beginning and the end) before the column split is performed. This option might be problematic if TABs ('\t') are used as separators.

**use quotes** This parameter indicates if quotes should be regarded. Quotes can be used to store special characters like *column separators*. For example if (,) is set as *column separator* and (") is set as *quotes character*, then a row (a,b,c,d) will be translated as 4 values for 4 columns. On the other hand ("a,b,c,d") will be translated as a single column value a,b,c,d. If this parameter is set to false, the *quotes character* parameter and the *escape character* parameter cannot be defined.

**quotes character** This parameter defines the quotes character and is only available if *use quotes* is set to true.

**escape character** This parameter specifies the character used to escape the quotes and is only available if *use quotes* is set to true. For example, if (") is used as *quotes character* and ('\') is used as *escape character*, then ("yes") will be translated as (yes) and (\"yes\") will be translated as ("yes").

**skip comments** This parameter is used to ignore comments in the CSV file (if any). If this option is set to true, a comment character should be defined using the *comment characters* parameter.

**comment characters** This parameter is available if *comment characters* is set to true. Lines beginning with these characters are ignored. If this character is present in the middle of the line, anything that comes in that line after this character is ignored. The *comment character* itself is also ignored.

**parse numbers** This parameter specifies whether numbers are parsed or not.

**decimal character** This character is used as the decimal character.

**grouped digits** This parameter decides whether grouped digits should be parsed or not. If this parameter is set to true, a *grouping character* parameter has to be specified.

**grouping character** This character is used as the grouping character. If this character is found between numbers, the numbers are combined and this character is ignored. For example if "22-14" is present in the CSV file and "-" is set as the *grouping character*, then "2214" will be stored.

**infinity string *(string)*** This parameter can be set to parse a specific infinity representation (e.g. "Infinity"). If it is not set, the local specific infinity representation will be used.

**date format** The parameter specifies the date and time format. Many predefined options exist but users can also specify a new format. If text in a CSV file column matches this date format, that column is automatically converted to date type.

Some corrections are automatically made on invalid date values. For example, a value '32-March' will automatically be converted to '1-April'.

Columns containing values which cannot be interpreted as numbers will be interpreted as nominal, as long as they do not match the date and time pattern of the date format parameter. If they match, this column of the CSV file will be automatically parsed as date and the corresponding Attribute will be of type date.

**first row as names** If this parameter is set to true, it is assumed that the first line of the CSV file has the names of the Attributes. If so, the Attributes are automatically named and the first line of the CSV file is not treated as a data line.

**annotations** If the *first row as names* is not set to true, annotations can be added using the 'Edit List' button of this parameter, which opens a new menu. This menu allows you to select any row and assign an annotation to it. Name, Comment and Unit annotations can be assigned. If row 0 is assigned a Name annotation, it is equivalent to setting the *first row as names* parameter to true. If you want to ignore any row, you can annotate them as Comment. Remember that row number in this menu does not count commented lines.

**time zone** Users can select any time zone from the list of provided time zones.

**locale** Users can select any locale from the list of provided locales.

**encoding** Users can select any encoding from the list of provided encodings.

**read all values as polynominal** This option allows you to disable the type handling for this operator. Every column will be read as a polynominal attribute.

**data set meta data information** This parameter allows to adjust or override the meta data of the CSV file. Column index, name, type and role can be specified here.

The Read CSV Operator automatically tries to determine an appropriate data type of the Attributes by reading the first few lines and checking the occurring values. Integer values are assigned the integer data type, real values the real data type. Values which cannot be interpreted as numbers are assigned the nominal data type, as long as they do not match the format of the *date format* parameter.

With the *data set meta data information* parameter, this automatic assignment can be adjusted or overwritten.

**read not matching values as missings** If this parameter is set to true, values that do not match with the expected value type are considered as missing values and are replaced by '?'. For example, if 'abc' is written in an integer column, it will be treated as a missing value. A question mark (?) in the CSV file is also read as a missing value.

**data management** This parameter determines how the data is represented internally. Users can select any option from the provided list.

## Tutorial Processes

### Read a CSV file

Save the following text in a text file.

```
att1,att2,att3,att4 # row 1
80.6, yes , 1996.JAN.21 ,22-14 # row 2
12.43,"yes",1997.MAR.30,23-22 # row 3
13.5,\"no\",1998.AUG.22,23-14 # row 4
23.3,yes,1876.JAN.32,42-65# row 5
21.6,yes,2001.JUL.12,xyz # row 6
12.56,",_?",2002.SEP.18,15-90# row 7
```

This is a sample CSV file. You can load this with the given tutorial Process by providing its path in the csv file parameter or by using the 'Choose a file' button. Run the Process and compare the results in the Results view with the CSV file. The Process performs the following actions:

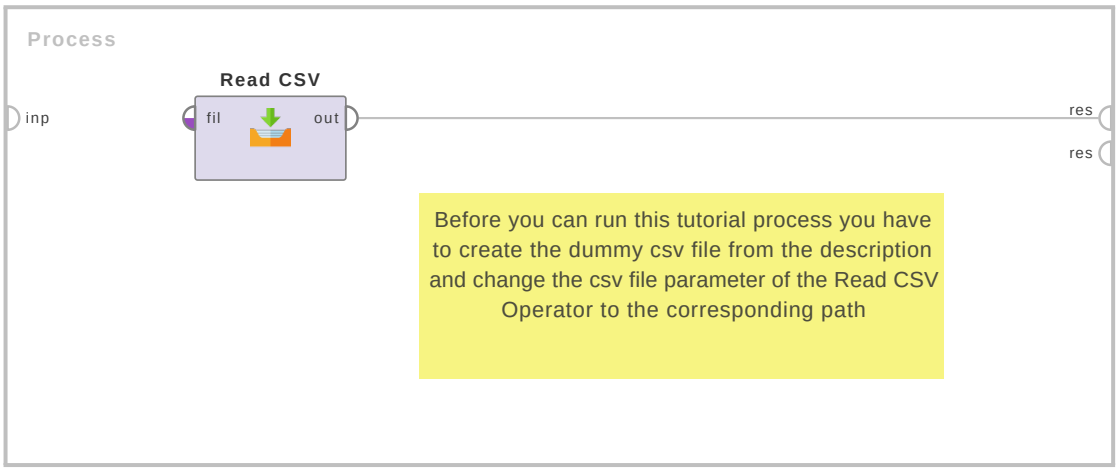



Figure 1.8: Tutorial process ‘Read a CSV file’.

‘#’ is defined as a comment character so ‘row {number}’ is ignored in all rows.

As the first row as names parameter is set to true, att1, att2, att3 and att4 are set as Attribute names.

The Attribute att1 is set as real , att2 as polynominal, att3 as date and att4 as real.

For Attribute att4, the ‘-’ character is ignored in all rows because the grouped digits parameter is set to true and ‘-’ is specified as the grouping character.

In row 2, the white spaces at the start and end of values are ignored because trim lines parameter is set to true.

In row 3, quotes are not ignored because use quotes is set to true, the content inside the quotes is taken as the value for Attribute att2.

In row 4, (\”no\”) is taken as a (no) in quotes, cause the escape character is set to ‘\’.

In row 5, the date value is automatically corrected from ‘JAN.32’ to ‘Feb.1’.

In row 6, an invalid real value for the Attribute att4 is replaced by ‘?’ because the read not matching values as missings parameter is set to true.

In row 7, quotes are used to retrieve special characters as values including the column separator (,) and a question mark.

# Read dBase


This operator can read dBase files.

## Description

This operator can read dBase files. It uses Stefan Haustein's kdb tools.

## Input Ports

**file *(fil)*** An dBase file is expected as a file object which can be created with other operators with file output ports like the Read File operator.

## Output Ports

**output *(out)*** This port delivers the dBase file in tabular form along with the meta data. This output is similar to the output of the Retrieve operator.

## Parameters

**label attribute *(string)*** The (case sensitive) name of the label attribute

**id attribute *(string)*** The (case sensitive) name of the id attribute

**weight attribute *(string)*** The (case sensitive) name of the weight attribute

**datamanagement *(selection)*** Determines, how the data is represented internally.

**data file *(filename)*** The file containing the data

# Read DASYLab


This operator can read DASYLab data files.

## Description

This operator allows to import data from DASYLab files (.DDF) into RapidMiner. Currently only universal format 1 is supported. External files (.DDB) and histogram data are currently not supported.

The parameter timestamp allows to configure whether and what kind of timestamp should be included in the example set. If it is set to *relative*, the timestamp attribute captures the amount of milliseconds since the file start time. If it is set to *absolute*, the absolute time is used to timestamp the examples.

## Input Ports

**file *(fil)*** A DASYLab file is expected as a file object which can be created with other operators with file output ports like the Read File operator.

## Output Ports

**output *(out)*** This port delivers the DASYLab file in tabular form along with the meta data. This output is similar to the output of the Retrieve operator.

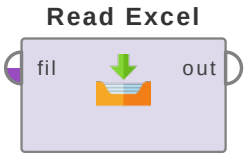
## Parameters

**filename *(filename)*** Name of the file to read the data from.

**datamanagement *(selection)*** Determines, how the data is represented internally.

**timestamp *(selection)*** Specifies whether to include an absolute timestamp, a timestamp relative to the beginning of the file (in seconds) or no timestamp at all.

# Read Excel




This operator reads an ExampleSet from the specified Excel file.

## Description

This operator can be used to load data from Microsoft Excel spreadsheets. This operator is able to read data from Excel 95, 97, 2000, XP, and 2003. The user has to define which of the spreadsheets in the workbook should be used as data table. The table must have a format such that each row is an example and each column represents an attribute. Please note that the first row of the Excel sheet might be used for attribute names which can be indicated by a parameter. The data table can be placed anywhere on the sheet and can contain arbitrary formatting instructions, empty rows and empty columns. Missing data values in Excel should be indicated by empty cells or by cells containing only "?".

For complete understanding of this operator read the parameters section. The easiest and shortest way to import an Excel file is to use the *import configuration wizard* from the Parameters panel. The best way, which may require some extra effort, is to first set all the parameters in the Parameters panel and then use the *wizard*. Please make sure that the Excel file is read correctly before building a process using it.

## Input Ports

**file *(fil)*** An Excel file is expected as a file object which can be created with other operators with file output ports like the Read File operator.

## Output Ports

**output *(out)*** This port delivers the Excel file in tabular form along with the meta data. This output is similar to the output of the Retrieve operator.

## Parameters

**import configuration wizard** This option allows you to configure this operator by means of a wizard. This user-friendly wizard makes the use of this operator easy.

**excel file** The path of the Excel file is specified here. It can be selected using the *choose a file* button.

**sheet selection** This option allows you to change the sheet selection between sheet number and sheet name.

**sheet number *(integer)*** The number of the sheet which you want to import should be specified here.

**sheet name *(string)*** The name of the sheet which you want to import should be specified here.

**imported cell range** This is a mandatory parameter. The range of cells to be imported from the specified sheet is given here. It is specified in 'xm:yn' format where 'x' is the column of the first cell of range, 'm' is the row of the first cell of range, 'y' is the column of the last cell of range, 'n' is the row of the last cell of range. 'A1:E10' will select all cells of the first five columns from row 1 to 10.

**first row as names *(boolean)*** If this option is set to true, it is assumed that the first line of the Excel file has the names of attributes. Then the attributes are automatically named and the first line of Excel file is not treated as a data line.

**annotations** If the *first row as names* parameter is not set to true, annotations can be added using the 'Edit List' button of this parameter which opens a new menu. This menu allows you to select any row and assign an annotation to it. *Name*, *Comment* and *Unit* annotations can be assigned. If row 0 is assigned *Name* annotation, it is equivalent to setting the *first row as names* parameter to true. If you want to ignore any rows you can annotate them as *Comment*.

**date format** The date and time format is specified here. Many predefined options exist; users can also specify a new format. If text in an Excel file column matches this date format, that column is automatically converted to *date* type. Some corrections are automatically made in the *date* type values. For example a value '32-March' will automatically be converted to '1-April'. Columns containing values which can't be interpreted as numbers will be interpreted as nominal, as long as they don't match the date and time pattern of the *date format* parameter. If they do, this column of the Excel file will be automatically parsed as *date* and the according attribute will be of *date* type.

**time zone** This is an expert parameter. A long list of time zones is provided; users can select any of them.

**locale** This is an expert parameter. A long list of locales is provided; users can select any of them.

**read all values as polynominal *(boolean)*** This option allows you to disable the type handling for this operator. Every column will be read as a polynominal attribute. To parse an excel date afterwards use 'date_parse(86400000 * (parse(date_attribute) - 25569))' (- 24107 for Mac Excel 2007) in the Generate Attributes operator.

**data set meta data information** This option is an important one. It allows you to adjust the meta data of the ExampleSet created from the specified Excel file. *Column index*, *name*, *type* and *role* can be specified here. The Read Excel operator tries to determine an appropriate type of the attributes by reading the first few lines and checking the occurring values. If all values are integers, the attribute will become an *integer*. Similarly if all values are real numbers, the attribute will become of type *real*. Columns containing values which can't be interpreted as numbers will be interpreted as nominal, as long as they don't match the date and time pattern of the *date format* parameter. If they do, this column of the Excel file will be automatically parsed as *date* and the according attribute will be of type *date*. Automatically determined types can be overridden using this parameter.

**read not matching values as missings *(boolean)*** If this value is set to true, values that do not match with the expected value type are considered as missing values and are replaced by '?'. For example if 'abc' is written in an *integer* column, it will be treated as a missing value. A question mark (?) or an empty cell in the Excel file is also read as a missing value.

**data management** This is an expert parameter. A long list is provided; users can select any option from this list.

## Tutorial Processes

### Reading an ExampleSet from an Excel file

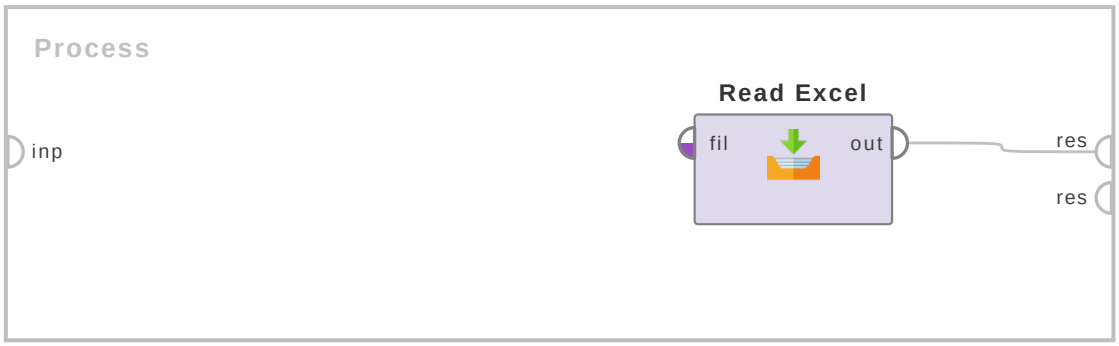
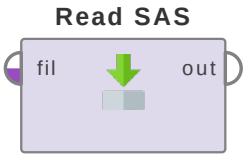



Figure 1.9: Tutorial process 'Reading an ExampleSet from an Excel file'.

For this Example Process you need an Excel file first. The one of this Example Process was created by copying the 'Golf' data set present in the Repositories into a new Excel file which was named 'golf'. The data set was copied on sheet 1 of the Excel file thus the sheet number parameter is given value 1. Make sure that you provide the correct location of the file in the Excel file parameter. The first cell of the sheet is A1 and last required cell is E15, thus the imported cell range parameter is provided value 'A1:E15'. As the first row of the sheet contains names of attributes, the first row as names parameter is checked. The remaining parameters were used with default values. Run the process, you will see almost the same results as you would have gotten from using the Retrieve operator to retrieve the 'Golf' data set from the Repository. You will see a difference in the meta data though, for example here the types and roles of attributes are different from those in the 'Golf' data set. You can change the role and type of attributes using the data set meta data information parameter. It is always good to make sure that all attributes are of desired role and type. In this example one important change that you would like to make is to change the role of the Play attribute. Its role should be changed to label if you want to use any classification operators on this data set.

# Read SAS




This operator is used for reading an SAS file.

## Description

This operator can read SAS (Statistical Analysis System) files. Please study the attached Example Process for understanding the use of this operator. Please note that when an SAS file is read, the roles of all the attributes are set to regular. Numeric columns use the "real" data type, nominal columns use the "polynominal" data type in RapidMiner.

## Input Ports

**file *(fil)*** An SAS file is expected as a file object which can be created with other operators with file output ports like the Read File operator.

## Output Ports

**output *(out)*** This port delivers the SAS file in tabular form along with the meta data. This output is similar to the output of the Retrieve operator.

## Parameters

**file *(filename)*** The path of the SAS file is specified here. It can be selected using the *choose a file* button.

## Tutorial Processes

### Use of the SAS operator

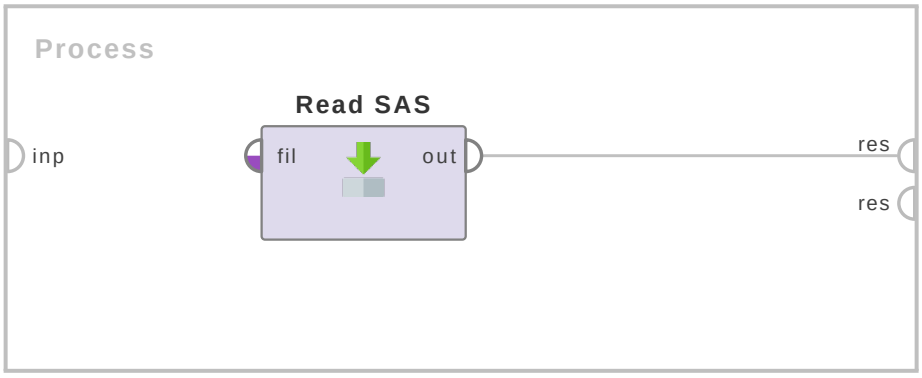



Figure 1.10: Tutorial process 'Use of the SAS operator'.

An SAS file is loaded using the Open File operator and then read via the Read SAS operator.

# Read SPSS


This operator is used for reading SPSS files.

## Description

The Read SPSS operator can read the data files created by SPSS (Statistical Package for the Social Sciences), an application used for statistical analysis. SPSS files are saved in a proprietary binary format and contain a dataset as well as a dictionary that describes the dataset. These files save data by 'cases' (rows) and 'variables' (columns).

These files have a '.SAV' file extension. SAV files are often used for storing datasets extracted from databases and Microsoft Excel spreadsheets. SPSS datasets can be manipulated in a variety of ways, but they are most commonly used to perform statistical analysis tests such as regression analysis, analysis of variance, and factor analysis.

## Input Ports

**file *(fil)*** This optional port expects a file object.

## Output Ports

**output *(out)*** Data from the SPSS file is delivered through this port mostly in form of an ExampleSet.

## Parameters

**filename *(filename)*** This parameter specifies the path of the SPSS file. It can be selected using the *choose a file* button.

**datamanagement *(selection)*** This parameter determines how the data is represented internally. This is an expert parameter. There are different options, users can choose any of them.

**attribute naming mode *(selection)*** This parameter determines which SPSS variable properties should be used for naming the attributes.

**use value labels *(boolean)*** This parameter specifies if the SPSS value labels should be used as values.

**recode user missings *(boolean)*** This parameter specifies if the SPSS user missings should be recoded to missing values.

**sample ratio *(real)*** This parameter specifies the fraction of the data set which should be read. If it is set to 1, the complete data set is read. If it is set to -1 then the *sample size* parameter is used for determining the size of the data to read.

**sample size *(integer)*** This parameter specifies the exact number of samples which should be read. If it is set to -1, then the *sample ratio* parameter is used for determining the size of data to read. If both are set to -1 then the complete data set is read.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

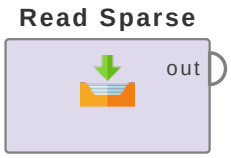
## Tutorial Processes

### Reading an SPSS file


Figure 1.11: Tutorial process 'Reading an SPSS file'.

You need to have an SPSS file for this process. In this process, the name of the SPSS file is airline_passengers.sav and it is placed in the D drive of the computer. The file is read using the Read SPSS operator. All parameters are used with default values. After execution of the process you can see the resultant ExampleSet in the Results Workspace.

# Read Sparse




This operator is used for reading files written in sparse formats.

## Description

This operator reads sparse format files. The lines of a sparse file have the form:
*label index:value index:value index:value...*
Where *index* may be an integer (starting with 1) for the regular attributes or one of the prefixes specified by the *prefix map* parameter. The following formats are supported:

- xy format: The label is the last token in each line.
- yx format: The label is the first token in each line.
- prefix format: The label is prefixed by 'l:'
- separate file format: The label is read from a separate file specified by the *label file* parameter.
- no label: The ExampleSet is unlabeled.

## Output Ports

**output** ***(out)*** This port delivers the required file in tabular form along with the meta data. This output is similar to the output of the Retrieve operator.

## Parameters

**format** ***(selection)*** This parameter specifies the format of the sparse data file.

**attribute description file** ***(filename)*** The name of the attribute description file is specified here. An attribute description file (extension: .aml) is required to retrieve meta data of the ExampleSet. This file is a simple XML document defining the properties of the attributes (like their name and range) and their source files. The data may be spread over several files. This file also contains the names of the files to read the data from. Therefore, the actual data files do not have to be specified as a parameter of this operator.

**data file** ***(filename)*** This parameter specifies the name of the data file. It is necessary if it is not specified in the attribute description file.

**label file** ***(filename)*** This parameter specifies the name of the file containing the labels. It is necessary if the *format* parameter is set to 'format separate file'

**dimension** ***(integer)*** This parameter specifies the dimension of the example space. It is necessary if the *attribute description file* parameter is not set.

**sample size** ***(integer)*** This parameter specifies the maximum number of examples which should be read. If it is set to -1, then all examples are read.

**use quotes *(boolean)*** This parameter indicates if quotes should be regarded. If this option is set to true, the *quotes character* parameter can be used for specifying the quotes character.

**quotes character *(char)*** This parameter defines the *quotes character*.

**datamanagement *(selection)*** This parameter determines how the data is represented internally. This is an expert parameter. There are different options, users can choose any of them.

**decimal point character *(string)*** This character is used as the decimal character.

**prefix map *(list)*** This parameter maps prefixes to names of special attributes.

**encoding *(selection)*** This is an expert parameter. A long list of encoding is provided; users can select any one of them.

## Tutorial Processes

### Writing and Reading a sparse file

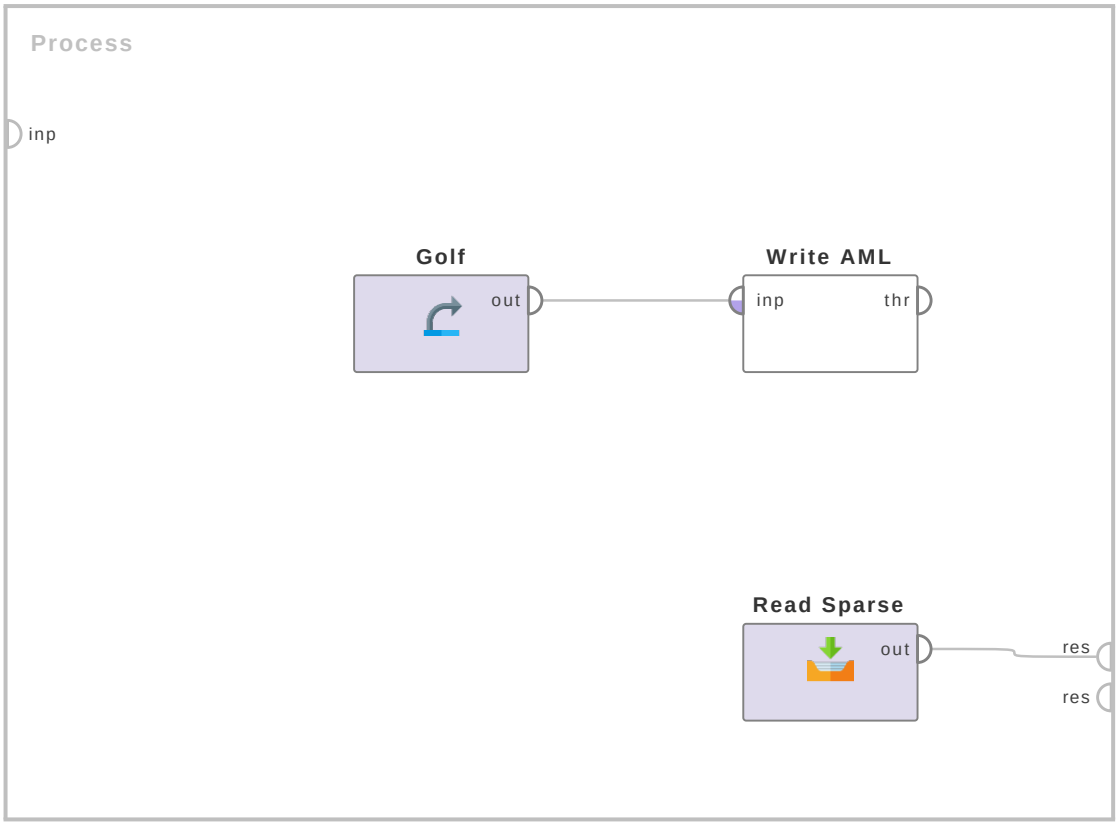



Figure 1.12: Tutorial process 'Writing and Reading a sparse file'.

This Example Process shows the Write AML operator can be used for writing a sparse file and how the Read Sparse operator can be used for reading a sparse file. The 'Golf' data set is loaded

using the Retrieve operator. This ExampleSet is provided as input to the Write AML operator. The example set file parameter is set to 'D:\golf_data' thus a file named 'golf_data' is created (if it does not already exist) in the 'D' drive of your computer. You can open the written file and make changes in it (if required). This file has the instances of the ExampleSet. The attribute description file parameter is set to 'D:\golf_att' thus a file named 'golf_att' is created (if it does not already exist) in the 'D' drive of your computer. You can open the written file and make changes in it (if required). This file has the meta data of the ExampleSet. The format parameter is set to 'sparse_xy' to write the file in xy sparse format. The Read Sparse operator is applied next to read the ExampleSet from the files. The attribute description file and data file parameters are set to 'D:\golf_att' and 'D:\golf_data' respectively. The format parameter is set to 'xy' because the file was written in xy format. All other parameters are used with default values. The resultant ExampleSet can be seen in the Results Workspace.

# Read Stata


This operator can read Stata data files.

## Description

This operator can read Stata files. Currently only Stata files of version 113 or 114 are supported.

## Input Ports

**file *(fil)*** This optional port expects a file object.

## Output Ports

**output *(out)*** Data from the Stata file is delivered through this port mostly in form of an ExampleSet.

## Parameters

**filename *(filename)*** Name of the file to read the data from.

**datamanagement *(selection)*** Determines, how the data is represented internally.

**attribute naming mode *(selection)*** Determines which variable properties should be used for attribute naming.

**handle value labels *(selection)*** Specifies how to handle attributes with value labels, i.e. whether to ignore the labels or how to use them.

**sample ratio *(real)*** The fraction of the data set which should be read (1 = all; only used if sample_size = -1)

**sample size *(integer)*** The exact number of samples which should be read (-1 = all; if not -1, sample_ratio will not have any effect)

**use local random seed *(boolean)*** Indicates if a local random seed should be used.

**local random seed *(integer)*** Specifies the local random seed

# Read XML


This operator is used for reading an XML file.

## Description

This operator can read XML files, where examples are represented by elements which match a given XPath and features are attributes and text-content of each element and its sub-elements.

This operator tries to determine an appropriate type of the attributes by reading the first few elements and checking the occuring values. If all values are integers, the attribute will become integer, if real numbers occur, it will be of type real. Columns containing values which can't be interpreted as numbers will be nominal, as long as they don't match the date and time pattern of the *date format* parameter. If they do, this attribute will be automatically parsed as date and the according feature will be of type date.

## Input Ports

**file *(fil)*** An XML file is expected as a file object which can be created with other operators with file output ports like the Read File operator.

## Output Ports

**output *(out)*** This port delivers the XML file in tabular form along with the meta data. This output is similar to the output of the Retrieve operator.

## Parameters

**parse numbers *(boolean)*** Specifies whether numbers are parsed or not.

**decimal character *(char)*** This character is used as the decimal character.

**grouped digits *(boolean)*** This option decides whether grouped digits should be parsed or not. If this option is set to true, a *grouping character* parameter should be specified.

**grouping character *(char)*** This character is used as the grouping character. If this character is found between numbers, the numbers are combined and this character is ignored. For example if "22-14" is present in the CSV file and "-" is set as *grouping character*, then "2214" will be stored.

**infinity string *(string)*** This parameter can be set to parse a specific infinity representation (e.g. "Infinity"). If it is not set, the local specific infinity representation will be used.

**date format *(string)*** The date and time format is specified here. Many predefined options exist; users can also specify a new format. If text in a CSV file column matches this date format, that column is automatically converted to *date* type. Some corrections are automatically made in *date* type values. For example a value '32-March' will automatically be converted to '1-April'. Columns containing values which can't be interpreted as numbers will be interpreted as nominal, as long as they don't match the date and time pattern of the

*date format* parameter. If they do, this column of the CSV file will be automatically parsed as *date* and the according attribute will be of *date* type.

**first row as names *(boolean)*** If this option is set to true, it is assumed that the first line of the CSV file has the names of the attributes. Then the attributes are automatically named and first line of the CSV file is not treated as a data line.

**annotations *(menu)*** If first row as names is not set to true, annotations can be added using the 'Edit List' button of this parameter which opens a new menu. This menu allows you to select any row and assign an annotation to it. *Name*, *Comment* and *Unit* annotations can be assigned. If row 0 is assigned a *Name* annotation, it is equivalent to setting the *first row as names* parameter to true. If you want to ignore any rows you can annotate them as *Comment*. Remember row number in this menu does not count commented lines.

**time zone *(selection)*** This is an expert parameter. A long list of time zones is provided; users can select any of them.

**locale *(selection)*** This is an expert parameter. A long list of locales is provided; users can select any of them.

**read all values as polynominal *(boolean)*** This option allows you to disable the type handling for this operator. Every xpath entry will be read as a polynominal attribute.

**data set meta data information *(menu)*** This option is an important one. It allows you to adjust the meta data of the CSV file. Column index, name, type and role can be specified here. The Read CSV operator tries to determine an appropriate type of the attributes by reading the first few lines and checking the occurring values. If all values are integers, the attribute will become an integer. Similarly if all values are real numbers, the attribute will become of type real. Columns containing values which can't be interpreted as numbers will be interpreted as nominal, as long as they don't match the date and time pattern of the *date format* parameter. If they do, this column of the CSV file will be automatically parsed as date and the according attribute will be of type *date*. Automatically determined types can be overridden using this parameter.

**read not matching values as missings *(boolean)*** If this value is set to true, values that do not match with the expected value type are considered as missing values and are replaced by '?'. For example if 'abc' is written in an integer column, it will be treated as a missing value. A question mark (?) in the CSV file is also read as a missing value.

**datamanagement *(selection)*** This is an expert parameter. A long list is provided; users can select any option from this list.

# Read XRFF


This operator is used for reading XRFF (eXtensible attribute-Relation File Format) files.

## Description

This operator can read XRFF files known from Weka. The XRFF (eXtensible attribute-Relation File Format) is an XML-based extension of the ARFF format in some sense similar to the original RapidMiner file format for attribute description files (.aml). You can see a sample XRFF file by studying the attached Example Process.

Since the XML representation takes up considerably more space because the data is wrapped into XML tags, one can also compress the data via gzip. RapidMiner automatically recognizes a file being gzip compressed, if the file's extension is .xrff.gz instead of .xrff.

The XRFF file is divided into two portions i.e. the header and the body. The header has the meta data description and the body has the instances. Via the class="yes" attribute in the attribute specification in the header, one can define which attribute should be used as a prediction label attribute. Although the RapidMiner terminology for such classes is "label" instead of "class" we support the terminology class in order to have compatibility with the original XRFF files.

## Input Ports

**file *(fil)*** This optional port expects a file object.

## Output Ports

**output *(out)*** The XRFF file is read from the specified path and the resultant ExampleSet is delivered through this port.

## Parameters

**data file *(filename)*** This parameter specifies the path of the XRFF file. It can be selected using the *choose a file* button.

**id attribute *(string)*** This parameter specifies the name of the id attribute. Please note that this field is case-sensitive.

**datamanagement *(selection)*** This parameter determines how the data is represented internally. This is an expert parameter. There are different options, users can choose any of them.

**decimal point character *(string)*** This parameter specifies the character that is used as decimal point.

**sample ratio *(real)*** This parameter specifies the fraction of the data set which should be read. If it is set to 1, the complete data set is read. If it is set to -1 then the *sample size* parameter is used for determining the size of the data to read.

**sample size *(integer)*** This parameter specifies the exact number of samples which should be read. If it is set to -1 the *sample ratio* parameter is used for determining the size of data to read. If both are set to -1 the complete data set is read.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Writing and Reading an XRFF file

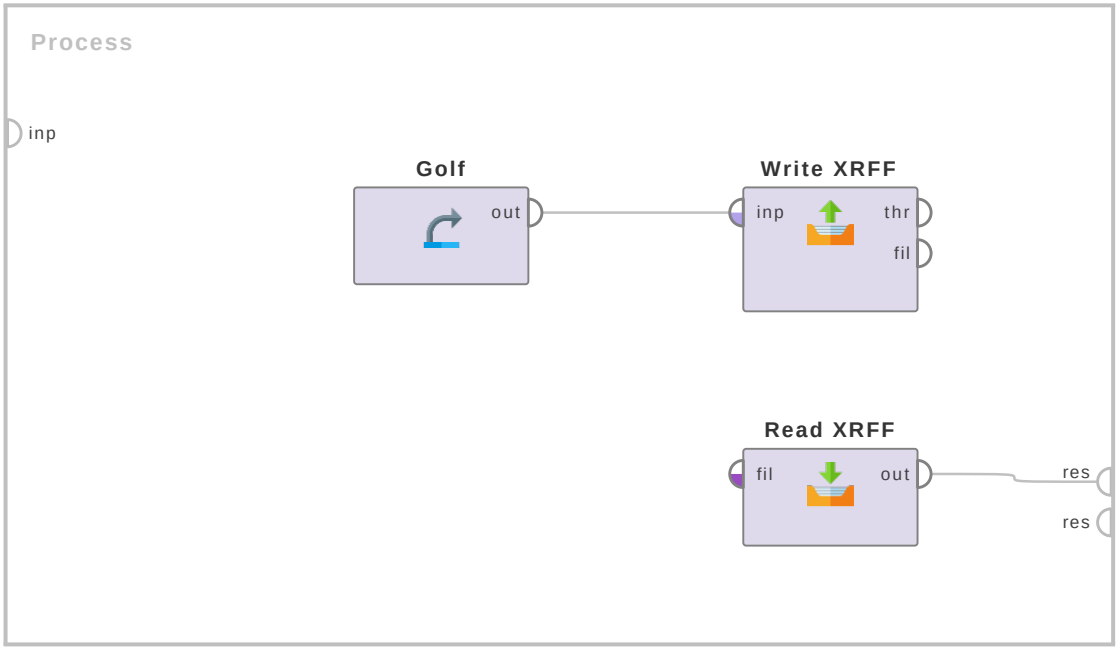
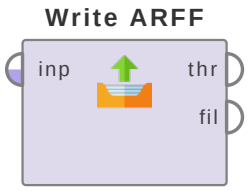



Figure 1.13: Tutorial process ‘Writing and Reading an XRFF file’.

This Example Process demonstrates the use of the Write XRFF and Read XRFF operators respectively. This Example Process shows how these operators can be used to write and read an ExampleSet. The ‘Golf’ data set is loaded using the Retrieve operator. This ExampleSet is provided as input to the Write XRFF operator. The example set file parameter is set to ‘D:\golf_xrff’ thus a file named ‘golf_xrff’ is created (if it does not already exist) in the ‘D’ drive of your computer. You can open the written file and make changes in it (if required). The Read XRFF operator is applied next. The data file parameter is set to ‘D:\golf_xrff’ to read the file that was just written using the Write XRFF operator. The remaining parameters are used with default values. The resultant ExampleSet can be seen in the Results Workspace.

## 1.1.2 Write

### Write ARFF




This operator is used for writing an ARFF file.

#### Description

This operator can write data in form of ARFF (Attribute-Relation File Format) files known from the machine learning library Weka. An ARFF file is an ASCII text file that describes a list of instances sharing a set of attributes. ARFF files were developed by the Machine Learning Project at the Department of Computer Science of The University of Waikato for use with the Weka machine learning software. Please study the attached Example Processes for understanding the basics and structure of the ARFF file format. Please note that when an ARFF file is written, the roles of the attributes are not stored. Similarly when an ARFF file is read, the roles of all the attributes are set to regular.

#### Input Ports

**input *(inp)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

#### Output Ports

**through *(thr)*** The ExampleSet that was provided at the input port is delivered through this output port without any modifications. This is usually used to reuse the same ExampleSet in further operators of the process.

**file *(fil)*** This port buffers the file object for passing it to the reader operators

#### Parameters

**example set file *(filename)*** The path of the ARFF file is specified here. It can be selected using the *choose a file* button.

**encoding *(selection)*** This is an expert parameter. A long list of encoding is provided; users can select any of them.

#### Tutorial Processes

##### The basics of ARFF

The 'Iris' data set is loaded using the Retrieve operator. The Write ARFF operator is applied on it to write the 'Iris' data set into an ARFF file. The example set file parameter is set to 'D:\Iris.txt'. Thus an ARFF file is created in the 'D' drive of your computer with the name 'Iris'. Open this file to see the structure of an ARFF file.

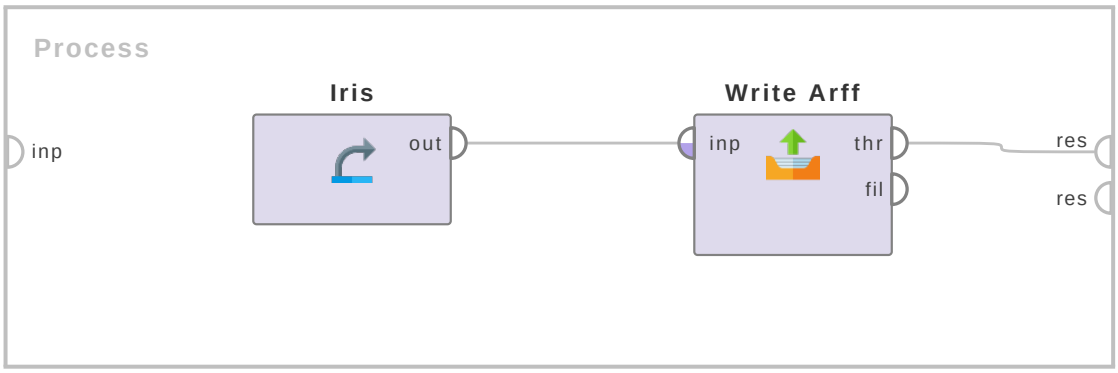



Figure 1.14: Tutorial process 'The basics of ARFF'.

ARFF files have two distinct sections. The first section is the Header information, which is followed by the Data information. The Header of the ARFF file contains the name of the Relation and a list of the attributes. The name of the Relation is specified after the @RELATION statement. The Relation is ignored by RapidMiner. Each attribute definition starts with the @ATTRIBUTE statement followed by the attribute name and its type. The resultant ARFF file of this Example Process starts with the Header. The name of the relation is 'RapidMinerData'. After the name of the Relation, six attributes are defined.

Attribute declarations take the form of an ordered sequence of @ATTRIBUTE statements. Each attribute in the data set has its own @ATTRIBUTE statement which uniquely defines the name of that attribute and its data type. The order of declaration of the attributes indicates the column position in the data section of the file. For example, in the resultant ARFF file of this Example Process the 'label' attribute is declared at the end of all other attribute declarations. Therefore values of the 'label' attribute are in the last column of the Data section.

The possible attribute types in ARFF are: numeric integer real {nominalValue1,nominalValue2,...} for nominal attributes string for nominal attributes without distinct nominal values (it is however recommended to use the nominal definition above as often as possible) date [date-format] (currently not supported by RapidMiner)

You can see in the resultant ARFF file of this Example Process that the attributes 'a1', 'a2', 'a3' and 'a4' are of real type. The attributes 'id' and 'label' are of nominal type. The distinct nominal values are also specified with these nominal attributes.

The ARFF Data section of the file contains the data declaration line @DATA followed by the actual example data lines. Each example is represented on a single line, with carriage returns denoting the end of the example. Attribute values for each example are delimited by commas. They must appear in the order that they were declared in the Header section (i.e. the data corresponding to the n-th @ATTRIBUTE declaration is always the n-th field of the example line). Missing values are represented by a single question mark (?).

A percent sign (%) introduces a comment and will be ignored during reading. Attribute names or example values containing spaces must be quoted with single quotes ('). Please note that in RapidMiner the sparse ARFF format is currently only supported for numerical attributes. Please use one of the other options for sparse data files provided by RapidMiner if you also need sparse data files for nominal attributes.

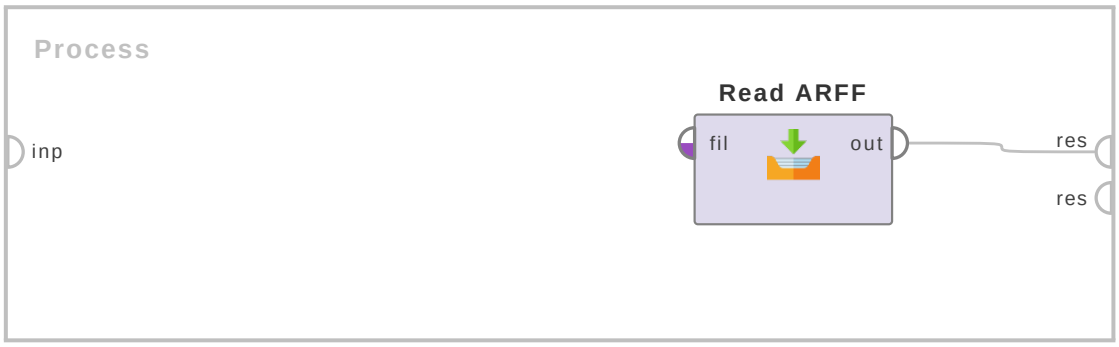
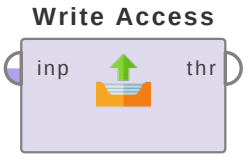



Figure 1.15: Tutorial process 'Reading an ARFF file using the Read ARFF operator'.

### Reading an ARFF file using the Read ARFF operator

The ARFF file that was written in the first Example Process using the Write ARFF operator is retrieved in this Example Process using the Read ARFF operator. The data file parameter is set to 'D:\Iris.txt'. Please make sure that you specify the correct path. All other parameters are used with default values. Run the process. You will see that the results are very similar to the original Iris data set of RapidMiner repository. Please note that the role of all the attributes is regular in the results of the Read ARFF operator. Even the roles of 'id' and 'label' attributes are set to regular. This is so because the ARFF files do not store information about the roles of the attributes.

# Write Access




This operator writes an ExampleSet into the specified Microsoft Access database.

## Description

The Write Access operator is used for writing an ExampleSet into the specified Microsoft Access database (.mdb or .accdb extension) using the UCanAccess jdbc driver. You only need to have a basic understanding of databases in order to use this operator properly. Please go through the parameters and the attached Example Process to understand the working of this operator.

## Input Ports

**input *(inp)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**through *(thr)*** The ExampleSet that was provided at the input port is delivered through this output port without any modifications. This is usually used to reuse the same ExampleSet in further operators of the process.

**file *(fil)*** This port memory buffers file object for passing it to the reader operators

## Parameters

**database file *(filename)*** This parameter specifies the path of the Access database ( i.e. the mdb or accdb file)

**username *(string)*** This parameter is used for specifying the username of the database (if any).

**password *(string)*** This parameter is used for specifying the password of the database (if any).

**table name *(string)*** This parameter is used for specifying the name of the required table from the specified database.

**overwrite mode *(selection)*** This parameter indicates if an existing table should be overwritten or the data should be appended.

**access version *(selection)*** If a new database is created this parameter specifies its format version. This parameter is not used if the database already exists.

## Tutorial Processes

### Writing and then reading data from an Access database

The 'Golf' data set is loaded using the Retrieve operator. The Write Access operator is used for writing this ExampleSet into the golf table of the 'golf_db.mdb' database. The database file parameter is provided with the path of the database file 'golf_db.mdb' and the name of the desired

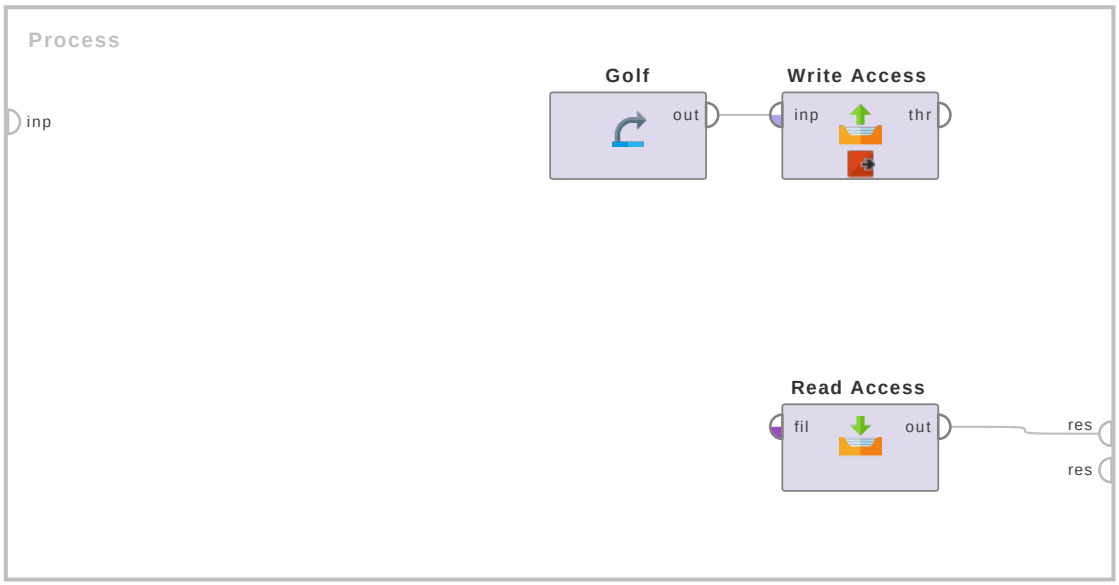
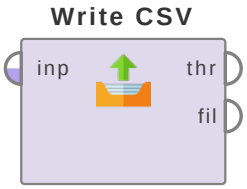



Figure 1.16: Tutorial process 'Writing and then reading data from an Access database'.

table is specified in the table name parameter ( i.e. it is set to 'golf'). A breakpoint is inserted here. No results are visible in RapidMiner at this stage but you can see that at this point of the execution the database has been created and the golf table has been filled with the examples of the 'Golf' data set.

Now the Read Access operator is used for reading the golf table from the 'golf_db.mdb' database. The database file parameter is provided with the path of the database file 'golf_db.mdb'. The define query parameter is set to 'table name'. The table name parameter is set to 'golf' which is the name of the required table. Continue the process, you will see the entire golf table in the Results Workspace. The define query parameter is set to 'table name' if you want to read an entire table from the database. You can also read a selected portion of the database by using queries. Set the define query parameter to 'query' and specify a query in the query parameter.

# Write CSV




This operator is used to write CSV files(Comma-Separated Values).

## Description

A comma-separated values (CSV) file stores tabular data (numbers and text) in plain-text form. CSV files have all values of an example in one line. Values for different attributes are separated by a constant separator. It may have many rows. Each row uses a constant separator for separating attribute values. The name suggests that the attributes values would be separated by commas, but other separators can also be used. This separator can be specified using the *column separator* parameter. Missing data values are indicated by empty cells.

## Input Ports

**input *(inp)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**through *(thr)*** The ExampleSet that was provided at the input port is delivered through this output port without any modifications.This is usually used to reuse the same ExampleSet in further operators of the process.

**file *(fil)*** The created CSV file is provided as a file object that can be used with other operators with file input ports like 'Write File'.

## Parameters

**csv file *(filename)*** path of the CSV file is specified here. It can be selected using the *choose a file* button.

**column separator *(string)*** Column separators for the CSV file can be specified here.

**write attribute names *(boolean)*** This parameter indicates if the attribute names should be written as the first row of the CSV file.

**quote nominal values *(boolean)*** This parameter indicates if the nominal values should be quoted with double quotes in the CSV file.

**format date attributes *(boolean)*** This parameter indicates if the date attributes should be written as a formatted string or as milliseconds past since January 1, 1970, 00:00:00 GMT.

**append to file *(boolean)*** This parameter indicates if new content should be appended to the file or if the pre-existing file content should be overwritten.

**encoding *(selection)*** This is an expert parameter. There are different options, users can choose any of them.

## Tutorial Processes

### Writing the Labor-Negotiations data set into a CSV file

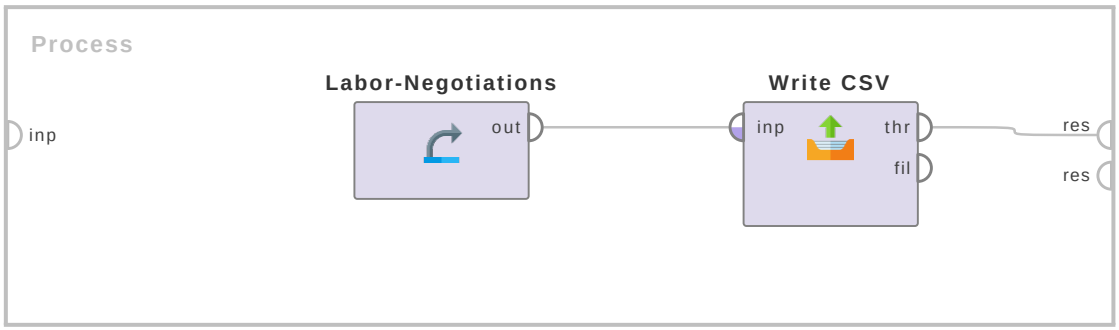
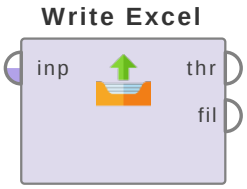



Figure 1.17: Tutorial process 'Writing the Labor-Negotiations data set into a CSV file'.

The 'Labor-Negotiations' data set is loaded using the Retrieve operator. The Write CSV operator is applied on it to write the 'Labor-Negotiations' data set in a CSV file. The csv file parameter is provided with this path: 'D:\Labor data set'. Thus a CSV file named 'Labor data set' is created in the 'D' drive of your computer. All parameters are used with default values. The write attribute names parameter is set to true thus the first line of the resultant CSV file has the names of the attributes of the 'Labor-Negotiations' data set. The quote nominal values parameter is also set to true, thus all nominal values are quoted with double quotes in the CSV file. Files written by the Write CSV operator can be loaded in RapidMiner using the Read CSV operator.

# Write Excel




This operator writes an ExampleSet to a Excel spreadsheet file.

## Description

The Write Excel operator can be used for writing an ExampleSet into a Microsoft Excel spreadsheet. This operator creates Excel files that are readable by Excel 95, 97, 2000, XP, 2003 and newer versions. Missing data values in the ExampleSet are indicated by empty cells in the Excel spreadsheet. The first row of the resultant Excel file has the names of attributes of the input ExampleSet. Files written by the Write Excel operator can be loaded in RapidMiner using the Read Excel operator.

## Input Ports

**input *(inp)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**through *(thr)*** The ExampleSet that was provided at the input port is delivered through this output port without any modifications.This is usually used to reuse the same ExampleSet in further operators of the process.

**file *(fil)*** The created Excel file is provided as a file object that can be used with other operators with file input ports like 'Write File'.

## Parameters

**excel file *(string)*** The path of the Excel file is specified here. It can be selected using the *choose a file* button.

**file format *(selection)*** Allows the user to specify if the resulting excel sheet should have the xls or xlsx format.

**encoding *(selection)*** This is an expert parameter furthermore it is shown with file format xls only. There are different options, users can choose any of them.

**sheet name *(string)*** This parameter is shown with file format xlsx only. The user can specify the name of the excel sheet.

**date format *(string)*** This is an expert parameter furthermore it is shown with file format xlsx only. Format dates should be saved in.

**number format *(string)*** This is an expert parameter furthermore it is shown with file format xlsx only. Format number should be saved in.

## Tutorial Processes

### Writing the Labor-Negotiations data set into an Excel file

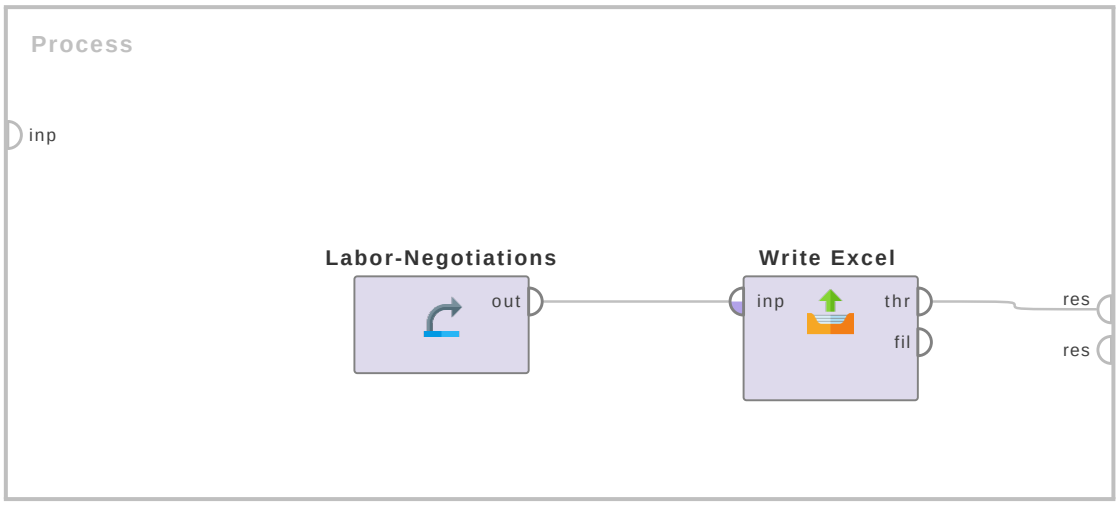
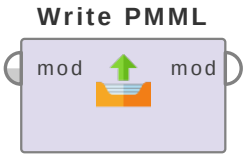



Figure 1.18: Tutorial process 'Writing the Labor-Negotiations data set into an Excel file'.

The Labor-Negotiations data set is loaded using the Retrieve operator. The Write Excel operator is applied on it to write the Labor-Negotiations data set in a Excel file. The excel file parameter is provided with this path: 'D:\Labor data set.xls'. Thus an Excel file named 'Labor data set' is created in the 'D' drive of your computer. Note that the first row of the resultant Excel file has the names of attributes of the Labor-Negotiations data set. Also note that all missing values in the Labor-Negotiations data set are represented by empty cells in the Excel file.

# Write PMML




This operator will save the given model to an XML file of PMML 4.0 format.

## Description

This operator will write the given model to an XML file of PMML 4.0 format. This format is a standard for data mining models and is understood by many data bases. It can be used for applying data mining models directly in the database. This way it can be applied on a regular basis on huge amounts of data.

This operator supports the following models:

- Decision Tree Models
- Rule Models
- Naive Bayes models for nominal attributes
- Linear Regression Models
- Logistic Regression Models
- Centroid based Cluster models like models of k-means and k-medoids

## Input Ports

**model input *(mod)*** The model input port.

## Output Ports

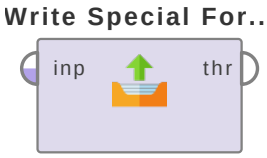
**model output *(mod)*** The model output port.

## Parameters

**file** Specifies the file for saving the pmml.

**version** Determines which PMML version should be used for export.

# Write Special Format




This operator writes an ExampleSet or subset of an ExampleSet in a special user defined format.

## Description

The path of the file is specified through the *example set file* parameter. The *special format* parameter is used for specifying the exact format. The character following the $ character introduces a command. Additional arguments to this command may be supplied by enclosing them in square brackets. The following commands can be used in the *special format parameter*:

- $a : This command writes all attributes separated by the default separator.
- $a[separator] : This command writes all attributes separated by a separator (the separator is specified as an argument in brackets).
- $s[separator][indexSeparator] : This command writes in sparse format. The separator and indexSeparator are provided as first and second arguments respectively. For all non zero attributes the following strings are concatenated: the column index, the value of the indexSeparator, the attribute value. The attributes are separated by the specified separator.
- $v[name] : This command writes the values of a single attribute. The attribute name is specified as an argument. This command can be used for writing both regular and special attributes.
- $k[index] : This command writes the values of a single attribute. The attribute index is specified as an argument. The indices start from 0. This command can be used for writing only regular attributes.
- $l : This command writes the values of the label attribute.
- $p : This command writes the values of the predicted label attribute.
- $d : This command writes all prediction confidences for all classes in the form ‘conf(class)=value’
- $d[class] : This command writes the prediction confidences for the defined class as a simple number. The required class is provided as an argument.
- $i : This command writes the values of the id attribute.
- $w : This command writes the example weights.
- $b : This command writes the batch number.
- $n : This command writes the newline character i.e. newline is inserted when this character is reached.
- $t : This command writes the tabulator character i.e. tab is inserted when this character is reached.
- $$ : This command writes the dollar sign.

- $[ : This command writes the '[' character i.e. the opening square bracket.
- $] : This command writes the ']' character i.e. the closing square bracket.

Please Make sure that the format string ends with $n or the *add line separator* parameter is set to true if you want examples to be separated by newlines.

## Input Ports

**input *(inp)*** This input port expects an ExampleSet. It is output of the Apply Model operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**through *(thr)*** The ExampleSet that was provided at the input port is delivered through this output port without any modifications. This is usually used to reuse the same ExampleSet in further operators of the process.

## Parameters

**example set file *(filename)*** The ExampleSet is written into the file specified through this parameter.

**special format *(string)*** This parameter specifies the exact format of the file. Many commands are available for specifying the format. These commands are discussed in the description of this operator.

**fraction digits *(integer)*** This parameter specifies the number of fraction digits in the output file. This parameter is used for rounding off real numbers. Setting this parameter to -1 will write all possible digits i.e. no rounding off is done.

**quote nominal values *(boolean)*** This parameter indicates if nominal values should be quoted with double quotes.

**add line separator *(boolean)*** This parameter indicates if each example should be followed by a line break or not . If set to true, each example is followed by a line break automatically.

**zipped *(boolean)*** This parameter indicates if the data file content should be zipped or not.

**overwrite mode *(selection)*** This parameter indicates if an existing file should be overwritten or data should be appended.

**encoding *(selection)*** This is an expert parameter. There are different options, users can choose any of them

## Tutorial Processes

### Writing labeled data set in a user-defined format

The k-NN classification model is trained on the 'Golf' data set. The trained model is then applied on the 'Golf-Testset' data set using the Apply Model operator. The resulting labeled data set is written in a file using the Write Special Format operator. Have a look at the parameters of the Write Special Format operator. You can see that the ExampleSet is written into a file named

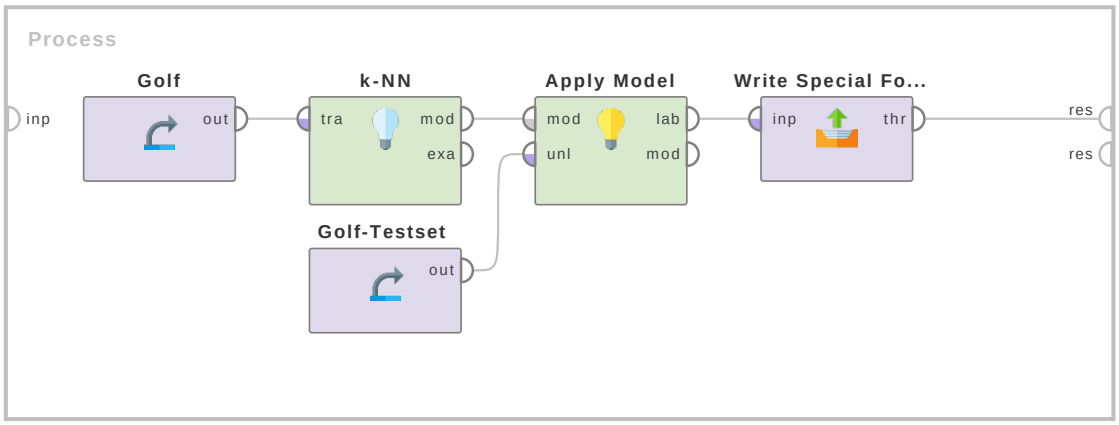
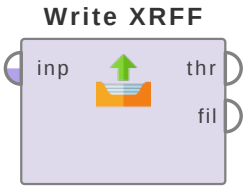



Figure 1.19: Tutorial process ‘Writing labeled data set in a user-defined format’.

‘special’. The special format parameter is set to ‘ $[ $l $] $t $p $t $d[yes] $t $d[no]’. This format string is composed of a number of commands, it can be interpreted as: ‘[label] predicted_label confidence (yes) confidence (no)’. This format string states that four attributes shall be written in the file i.e. ‘label’, ‘predicted label’, ‘confidence (yes)’ and ‘confidence (no)’. Each attribute should be separated by a tab. The label attribute should be enclosed in square brackets. Run the process and see the written file for verification.

# Write XRFF




Writes the values of all examples into an XRFF-file.

## Description

Writes values of all examples into an XRFF file which can be used by the machine learning library Weka. The XRFF format is described in the XrffExampleSource operator which is able to read XRFF files to make them usable with RapidMiner.

Please note that writing attribute weights is not supported, please use the other RapidMiner operators for attribute weight loading and writing for this purpose.

## Input Ports

**input *(inp)*** This input port expects an ExampleSet.

## Output Ports

**through *(thr)*** The ExampleSet that was provided at the input port is delivered through this output port without any modifications. This is usually used to reuse the same ExampleSet in further operators of the process.

**file *(fil)*** This port buffers the file object for passing it to the reader operators

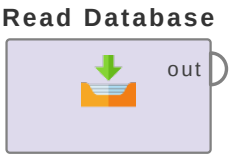
## Parameters

**example set file *(filename)*** The path of the XRFF file is specified here. It can be selected using the *choose a file* button.

**encoding *(selection)*** This is an expert parameter. A long list of encoding is provided; users can select any of them.

# 1.2 Database

## Read Database




This operator reads an ExampleSet from a SQL database.

### Description

The Read Database operator is used for reading an ExampleSet from the specified SQL database. You need to have at least basic understanding of databases, database connections and queries in order to use this operator properly. Go through the parameters and Example Process to understand the flow of this operator.

When this operator is executed, the table delivered by the query will be copied into the memory of your computer. This will give all subsequent operators a fast access on the data. Even learning schemes like the Support Vector Machine with their high number of random accesses will run fast.

The java *ResultSetMetaData* interface does not provide information about the possible values of nominal attributes. The internal indices the nominal values are mapped to, will depend on the ordering they appear in the table. This may cause problems only when processes are split up into a training process and a testing process. This is not a problem for learning schemes which are capable of handling nominal attributes. If a learning scheme like the SVM is used with nominal data, RapidMiner pretends that nominal attributes are numerical and uses indices for the nominal values as their numerical value. The SVM may perform well if there are only two possible values. If a test set is read in another process, the nominal values may be assigned different indices, and hence the SVM trained is useless. This is not a problem for the label attributes, since the classes can be specified using the *classes* parameter and hence all learning schemes intended to use with nominal data are safe to use. You might avoid this problem if you first combine both ExampleSets using the Append operator and then split it again using two Filter Examples operators.

### Differentiation

- **Execute SQL** The Read Database operator is used for loading data from a database into RapidMiner. The Execute SQL operator cannot be used for loading data from databases. It can be used for executing SQL statements like CREATE or ADD etc on the database. See page 834 for details.

### Output Ports

**output *(out)*** This port delivers the result of the query on database in tabular form along with the meta data. This output is similar to the output of the Retrieve operator.

### Parameters

**define connection *(selection)*** This parameter indicates how the database connection should be specified. It gives you three options: predefined, url and jndi.

**connection** ***(string)*** This parameter is only available when the *define connection* parameter is set to *predefined*. This parameter is used to connect to the database using a predefined connection. You can have many predefined connections. You can choose one of them using the drop down box. You can add a new connection or modify previous connections using the button next to the drop down box. You may also accomplish this by clicking on the *Manage Database Connections...* from the *Tools* menu in the main window. A new window appears. This window asks for several details e.g. *Host, Port, Database system, schema, username* and *password*. The *Test* button in this new window will allow you to check whether the connection can be made. Save the connection once the test is successful. After saving a new connection, it can be chosen from the drop down box of the *connection* parameter. You need to have basic understanding of databases for configuring a connection.

**database system** ***(selection)*** This parameter is only available when the *define connection* parameter is set to *url*. This parameter is used to select the database system in use. It can have one of the following values: MySQL, PostgreSQL, Sybase, HSQLDB, ODBC Bridge (e.g. Access), Microsoft SQL Server (JTDS), Ingres, Oracle.

**database url** ***(string)*** This parameter is only available when the *define connection* parameter is set to *url*. This parameter is used to define the URL connection string for the database, e.g. 'jdbc:mysql://foo.bar:portnr/database'.

**username** ***(string)*** This parameter is only available when the *define connection* parameter is set to *url*. This parameter is used to specify the username of the database.

**password** ***(string)*** This parameter is only available when the *define connection* parameter is set to *url*. This parameter is used to specify the password of the database.

**jndi name** ***(string)*** This parameter is only available when the *define connection* parameter is set to *jndi*. This parameter is used to give the JNDI a name for a data source.

**define query** ***(selection)*** Query is a statement that is used to select required data from the database. This parameter specifies whether the database query should be defined directly, through a file or implicitly by a given table name. The SQL query can be auto generated giving a table name, passed to RapidMiner via a parameter or, in case of long SQL statements, in a separate file. The desired behavior can be chosen using the *define query* parameter. Please note that column names are often case sensitive and might need quoting.

**query** ***(string)*** This parameter is only available when the *define query* parameter is set to *query*. This parameter is used to define the SQL query to select desired data from the specified database.

**query file** ***(filename)*** This parameter is only available when the *define query* parameter is set to *query file*. This parameter is used to select a file that contains the SQL query to select desired data from the specified database. Long queries are usually stored in files. Storing queries in files can also enhance reusability.

**table name** ***(string)*** This parameter is only available when the *define query* parameter is set to *table name*. This parameter is used to select the required table from the specified database.

**prepare statement** ***(boolean)*** If checked, the statement is prepared, and '?' can be filled in using the *parameters* parameter.

**parameters** ***(enumeration)*** Parameters to insert into '?' placeholders when statement is prepared.

## Related Documents

- **Execute SQL** (page 834)

## Tutorial Processes

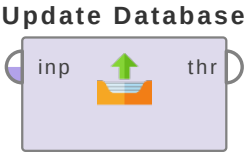
### Reading ExampleSet from a mySQL database


Figure 1.20: Tutorial process 'Reading ExampleSet from a mySQL database'.

The Read Database operator is used to read a mySQL database. The define connection parameter is set to predefined. The define connection parameter was configured using the button next to the drop down box. The name of the connection was set to 'mySQLconn'. The following values were set in the connection parameter's wizard. The Database system was set to 'mySQL'. The Host was set to 'localhost'. The Port was set to '3306'. The Database scheme was set to 'golf'; this is the name of the database. The User was set to 'root'. No password was provided. You will need a password if your database is password protected. Set all the values and test the connection. Make sure that the connection works.

The define query parameter was set to 'table name'. The table name parameter was set to 'golf_table' which is the name of the required table in the 'golf' database. Run the process, you will see the entire 'golf_table' in the Results Workspace. The define query parameter is set to 'table name' if you want to read an entire table from the database. You can also read a selected portion of the database by using queries. Set the define query parameter to 'query' and specify a query in the query parameter. One sample query is already defined in this example. This query reads only those examples from 'golf_table' where the 'Outlook' attribute has the value 'sunny'.

# Update Database




This operator updates the values of all examples with matching ID values in a database.

## Description

The Update Database operator is used for updating an existing table in the specified SQL database. You need to have at least basic understanding of databases and database connections in order to use this operator properly. Go through the parameters and the attached Example Process to understand the flow of this operator.

The user can specify the database connection, a table name and ID column names. The most convenient way of defining the necessary parameters is the *Manage Database Connections* wizard. The most important parameters (database URL and user name) will be automatically determined by this wizard.

The row(s) to update are specified via the db id attribute name parameter. If the id columns of the table do not match all the id values of any given example, the row will be inserted instead. The ExampleSet attribute names must be a subset of the table column names, otherwise the operator will fail.

## Input Ports

**input *(inp)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**through *(thr)*** The ExampleSet that was provided at the input port is delivered through this output port without any modifications. This is usually used to reuse the same ExampleSet in further operators of the process.

## Parameters

**define connection *(selection)*** This parameter indicates how the database connection should be specified. It gives you three options: predefined, url and jndi.

**connection *(string)*** This parameter is only available when the *define connection* parameter is set to *predefined*. This parameter is used for connecting to the database using a predefined connection. You can have many predefined connections. You can choose one of them using the drop down box. You can add a new connection or modify previous connections using the button next to the drop down box. You may also accomplish this by clicking on *Manage Database Connections...* from the *Tools* menu in the main window. A new window appears. This window asks for several details e.g. *Host, Port, Database system, schema, username* and *password*. The *Test* button in this new window will allow you to check whether the connection can be made. Save the connection once the test is successful. After saving a new connection, it can be chosen from the drop down box of the *connection* parameter. You need to have basic understanding of databases for configuring a connection.

**database system** ***(selection)*** This parameter is only available when the *define connection* parameter is set to *url*. This parameter is used for selecting the database system in use. It can have one of the following values: MySQL, PostgreSQL, Sybase, HSQLDB, ODBC Bridge (e.g. Access), Microsoft SQL Server (JTDS), Ingres, Oracle.

**database url** ***(string)*** This parameter is only available when the *define connection* parameter is set to *url*. This parameter is used for defining the URL connection string for the database, e.g. 'jdbc:mysql://foo.bar:portnr/database'.

**username** ***(string)*** This parameter is only available when the *define connection* parameter is set to *url*. This parameter is used for specifying the username of the database.

**password** ***(string)*** This parameter is only available when the *define connection* parameter is set to *url*. This parameter is used for specifying the password of the database.

**jndi name** ***(string)*** This parameter is only available when the *define connection* parameter is set to *jndi*. This parameter is used for giving the JNDI a name for a data source.

**table name** This parameter is used for selecting the required table from the specified database. Please note that you can also write a table name here, if the table does not exist it will be created during writing.

**attribute filter type** ***(selection)*** This parameter allows you to select the ID attribute which values ALL have to match in the example set and the database for the row to be updated. It has the following options:

- **all** Does not make sense in this context so do not use, will break the process.
- **single** This option allows the selection of a single id attribute.
- **subset** This option allows the selection of multiple id attributes through a list. This option will not work if the meta data is not known.
- **regular_expression** This option allows you to specify a regular expression for the id attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameter panel.
- **value_type** This option allows selection of all the id attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. The user should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter panel.
- **block_type** This option is similar in working to the *value_type* option. This option allows the selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric_value_filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameter panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

## Tutorial Processes

### Updating an ExampleSet in a mySQL database

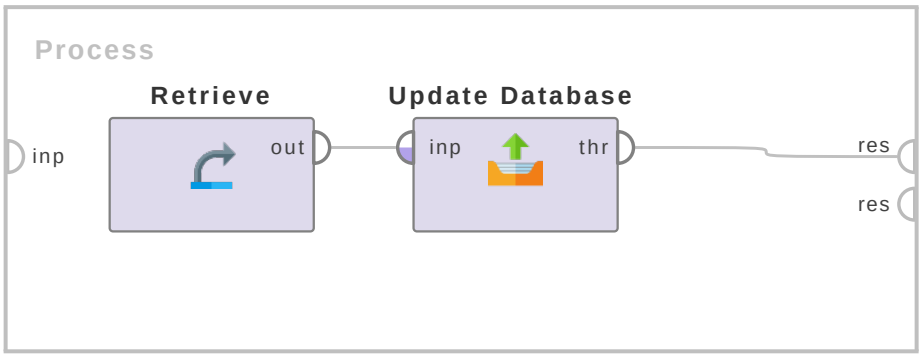
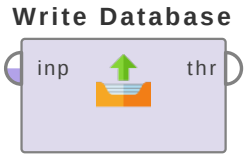



Figure 1.21: Tutorial process 'Updating an ExampleSet in a mySQL database'.

The 'Iris' data set is loaded using the Retrieve operator. The Update Database operator is used to update an existing database table named "Test" in the "My connection" SQL database. Rows in the example set and table which match on their "ID" column will be updated. If no match can be found, the row will be inserted instead.

# Write Database




This operator writes an ExampleSet to an SQL database.

## Description

The Write Database operator is used for writing an ExampleSet to the specified SQL database. You need to have at least basic understanding of databases and database connections in order to use this operator properly. Go through the parameters and the attached Example Process to understand the flow of this operator.

The user can specify the database connection and a table name. Please note that the table will be created during writing if it does not exist. The most convenient way of defining the necessary parameters is the *Manage Database Connections* wizard. The most important parameters (database URL and user name) will be automatically determined by this wizard. At the end, you only have to define the table name. This operator only supports the writing of the complete ExampleSet consisting of all regular and special attributes and all examples. If this is not desired, perform some preprocessing operators like the Select Attributes or Filter Examples operators before applying the Write Database operator. Data from database tables can be read in RapidMiner by using the Read Database operator.

## Input Ports

**input *(inp)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**through *(thr)*** The ExampleSet that was provided at the input port is delivered through this output port without any modifications. This is usually used to reuse the same ExampleSet in further operators of the process.

## Parameters

**define connection *(selection)*** This parameter indicates how the database connection should be specified. It gives you three options: predefined, url and jndi.

**connection *(string)*** This parameter is only available when the *define connection* parameter is set to *predefined*. This parameter is used for connecting to the database using a predefined connection. You can have many predefined connections. You can choose one of them using the drop down box. You can add a new connection or modify previous connections using the button next to the drop down box. You may also accomplish this by clicking on *Manage Database Connections...* from the *Tools* menu in the main window. A new window appears. This window asks for several details e.g. *Host, Port, Database system, schema, username* and *password*. The *Test* button in this new window will allow you to check whether the connection can be made. Save the connection once the test is successful. After saving a new connection, it can be chosen from the drop down box of the *connection* parameter. You need to have basic understanding of databases for configuring a connection.

**database system** ***(selection)*** This parameter is only available when the *define connection* parameter is set to *url*. This parameter is used for selecting the database system in use. It can have one of the following values: MySQL, PostgreSQL, Sybase, HSQLDB, ODBC Bridge (e.g. Access), Microsoft SQL Server (JTDS), Ingres, Oracle.

**database url** ***(string)*** This parameter is only available when the *define connection* parameter is set to *url*. This parameter is used for defining the URL connection string for the database, e.g. 'jdbc:mysql://foo.bar:portnr/database'.

**username** ***(string)*** This parameter is only available when the *define connection* parameter is set to *url*. This parameter is used for specifying the username of the database.

**password** ***(string)*** This parameter is only available when the *define connection* parameter is set to *url*. This parameter is used for specifying the password of the database.

**jndi name** ***(string)*** This parameter is only available when the *define connection* parameter is set to *jndi*. This parameter is used for giving the JNDI a name for a data source.

**table name** This parameter is used for selecting the required table from the specified database. Please note that you can also write a table name here, if the table does not exist it will be created during writing.

**overwrite mode** ***(selection)*** This parameter indicates if an existing table should be overwritten or data should be appended to the existing data.

**set default varchar length** ***(boolean)*** This parameter allows you to set *varchar* columns to default length.

**default varchar length** ***(integer)*** This parameter is only available when the *set default varchar length* parameter is set to true. This parameter specifies the default length of *varchar* columns.

**add generated primary keys** ***(boolean)*** This parameter indicates whether a new attribute holding the auto generated primary keys should be added to the table in the database.

**db key attribute name** ***(string)*** This parameter is only available when the *add generated primary keys* parameter is set to true. This parameter specifies the name of the attribute for the auto generated primary keys.

**batch size** ***(integer)*** This parameter specifies the number of examples which are written at once with one single query to the database. Larger values can greatly improve the speed. However, too large values can drastically decrease the performance. Moreover, some databases have restrictions on the maximum number of values written at once.

## Tutorial Processes

### Writing an ExampleSet to a mySQL database

The 'Golf' data set is loaded using the Retrieve operator. The Write Database operator is used for writing this data set to a mySQL database. The define connection parameter is set to predefined and it is configured using the button next to the drop down box. The name of the connection is set to 'mySQLconn'. The following values are set in the connection parameter's wizard: the Database system is set to 'mySQL'. The Host is set to 'localhost'. The Port is set to '3306'. The Database scheme is set to 'golf'; this is the name of the database. The User is set to 'root'. No

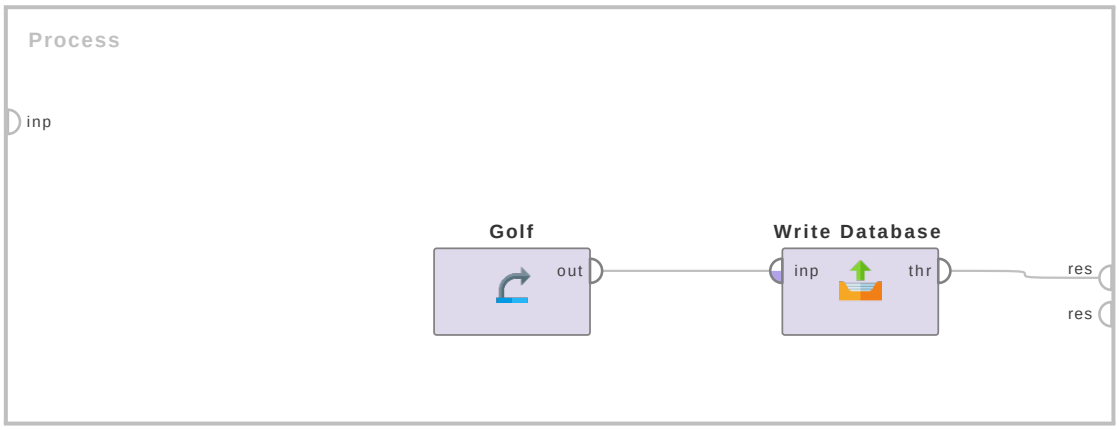



Figure 1.22: Tutorial process 'Writing an ExampleSet to a mySQL database'.

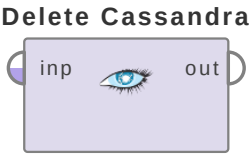
password is provided. You will need a password if your database is password protected. Set all the values and test the connection. Make sure that the connection works.

The table name parameter is set to 'golf_table' which is the name of the required table in the 'golf' database. Run the process, you will see the entire 'golf_table' in the Results Workspace. You can also check the 'golf' database in phpmyadmin to see the 'golf_table'. You can read this table from the database using the Read Database operator. Please study the Example Process of the Read Database operator for more information.

# 1.3 NoSQL

## 1.3.1 Cassandra

### Delete Cassandra




This operator deletes data from a Cassandra table. The input example set is expected to have an ID attribute which is used to define the rows that will be deleted from Cassandra.

#### Description

The Delete Cassandra operator is used to delete data from a Cassandra table.

The data to be deleted is defined by the ID attribute of the provided example set. If the selected table contains a compound primary key, additional attributes can be added to the key with the parameter 'additional_primary_keys'.

#### Input Ports

**input *(inp)*** The example set that defines which data should be deleted from the Cassandra database.

#### Output Ports

**output *(out)*** The passed through example set.

#### Parameters

**conncetion *(configurable)*** The connection details for the Cassandra connection have to be specified. If you have already configured a Cassandra connection, you can select it from the drop-down list. If you have not configured a Cassandra connection yet, select the Cassandra icon right to the drop-down list. Create a new Cassandra connection in the Manage connections box. The contact points and keyspace name are mandatory.

**consistency level *(selection)*** The consistency level for the Cassandra query. The consistency level defines how many Cassandra nodes have to respond to the query in order to be successful. Possible levels are: ONE, TWO, THREE, QUORUM, ALL, ANY

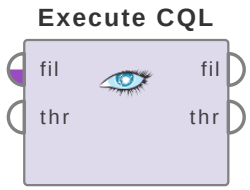
- **ONE** A write must be written at least to one node.
- **TWO** A write must be written at least to two nodes.
- **THREE** A write must be written at least to three nodes.
- **QUORUM** A write must be written at least on a quorum of nodes. A quorum is calculated as (rounded down to a whole number): (replication_factor / 2) + 1. For example, with a replication factor of 3, a quorum is 2 (can tolerate 1 node down). With a replication factor of 6, a quorum is 4 (can tolerate 2 nodes down).
- **ALL** A write must be written on all nodes in the cluster for that row key.
- **ANY** A write must be written to at least one node

**table name *(string)*** Specify the table from which data should be deleted.

**batch size** ***(integer)*** Define the maximum number of rows which should be deleted with one request.

**primary key attributes** ***(enumeration)*** If the selected Cassandra table has a compound primary key this parameter allows you to add more attributes to the primary key.

# Execute CQL




This operator is used to execute a CQL statement on a Cassandra database.

## Description

The Execute CQL operator is used to execute CQL statements on a Cassandra cluster. It cannot return data though and therefore 'SELECT' will not yield any results.

## Input Ports

**file *(fil)*** The CQL file which specifies the CQL statement to be executed. If the 'define query' parameter is set to the 'query file' option, the input port 'file' is used for the CQL file. Note: If the input port is connected to another operator with output port file and the input port is connected to it, the 'query file' option of the 'define query file' parameter is ignored.

**through *(thr)*** An arbitrary Input/Output (IO) object that is passed through the operator.

## Output Ports

**file *(fil)*** If the input port 'file' is connected, the unchanged CQL file is returned.

**through *(thr)*** An arbitrary Input/Output (IO) object that is passed through the operator.

## Parameters

**conncetion *(configurable)*** The connection details for the Cassandra connection have to be specified. If you have already configured a Cassandra connection, you can select it from the drop-down list. If you have not configured a Cassandra connection yet, select the Cassandra icon right to the drop-down list. Create a new Cassandra connection in the Manage connections box. The contact points and keyspace name are mandatory.

**consistency level *(selection)*** The consistency level for the Cassandra query. The consistency level defines how many Cassandra nodes have to respond to the query in order to be successful. Possible levels are: ONE, TWO, THREE, QUORUM, ALL, ANY

- **ONE** A write must be written at least to one node.
- **TWO** A write must be written at least to two nodes.
- **THREE** A write must be written at least to three nodes.
- **QUORUM** A write must be written at least on a quorum of nodes. A quorum is calculated as (rounded down to a whole number): (replication_factor / 2) + 1. For example, with a replication factor of 3, a quorum is 2 (can tolerate 1 node down). With a replication factor of 6, a quorum is 4 (can tolerate 2 nodes down).
- **ALL** A write must be written on all nodes in the cluster for that row key.
- **ANY** A write must be written to at least one node

**define query *(selection)*** This parameter allows to select the mode the data of a query should be defined.

- **query** Define a CQL query via the ‘query’ parameter.
- **query file** Load CQL query from file. If ‘file’ input port is connected, the query is loaded from the provided file object.

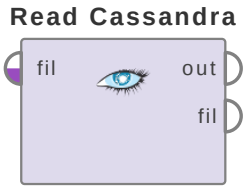
**query *(string)*** The CQL query that defines the data that should be queried can be specified here. It is shown if ’define query’ is set to ‘query’. The operator cannot return data though and therefore ‘SELECT’ will not yield any results.

**query file *(file)*** The CQL file which contains the CQL statement that defines the data that should be queried can be specified here. It is shown if ‘define query’ is set to ‘query file’. The operator cannot return data though and therefore ‘SELECT’ will not yield any results.

**prepare statement *(boolean)*** This parameter specifies whether the query will be a prepared query or a normal query. If activated, the parameter ‘parameters’ is shown.

**parameters *(enumeration)*** If you have activated the ‘prepare statement’ checkbox, this parameter allows to specify prepared values for the query. Every ‘?’ from the specified CQL query will be replaced by the prepared values in the order they are listed in the Edit parameter list: parameters. Note: If you select the wrong type for the parameter, an error message informs you about.

# Read Cassandra




This operator reads an example set from a Cassandra table.

## Description

The example set to be read can be specified via a CQL statement, a CQL file or by specifying a table name.

## Input Ports

**file *(fil)*** The CQL file which specifies the CQL statement to be executed. If the 'define query' parameter is set to the 'query file' option, the input port 'file' is used for the CQL file. Note: If the input port is connected to another operator with output port file and the input port is connected to it, the 'query file' option of the 'define query file' parameter is ignored.

## Output Ports

**output *(out)*** The example set specified via either the CQL statement or the table.

**file *(fil)*** If the input port 'file' is connected, the unchanged CQL file is returned.

## Parameters

**conncetion *(configurable)*** The connection details for the Cassandra connection have to be specified. If you have already configured a Cassandra connection, you can select it from the drop-down list. If you have not configured a Cassandra connection yet, select the Cassandra icon right to the drop-down list. Create a new Cassandra connection in the Manage connections box. The contact points and keyspace name are mandatory.

**consistency level *(selection)*** The consistency level for the Cassandra query. The consistency level defines how many Cassandra nodes have to respond to the query in order to be successful. Possible levels are: ONE, TWO, THREE, QUORUM, ALL, ANY

- **ONE** A write must be written at least to one node.
- **TWO** A write must be written at least to two nodes.
- **THREE** A write must be written at least to three nodes.
- **QUORUM** A write must be written at least on a quorum of nodes. A quorum is calculated as (rounded down to a whole number): (replication_factor / 2) + 1. For example, with a replication factor of 3, a quorum is 2 (can tolerate 1 node down). With a replication factor of 6, a quorum is 4 (can tolerate 2 nodes down).
- **ALL** A write must be written on all nodes in the cluster for that row key.
- **ANY** A write must be written to at least one node

**define query *(selection)*** This parameter allows to select the mode the data of a query should be defined.

- **query** Define a CQL query via the ‘query’ parameter.
- **query file** Load CQL query from file. If ‘file’ input port is connected, the query is loaded from the provided file object.
- **query table** Select a table to be loaded without defining a CQL query.

**query** ***(string)*** This parameter is only displayed when you have selected the ‘query’ parameter. If you click in the ‘Edit text...’ field, the ‘Edit parameter: query’ editor opens and you specify the CQL query. Only SELECT statements are allowed.

**query file** ***(file)*** This parameter is only displayed when you have selected the ‘query file’ parameter. You can select the file that contains the CQL statement that defines the data. Only SELECT statements are allowed. Note: If the Input port of the Read Cassandra operator is connected to an Open file operator, this parameter is not displayed.

**prepare statement** ***(boolean)*** If you have either select ‘query’ or ‘query file’ for the ‘define query’ operator, this parameter is displayed. It specifies whether the query will be a prepared query or a normal query. If activated, the parameter ’parameters’ is shown.

**parameters** ***(enumeration)*** If you have activated the ’prepare statement’ checkbox, this parameter allows to specify prepared values for the query. Every ‘?’ from the specified CQL query will be replaced by the prepared values in the order they are listed in the Edit parameter list: parameters. Note: If you select the wrong type for the parameter, an error message informs you about.

**table** ***(string)*** If ‘define query’ is set to ’query table’, this parameter is displayed. It allows to select the table that should be read.

**datamanagement** ***(selection)*** This parameter allows you to select the appropriate data type for the internal data description.

# Write Cassandra


This operator writes an example set to a Cassandra database.

## Description

The 'Write Cassandra' operator writes an example set to a Cassandra database. The input example set is expected to have an ID attribute which is used as primary key for the selected Cassandra table. If the table has a compound primary key use the parameter 'primary key attributes' to add more attribute as key attributes.

## Input Ports

**input *(inp)*** Requires an example set read by an appropriate operator. The example set must contain an ID attribute. Therefore a Set rule operator must be added to the process in order to specify the ID attribute.

## Output Ports

**output *(out)*** The passed through example set that is written to the Cassandra database.

## Parameters

**conncetion *(configurable)*** The connection details for the Cassandra connection have to be specified. If you have already configured a Cassandra connection, you can select it from the drop-down list. If you have not configured a Cassandra connection yet, select the Cassandra icon right to the drop-down list. Create a new Cassandra connection in the Manage connections box. The contact points and keyspace name are mandatory.

**consistency level *(selection)*** The consistency level for the Cassandra query. The consistency level defines how many Cassandra nodes have to respond to the query in order to be successful. Possible levels are: ONE, TWO, THREE, QUORUM, ALL, ANY

- **ONE** A write must be written at least to one node.
- **TWO** A write must be written at least to two nodes.
- **THREE** A write must be written at least to three nodes.
- **QUORUM** A write must be written at least on a quorum of nodes. A quorum is calculated as (rounded down to a whole number): (replication_factor / 2) + 1. For example, with a replication factor of 3, a quorum is 2 (can tolerate 1 node down). With a replication factor of 6, a quorum is 4 (can tolerate 2 nodes down).
- **ALL** A write must be written on all nodes in the cluster for that row key.
- **ANY** A write must be written to at least one node

**table name *(string)*** Name of the table to which the example set should be written. If a table with the same name already exists, it is updated, presupposed the example set is compatible, i.e., attribute names and types do match. In case the table does not exist yet, a new

table with this name is created and the example set is written to this table. The ID attribute of the example set is used as primary key. In case index columns should be defined for the newly created table, use the parameter ‘index columns’.

**batch size** ***(integer)*** This parameter defines the maximum number of rows which should be written with one request. Default value is 1000.

**primary key attributes** ***(enumeration)*** If the Cassandra table already exists and has a compound primary key, you can add more attributes to the primary key that is used to store the example set. If the Cassandra table does not exist yet, you can add primary key attributes in the Edit parameter list: primary key attributes to create a compound primary key. This primary key consists on the ID attribute and the selected attributes.

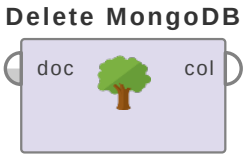
**index columns** ***(enumeration)*** This option is only required in case the Cassandra table does not exists yet. It allows you to define columns as index columns for the newly created table in the Edit paramater list: index columns.

**use ttl** ***(boolean)*** If the checkbox is activated, an additional parameter ‘ttl’ (Time To Live) is displayed. The parameter allows you to specify a time interval value in seconds for the written data. If set, the inserted values are automatically removed from the database after the specified time interval. Note: This remove action affects only the inserted values, not the column themselves. This means that any subsequent update of the column will reset the ‘ttl’ value. By default, values are never removed.

**ttl** ***(integer)*** If the ‘use_ttl’ checkbox is activated, you can specify a value in seconds. By default this value is 120 seconds. You can enter any positive number >= 1.

## 1.3.2 MongoDB

### Delete MongoDB




Deletes a set of MongoDB documents.

#### Description

This operator can be used to delete documents from the specified MongoDB collection. The default configuration of the operator assumes that documents are deleted via their ID, however, more general deletion queries are supported as well.

#### Input Ports

**documents *(doc)*** The documents to be deleted form the specified MongoDB collection.

#### Output Ports

**documents *(doc)*** The documents that have been deleted from the collection. This collection is a subset of the input collection: skipped documents are not included.

#### Parameters

**mongodb instance *(Configurable)*** The MongoDB instance to be used for storing the documents.

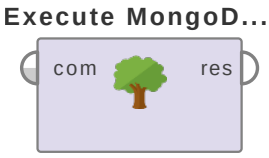
**write concern *(Selection)*** The write concern which controls the acknowledgment of write operations by MongoDB. See the MongoDB documentation for details.

**collection *(String)*** The MongoDB collection in which the documents are stored.

**require id *(Boolean)*** If checked the operator requires documents to include a MongoDB ID, i.e., to include the "_id" field. Documents missing an ID are considered invalid. Otherwise, all documents are passed to the database.

**skip invalid documents *(Boolean)*** If checked, invalid documents (i.e., not in JSON format) are skipped and a warning is logged. Otherwise, the process execution is stopped.

# Execute MongoDB Command




Runs a user specified command on the MongoDB instance.

## Description

This operator can be used to execute arbitrary MongoDB commands. Commands are specified and results returned via JSON/BSON documents. For instance, the command {"create": "myCollection"} creates a new collection of the name "myCollection".

## Input Ports

**command *(com)*** The database command to be executed (a JSON/BSON document). Alternatively, you can specify this document via the command parameter. Note that this parameter is only visible if no document is connected to the input port.

## Output Ports

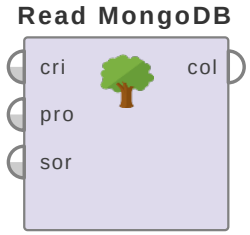
**result *(res)*** The document containing the results of the MongoDB database command.

## Parameters

**mongodb instance *(Configurable)*** The MongoDB instance to be used to run the command.

**command *(String)*** The database command to be executed (a JSON/BSON document). Alternatively, you can specify this document via the command input port.

# Read MongoDB




Reads documents from a MongoDB collection.

## Description

This operator retrieves a collection of documents from the specified MongoDB collection. The query criteria, the query projection and sorting criteria can be specified via JSON/BSON documents.

## Input Ports

**criteria *(cri)*** The query criteria which can be used to select only specific documents (a JSON/BSON document). Alternatively, you can specify this document via the criteria parameter. Note that the parameter is only visible if no document is connected to this input port.

**projection *(pro)*** The query projection which can be used to include/exclude specific fields from the results (a JSON/BSON document). Alternatively, you can specify this document via the projection parameter. Note that the parameter is only visible if no document is connected to this input port.

**sorting criteria *(sor)*** The sorting criteria which can be used to sort the returned documents in a specific order (a JSON/BSON document). Alternatively, you can specify this document via the sort document and sorting criteria parameters. Note that the parameter is only visible if no document is connected to this input port.

## Output Ports

**collection *(col)*** The documents retrieved from the MongoDB collection.

## Parameters

**mongodb instance *(configurable)*** The MongoDB instance to be used for storing the documents.

**collection *(string)*** The MongoDB collection in which the documents are stored.

**criteria *(String)*** The query criteria which can be used to select only specific documents (a JSON/BSON document). Alternatively, you can specify this document via the criteria input port.

**projection *(String)*** The query projection which can be used to include/exclude specific fields from the results (a JSON/BSON document). Alternatively, you can specify this document via the projection input port.

**sort documents *(boolean)*** If checked, a sorting criteria document can be specified to sort the query results. Alternatively, you can enable this behavior by connection a sorting criteria document to the sorting input port.

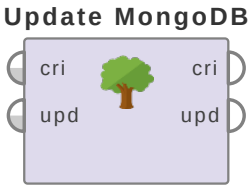
**sorting criteria *(String)*** The sorting criteria which can be used to sort the returned documents in a specific order (a JSON/BSON document). Alternatively, you can specify this document via the sorting input port.

**limit results *(boolean)*** Whether the number of results should be limited.

**limit *(integer)*** The number of documents to be queried.

**skip *(integer)*** The number of documents to be skipped.

# Update MongoDB




Updates one or more documents in a MongoDB collection.

## Description

This operator updates one or more documents in the specified MongoDB collection. An update can thereby refer to the replacement of an document or to the modification of individual fields. The update consists of two parts: the query criteria to identify the document(s) and an update object that contains the new data.

The default update behavior of MongoDB is to replace entire documents. Special BSON operators are required to update individual fields. However, this operator tries to update individual fields by default to prevent data loss.

## Input Ports

**criteria *(cri)*** The JSON/BSON document to identify the document(s) to update.

**update *(upd)*** The JSON/BSON document containing the updated data. If using BSON operator such as "$set" ensure that the parameter "update individual fields" is disabled.

## Output Ports

**criteria *(cri)*** Pass through of the input criteria document (if any).

**update *(upd)*** Pass through of the input update document.

## Parameters

**mongodb instance *(configurable)*** The MongoDB instance to be used for storing the documents.

**write concern *(selection)*** The write concern which controls the acknowledgment of write operations by MongoDB. See MongoDB documentation for details.

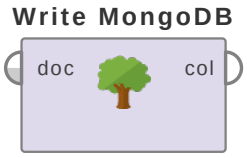
**collection *(string)*** The MongoDB collection in which the documents are stored.

**update individual fields *(boolean)*** If checked, the operator uses the MongoDB operator "$set" to update the fields of the provided update object without replacing other data. Otherwise, the operator simply replaces matching documents with the provided update document.

**insert unmatched documents (upsert) *(boolean)*** If checked, the operator adds the update document to the collection when no document matches the query criteria. Otherwise, the collections remains unchanged.

**update multiple documents *(boolean)*** If checked, all documents that match the query criteria are updated. Otherwise, only the first match is updated.

# Write MongoDB




Writes documents to a MongoDB collection.

## Description

This operator stores JSON/BSON documents in the specified MongoDB collection.

## Input Ports

**documents *(doc)*** The example set(s) containing the entries which should be transformed to JSON documents.

## Output Ports

**documents *(doc)*** The documents that have been written to the collection. This collection is a subset of the input collection: skipped documents are not included.

## Parameters

**mongodb instance *(Configurable)*** The MongoDB instance to be used for storing the documents.

**write concern *(Selection)*** The write concern which controls the acknowledgment of write operations by MongoDB. See the MongoDB documentation for details.

**collection *(String)*** The MongoDB collection in which the documents are stored.

**skip invalid documents *(Boolean)*** If checked, invalid documents (i.e., not in JSON format) are skipped and a warning is logged. Otherwise, the process execution is stopped.

# 1.3.3 Solr

## Add to Solr (Data)


This operator adds an example set to Solr.

### Description

To connect to a Solr server, you have to specify a Solr connection. This comprises the URL of a Solr server and an optional user/password combination for authentication. Typically, the Solr server URL ends with the string '/solr'.

The next step is to select a collection on the server. A collection can be imagined as a table. It is composed of several columns, which are called Solr fields. A Solr field has a type (e.g. number) and a key (the name of the column). Each entry in Solr can be imagined as a row and contains values for the respective fields.

A RapidMiner example set has a very similar structure. It also can be imagined as a table. Therefore every row of RapidMiner is added as row in Solr. The RapidMiner attributes are used as Solr collection fields.

### Input Ports

**input *(inp)*** This port connects the example set, which has to be added.

### Output Ports

**through *(thr)*** The added example set is provided at this port.

### Parameters

**connection *(configurable)*** The connection details for the Solr connection have to be specified. If you have already configured a Solr connection, you can select it from the drop-down list. If you have not configured a Solr connection yet, select the icon to the right of the drop-down list. Create a new Solr connection in the Manage connections dialog. The Solr server URL is required. Additionally, you can specify a username/password combination for authentication.

**collection *(string)*** Provide the name of the Solr collection, which has to be used to access data.

# Add to Solr (Documents)


This operator adds collections of documents to Solr.

## Description

To connect to a Solr server, you have to specify a Solr connection. This comprises the URL of a Solr server and an optional user/password combination for authentication. Typically, the Solr server URL ends with the string '/solr'.

The next step is to select a collection on the server. A collection can be imagined as a table. It is composed of several columns, which are called Solr fields. A Solr field has a type (e.g. number) and a key (the name of the column). Each entry in Solr can be imagined as a row and contains values for the respective fields.

A RapidMiner document has a set of metadata records, which consist of a key and a related value. The metadata keys are mapped to the Solr attributes. RapidMiner documents have an additional body. Therefore you can select a Solr field, in which the document body will be stored.

## Input Ports

**documents *(doc)*** This port connects a collection of documents, which has to be added. This port is extendable.

## Output Ports

**documents *(doc)*** The added collection of documents are provided at this port. This port is extendable.

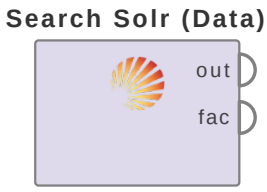
## Parameters

**connection *(configurable)*** The connection details for the Solr connection have to be specified. If you have already configured a Solr connection, you can select it from the drop-down list. If you have not configured a Solr connection yet, select the icon to the right of the drop-down list. Create a new Solr connection in the Manage connections dialog. The Solr server URL is required. Additionally, you can specify a username/password combination for authentication.

**collection *(string)*** Provide the name of the Solr collection, which has to be used to access data.

**document body field *(string)*** The Solr field, which is used for the RapidMiner document body.

# Search Solr (Data)




This operator searches for Solr entries and generates an example set.

## Description

To connect to a Solr server, you have to specify a Solr connection. This comprises the URL of a Solr server and an optional user/password combination for authentication. Typically, the Solr server URL ends with the string '/solr'.

The next step is to select a collection on the server. A collection can be imagined as a table. It is composed of several columns, which are called Solr fields. A Solr field has a type (e.g. number) and a key (the name of the column). Each entry in Solr can be imagined as a row and contains values for the respective fields.

A RapidMiner example set has a very similar structure. It also can be imagined as a table. Therefore every row of Solr is added as row in RapidMiner. The Solr collection fields are used as RapidMiner attributes.

To search Solr, you have to specify a query string. You can add filters to refine your query. E.g., if you want to receive no items with an attribute key "popularity" and the value "6", use "!popularity:6". The range of the entries to receive can be set by the attributes offset and rows. You can specify, which field is used to sort the received entries. It is also possible to enable faceting. Faceted search breaks up search results into multiple categories. Use "facet fields" and "date facets" to specify Solr fields for faceting.

If a Solr field supports multiple elements, the related values are provided as a JSON array.

## Output Ports

**output** ***(out)*** This port provides the main search result. It consists of an example set.

**facets** ***(fac)*** This port is used to provide results of the faceted search. An example set is provided and contains the field name, the value which was found, and the number of occurrences.

## Parameters

**connection** ***(configurable)*** The connection details for the Solr connection have to be specified. If you have already configured a Solr connection, you can select it from the drop-down list. If you have not configured a Solr connection yet, select the icon to the right of the drop-down list. Create a new Solr connection in the Manage connections dialog. The Solr server URL is required. Additionally, you can specify a username/password combination for authentication.

**collection** ***(string)*** Provide the name of the Solr collection, which has to be used to access data.

**query** ***(string)*** The term to search for.

**filter query** ***(string)*** A filter, which does not influence the relevancy score, which is the default sort order. With this field, you can refine your query. E.g. if the field name has to contain John, but must not contain Doe, you can use 'name:John -name:Doe'.

**offset** ***(integer)*** The first document index to fetch.

**limit** ***(integer)*** The maximum number of results.

**sort** ***(boolean)*** Specifies, if search results are sorted.

**sort field** ***(string)*** The Solr field which is used for sorting.

**sort order** ***(selection)*** The sorting order of results.

**faceted search** ***(boolean)*** Specifies, if faceted searching is used.

**categorical facets** ***(enumeration)*** The facets to use for faceted search.

**date facets** ***(enumeration)*** The date facets to use for faceted search. A single date facet consists of the field name, a start date, an end date, and a gap.

**include generated fields** ***(boolean)*** Specifies, if automatically generated fields are included into search results. These fields can consist of SolrCloud fields or can be based on dynamic Solr fields.

# Search Solr (Documents)


This operator searches for Solr entries and generates a document for each result.

## Description

To connect to a Solr server, you have to specify a Solr connection. This comprises the URL of a Solr server and an optional user/password combination for authentication. Typically, the Solr server URL ends with the string '/solr'.

The next step is to select a collection on the server. A collection can be imagined as a table. It is composed of several columns, which are called Solr fields. A Solr field has a type (e.g. number) and a key (the name of the column). Each entry in Solr can be imagined as a row and contains values for the respective fields.

A RapidMiner document has a set of metadata records, which consist of a key and a related value. The metadata keys are mapped to the Solr attributes. RapidMiner documents have an additional body. Therefore you can select a Solr field, whose contents will bestored in the RapidMiner document body.

To search Solr, you have to specify a query string. You can add filters to refine your query. E.g., if you want to receive no items with an attribute key "popularity" and the value "6", use "!popularity:6". The range of the entries to receive can be set by the attributes offset and rows. You can specify, which field is used to sort the received entries. It is also possible to enable faceting. Faceted search breaks up search results into multiple categories. Use "facet fields" and "date facets" to specify Solr fields for faceting.

If a Solr field supports multiple elements, the related values are provided as a JSON array.

## Output Ports

**output *(out)*** This port provides the main search result. It consists of a collection of documents.

**facets *(fac)*** This port is used to provide results of the faceted search. An example set is provided and contains the field name, the value which was found, and the number of occurrences.

## Parameters

**connection *(configurable)*** The connection details for the Solr connection have to be specified. If you have already configured a Solr connection, you can select it from the drop-down list. If you have not configured a Solr connection yet, select the icon to the right of the drop-down list. Create a new Solr connection in the Manage connections dialog. The Solr server URL is required. Additionally, you can specify a username/password combination for authentication.

**collection *(string)*** Provide the name of the Solr collection, which has to be used to access data.

**query *(string)*** The term to search for.

**document body field *(string)*** The Solr field, which is used as the RapidMiner document body.

**filter query** ***(string)*** A filter, which does not influence the relevancy score, which is the default sort order. With this field, you can refine your query. E.g. if the field name has to contain John, but must not contain Doe, you can use ‘name:John -name:Doe’.

**offset** ***(integer)*** The first document index to fetch.

**limit** ***(integer)*** The maximum number of results.

**sort** ***(boolean)*** Specifies, if search results are sorted.

**sort field** ***(string)*** The Solr field which is used for sorting.

**sort order** ***(selection)*** The sorting order of results.

**faceted search** ***(boolean)*** Specifies, if faceted searching is used.

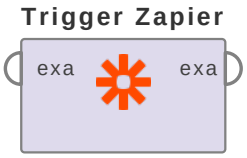
**categorical facets** ***(enumeration)*** The facets to use for faceted search.

**date facets** ***(enumeration)*** The date facets to use for faceted search. A single date facet consists of the field name, a start date, an end date, and a gap.

**include generated fields** ***(boolean)*** Specifies, if automatically generated fields are included into search results. These fields can consist of SolrCloud fields or can be based on dynamic Solr fields.

# 1.4 Applications

## Trigger Zapier




This operator allows you to use the Zapier service connecting to a huge collection of data sinks.

### Description

Zapier works by defining triggers and actions and combining them into Zaps. The operator can be used as such a trigger. Zapier can then be used to send data to arbitrary actions. Each example in your example set will trigger one action. Note: The Trigger Zapier operator therefore always needs another operator that provides an example set.

To use this operator, perform the following steps.

- Create an account on www.zapier.com and create a new Zap. (You can quickly get there by clicking the button next to the "zapier url" parameter of the "Trigger Zapier" operator.)
- Select "RapidMiner" as the trigger service and "Trigger Zapier" operator as the trigger.
- Select an action service from the list and an action.
- In step 2 of the Zap creation ("Select RapidMiner account"), a URL is provided that you must first copy to the clipboard and then paste as the *zapier url* parameter of this operator
- In step 3 of the Zap creation (Select an action service account) log in the Action service and grant access to the Zapier service.
- The fields you can select in Step 5 ("Match up") correspond to the attributes of the example set received by this operator. Note: When you use this operator for the first time, the list may be empty. Therefore it is recommended to execute a dry run of your process in RapidMiner studio to let Zapier know which attributes will be expected. To that end, check the *test hook* parameter and run your process. It may be possible that you must refresh the Zapier page afterwards. Then you should be able to pick appropriate attributes from the attribute list.
- In step 7 save and activate your trigger on the Zapier web page. In a productive run make sure that the *test hook* parameter is switched off.

If you run your process, you should see that one action in Zapier is triggered for each example in your example set.

### Input Ports

**example set in *(exa)*** This will trigger your Zap once for each example in the example set.

### Output Ports

**example set in *(exa)*** The same example set as received as an input.

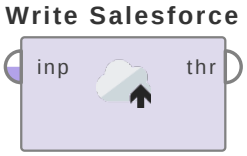
## Parameters

**zapier url** ***(HTTPS url)*** The URL to which your requests are sent. Zapier shows the URL in Step 2 (”Select a RapidMiner account”). Make sure to use HTTPS.

**test hook** ***(boolean)*** If checked, only test messages will be sent to Zapier. These will not trigger your Zap. You can use test mode to populate the choices in Step 5 (”Match up”) of the Zap editor.

## 1.4.1 Salesforce

### Delete Salesforce




This operator deletes records of a Salesforce object from the input example set.

#### Description

This operator deletes entries of a Salesforce object from the input example set in the specified Salesforce instance. Each example of the input data will delete one Salesforce object, identified by the ID attribute.

If the skip invalid parameter is selected, each example for which the deletion in Salesforce failed will be ignored.

#### Input Ports

**input *(inp)*** The example set containing the entries which should be deleted. The entries are identified by a Salesforce ID.

#### Output Ports

**through *(thr)*** The unmodified input example set.

#### Parameters

**connection *(configurable)*** The connection details for the Salesforce connection have to be specified. If you have already configured a Salesforce connection, you can select it from the drop-down list. If you have not configured a Salesforce connection yet, select the icon to the right of the drop-down list. Create a new Salesforce connection in the Manage connections box. This includes username, password and the security token. The URL is predefined but can be changed to work on a different API version.

**skip invalid rows *(boolean)*** If selected, skips and ignores failed deletions of a record. In such cases, invalid deletion IDs will be skipped. If not selected, the process will fail if a record cannot be deleted and revert all previous deletions.

# Read Salesforce


This operator creates an example set from a Salesforce object. Each record is represented by an example containing an attribute for each field.

## Description

This operator reads an example set from a Salesforce object in the specified Salesforce instance. Each record is represented by an example containing an attribute for each field.

You can either use the simplified user interface to create the query, or use the advanced SOQL editor which allows you to directly enter SOQL queries.

Note that datetime fields are always treated as UTC, using the pattern "yyyy-MM-dd'T'HH:mm:ss.SSSX". Date fields use "yyyy-MM-dd" and time fields "HH:mm:ss.SSSX".

## Output Ports

**output** ***(out)*** The example set created from the result of the Salesforce query. Each queried field corresponds to an attribute and each record is represented as an example.

## Parameters

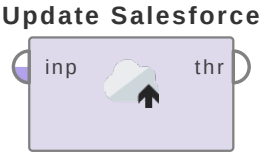
**connection** ***(configurable)*** The connection details for the Salesforce connection have to be specified. If you have already configured a Salesforce connection, you can select it from the drop-down list. If you have not configured a Salesforce connection yet, select the icon to the right of the drop-down list. Create a new Salesforce connection in the Manage connections box. This includes username, password and the security token. The URL is predefined but can be changed to work on a different API version.

**query** ***(salesforce_query)*** The SOQL query which will be used to query Salesforce. You can either select the Simple or the Advanced SOQL mode. The Simple mode supports you and eases the query creation, while the Advanced SOQL mode allows you to use the full power of SOQL.

**guess value types** ***(boolean)*** If selected, the operator tries to guess the value types for each column. It does so by taking the first ten rows of the returned data and trying to parse it as an integer, number, date_time, date, time (in this order). If this all fails, the attribute is treated as nominal. If this option is not selected, the operator treats all attributes as nominal. Other operators have to be applied afterwards to convert the attributes to the desired value type.

**batch size** ***(integer)*** The batch size the query should use. If you query more records than the batch size, they are retrieved in chunks of the specified size. This parameter is only for performance optimization and does not affect the result.

# Update Salesforce




This operator updates records of a Salesforce object from the input example set.

## Description

This operator updates entries for a Salesforce object from the input example set in the specified Salesforce instance. Each example of the input data will update one record. The selected attributes will be used as the respective field values. Each record is identified by its ID, which is taken from the ID attribute.

To select the fields which should be updated, you can use the attribute selection parameters. Attributes which are not selected are ignored.

Note: Datetime fields are always treated as UTC (Coordinated Universal Time), using the pattern "yyyy-MM-dd'T'HH:mm:ss.SSSX". Date fields use "yyyy-MM-dd" and time fields "HH:mm:ss.SSSX".

## Input Ports

**input *(inp)*** The example set containing the entries which should be updated. Note: The example set must have an ID column by which the records can be identified in the Salesforce object.

## Output Ports

**through *(thr)*** The unmodified input example set.

## Parameters

**connection *(configurable)*** The connection details for the Salesforce connection have to be specified. If you have already configured a Salesforce connection, you can select it from the drop-down list. If you have not configured a Salesforce connection yet, select the icon to the right of the drop-down list. Create a new Salesforce connection in the Manage connections box. This includes username, password and the security token. The URL is predefined but can be changed to work on a different API version.

**object name *(selection)*** The name of the Salesforce object for which you want to update records.

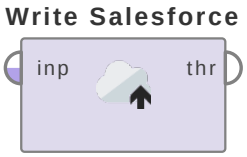
**skip invalid rows *(boolean)*** If selected, skips and ignores failed creations of a record. In such cases, the ID column value is set to missing. If not selected, the process will fail if a record cannot be created and no records will be created in Salesforce at all.

**attribute filter type *(selection)*** You can specify which attributes should be updated. By default all attributes are updated. Possible values are: all, single, subset, regular_expression, value_type, block_type, no_missing_values, numeric_value_filter.

**invert selection *(boolean)*** If the checkbox is activated, the attributes selection is toggled: all attributes that are selected before, are excluded and all excluded attributes are included. If the checkbox is deactivated (default) the attribute selection is applied.

**includes special attributes *(boolean)*** If the checkbox is activated, the operator is also applied to special attributes. If the checkbox is deactivated, the special attributes are ignored.

# Write Salesforce




This operator creates records for a Salesforce object from the input example set.

## Description

This operator creates entries for a Salesforce object from the input example set in the specified Salesforce instance. Each example of the input data will create one Salesforce entry. The selected attributes will be used as the respective field values.

To select the fields which should be created, you can use the attribute selection parameters. Attributes which are not selected are ignored.

Note that datetime fields are always treated as UTC (Coordinated Universal Time), using the pattern "yyyy-MM-dd'T'HH:mm:ss.SSSX". Date fields use "yyyy-MM-dd" and time fields "HH:mm:ss.SSSX".

If the skip invalid parameter is selected, each row for which the creation in Salesforce failed will return a missing value in the ID column.

## Input Ports

**input *(inp)*** The example set containing the entries which should be created. Note: It is strictly forbidden to have an ID column in the example set.

## Output Ports

**through *(thr)*** The input example set including an ID column containing the IDs generated by Salesforce for each entry or a missing value if the entry could not be created and the parameter 'skip_invalid_rows' is set to true.

## Parameters

**connection *(configurable)*** The connection details for the Salesforce connection have to be specified. If you have already configured a Salesforce connection, you can select it from the drop-down list. If you have not configured a Salesforce connection yet, select the icon to the right of the drop-down list. Create a new Salesforce connection in the Manage connections box. This includes username, password and the security token. The URL is predefined but can be changed to work on a different API version.

**object name *(selection)*** The name of the Salesforce object for which to create records.

**salesforce id column *(string)*** The name of the column which will contain the IDs for each successfully created entry. The name must not be used for any existing column in the incoming example set.

**skip invalid rows *(boolean)*** If selected, skips and ignores failed creations of a record. In such cases, the ID column value is set to missing. If not selected, the process will fail if a record cannot be created and no records will be created in Salesforce at all.

**attribute filter type** ***(selection)*** You can specify which attributes should be updated. By default all attributes are updated. Possible values are: all, single, subset, regular_expression, value_type, block_type, no_missing_values, numeric_value_filter.

**invert selection** ***(boolean)*** If the checkbox is activated, the attributes selection is toggled: all attributes that are selected before, are excluded and all excluded attributes are included. If the checkbox is deactivated (default) the attribute selection is applied.

**includes special attributes** ***(boolean)*** If the checkbox is activated, the operator is also applied to special attributes. If the checkbox is deactivated, the special attributes are ignored.

# 1.4.2 Mozenda

## Read Mozenda


This operator loads the specified view from the Mozenda cloud storage and returns it's data as an example set.

### Description

After you have created a Mozenda account, you can get a Mozenda view as an example set, using this operator.

### Output Ports

**output** ***(out)*** The example set created from fetching the specified Mozenda view.

### Parameters

**connection** ***(configurable)*** The connection details for the Mozenda connection have to be specified. If you have already configured a Mozenda connection, you can select it from the drop-down list. If you have not configured a Mozenda connection yet, select the cloud icon to the right of the drop-down list. Create a new Mozenda connection in the Manage connections box. An API key is required, which is available in your Mozenda web interface. Test the connection and click the Save all changes button.

**collection** ***(selection)*** Select the Mozenda collection from the drop-down list. All available collections are displayed in the drop-down list. To update the list of collections you need to update the cache of the specified Mozenda connection. Click on the cloud icon to the right of your selected connection. Select your Mozenda connection in the Manage connections box and click on the menu next to the Test button to update the cache.

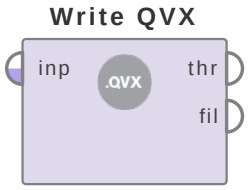
**view** ***(selection)*** Select the Mozenda view from the drop-down list. All available views associated with the selected Mozenda collection are displayed in the drop-down list. To update the list of views you need to update the cache of the specified Mozenda connection. Click on the cloud icon to the right of your selected connection. Select your Mozenda connection in the Manage connections box and click on the menu next to the Test button to update the cache.

**page number** ***(integer)*** This parameter defines the page number of the page that is returned from the selected Mozenda view.

**page item count** ***(integer)*** This parameter defines the number of items that are requested to be on one page of the Mozenda view.

# 1.4.3 Qlik

## Write QVX




This operator writes data in Qlik's QVX data exchange format.

### Description

This operator can write Example Sets in Qlik's data exchange format QVX. The operator can either send the file as specified by the "file" parameter or send it to the output port labeled "file". If that port is connected, the "file" parameter can no longer be used and a file object is sent to the port. This file object can subsequently be used in two ways:

- It can be further processed, e.g. written to the repository by using one of the file operators like Write File.
- It sent to the result port of the process, e.g. when using it as output of a RapidMiner Server web service. This is an easy way to connect RapidMiner Server as a data source to Qlik.

### Input Ports

**input *(inp)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

### Output Ports

**through *(thr)*** The ExampleSet that was provided at the input port is delivered through this output port without any modifications. This is usually used to reuse the same ExampleSet in further operators of the process.

**file *(fil)*** This port buffers the file object for passing it to the reader operators

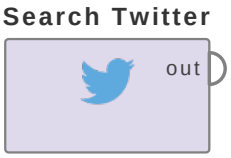
### Parameters

**file *(filename)*** The file to which this operator will write the input example set. Only available if the file port is not connected.

**table name *(string)*** The name by which this table will be known in Qlik.

# 1.4.4 Twitter

## Get Twitter Relations




This operator gets friends or followers of a specific user.

### Description

With the Get Twitter Relations operator, you can specify a Twitter user and receive the users friends or followers.

Select a Twitter connection to specify the Twitter account for the Twitter API access. Specify the user name or the user ID of interest. Finally, set if you want to receive the followers or friends of the specified user.

Note that the standard Twitter API has strict rate limits! Please consult the Twitter documentation on how to avoid hitting these rate limits.

### Output Ports

**output** ***(out)*** An example set consisting of data from the Twitter API. This consists of the IDs of friends or the IDs of followers. Additionally it contains the name or ID, that was searched for.

### Parameters

**connection** ***(configurable)*** The connection details for the Twitter connection have to be specified. If you have already configured a Twitter connection, you can select it from the drop-down list. If you have not configured a Twitter connection yet, select the icon to the right of the drop-down list. Create a new Twitter connection in the Manage Connections box.

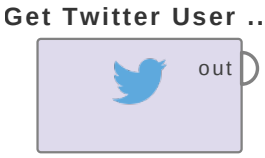
**query type** ***(selection)*** Specifies whether a user should be searched by id or screen name.

**id** The id of the user.

**name** The screen name of the user.

**relation** ***(selection)*** Specifies whether friends or followers of that user should be retrieved.

# Get Twitter User Details




This operator shows properties of a specific user.

## Description

With the Get Twitter User Details operator, you can specify a Twitter user and receive a list of properties of the user.

Select a Twitter connection to specify the Twitter account for the Twitter API access. Specify the user name or the user ID to get information about the user.

Note that the standard Twitter API has strict rate limits! Please consult the Twitter documentation on how to avoid hitting these rate limits.

## Output Ports

**output** ***(out)*** An example set consisting of data from the Twitter API. This comprises the users ID, name, screen name, description, URL, creation date, verification and protection info, the number of followers and friends, the number of tweets, the language, the profile image, and the time zone.

## Parameters

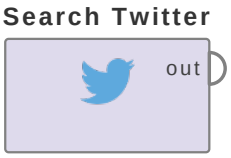
**connection** ***(configurable)*** The connection details for the Twitter connection have to be specified. If you have already configured a Twitter connection, you can select it from the drop-down list. If you have not configured a Twitter connection yet, select the icon to the right of the drop-down list. Create a new Twitter connection in the Manage Connections box.

**query type** ***(selection)*** Specifies whether a user should be searched by id or screen name.

**id** ***(long)*** The id of the user.

**user** ***(string)*** The screen name of the user.

# Get Twitter User Statuses




This operator searches for Twitter statuses of a specific user.

## Description

With the Get Twitter User Statuses operator, you can specify a Twitter user and receive a list of statuses of the user. The list of statuses contains additional data with context of the statuses. There are advanced parameters you can use to specify additional search restrictions.

Select a Twitter connection to specify the Twitter account for the Twitter API access. Specify at least the user name or the user ID of interest. There are advanced parameters you can use to specify additional search restrictions. For example, you can increase the number of pages. This will increase the number of search results.

Note that the standard Twitter API has strict rate limits! Please consult the Twitter documentation on how to avoid hitting these rate limits.

## Output Ports

**output** ***(out)*** An example set consisting of data from the Twitter API. This comprises the tweet text, the tweet ID, the number of retweets, the date of creation, the language, the geolocation, the used source of the tweet, and user information.

## Parameters

**connection** ***(configurable)*** The connection details for the Twitter connection have to be specified. If you have already configured a Twitter connection, you can select it from the drop-down list. If you have not configured a Twitter connection yet, select the icon to the right of the drop-down list. Create a new Twitter connection in the Manage Connections box.

**query type** ***(selection)*** Specifies whether a user should be searched by id or screen name.

**id** ***(long)*** The id of the user.

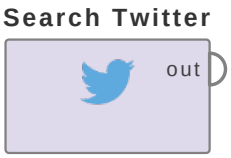
**user** ***(string)*** The screen name of the user.

**limit** ***(integer)*** The limit on the number of tweets to return.

**since id** ***(long)*** Returns results with an ID greater than (that is, more recent than) the specified ID.

**max id** ***(long)*** Returns results with an ID less than (that is, older than) or equal to the specified ID.

# Search Twitter




This operator searches for Twitter statuses.

## Description

With the Search Twitter operator, you can specify a query and get Twitter statuses containing this query. The list of statuses contains additional data with context of the statuses. In the expert mode, you can specify additional search restrictions.

Select a Twitter connection to specify the Twitter account for the Twitter API access. Specify at least a query to search Twitter for it. There are advanced parameters you can use to specify additional search restrictions. For example, you can limit the search results to a language.

Note that the standard Twitter API has strict rate limits! Please consult the Twitter documentation on how to avoid hitting these rate limits.

## Output Ports

**output *(out)*** An example set consisting of data from the Twitter API. This comprises the tweet text, the tweet ID, the number of retweets, the date of creation, the language, the geolocation, the used source of the tweet, and user information.

## Parameters

**connection *(configurable)*** The connection details for the Twitter connection have to be specified. If you have already configured a Twitter connection, you can select it from the drop-down list. If you have not configured a Twitter connection yet, select the icon to the right of the drop-down list. Create a new Twitter connection in the Manage Connections box.

**query *(string)*** The term that should be searched.

**result type *(selection)*** Specifies the preferred search result type.

**limit *(integer)*** The limit on the number of tweets to return.

**since id *(long)*** Returns results with an ID greater than (that is, more recent than) the specified ID.

**max id *(long)*** Returns results with an ID less than (that is, older than) or equal to the specified ID.

**language *(string)*** Restricts tweets to the given language, specified by an ISO 639-1 code.

**locale *(string)*** Specifies the language of the query you are sending. (The official Twitter API mentions, that only 'ja' is currently effective.)

**until *(string)*** Returns tweets generated before the given date. The values year, month, and day are used as search parameters.

**filter by geo location *(boolean)*** Indicates if the results should be filtered by a geo location.

**latitude** ***(double)*** The latitude of the geo location.

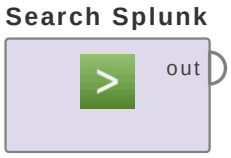
**longitude** ***(double)*** The longitude of the geo location.

**radius** ***(double)*** The radius of the geo location.

**radius unit** ***(selection)*** The unit of the geo location radius.

# 1.4.5 Splunk

## Search Splunk




Reads search results from a Splunk® server.

### Description

This operator can be used to query a Splunk® server based on a query term and returns the results as an example set. Search results can be restricted by specifying a time frame.

### Output Ports

**result *(res)*** The example set consisting of the search results.

### Parameters

**connection *(Configurable)*** The Splunk® connection to use. Select a connection from the drop-down or click the button to create a new one.

**query *(String)*** The Splunk® query in Splunk Process Language (SPL).

**pagination *(Boolean)*** If set, only a limited number of results will be returned, starting from a given offset.

**offset *(Integer)*** Offset from which the result set should start.

**limit *(Integer)*** Maximum number of results to return.

**earliest time *(Time)*** If this parameter is set, it specifies the earliest time in the time range to search.

**latest time *(Time)*** If this parameter is set, it specifies the latest time in the time range to search.

# 1.5 Cloud Storage

## 1.5.1 Amazon S3

### Loop Amazon S3


This operator loops over all files in the specified bucket/folder from the Amazon S3 cloud storage.

#### Description

After you have configured your Amazon S3 account, you can process all Amazon S3 files within the selected folder.

Be aware that the operator cannot read the file as example set. For this reason, you must connect the file input in the inner process of this operator to another appropriate operator to process the file. For example, if you want to load Excel files from your Amazon S3 folder, you must connect the file input in the inner process with the Read Excel operator.

#### Input Ports

**in** ***(in )*** Optional input data which is delivered to the inner process.

#### Output Ports

**out** ***(out)*** Output data of the inner process.

#### Parameters

**connection** ***(configurable)*** The connection details for the Amazon S3 connection have to be specified. If you have already configured a Amazon S3 connection, you can select it from the drop-down list. If you have not configured a Amazon S3 connection yet, select the icon to the right of the drop-down list. Create a new Amazon S3 connection in the Manage connections box. The access key, secret key and the region are required. Note: It is very important to select the correct region for your connection. Otherwise an error occurs.

**folder** ***(selection)*** Provide the name of the Amazon S3 'folder' over which you want to loop. Note that the concept of folders does not exist in Amazon S3, so the default delimiter ('/') is used to represent them. If your file was stored as 'name1/name2/my_file.xls' on Amazon S3, the file 'my_file.xls' would be displayed as residing in the folder 'name1/name2/'.

**filter** ***(string)*** Optional filter via a regular expression which is used to exclude files from looping over them, e.g. 'a.*b' for all files starting with 'a' and ending with 'b'. Ignored if empty.

**filtered string** ***(selection)*** Indicates which part of the file name is matched against the filter expression.

- **file_name** Filtered on the name, e.g. 'myfolder/myfile.txt'
- **full_path** Filtered on the full path, e.g. 'mybucket/myfolder/myfile.txt'

- **parent_path** Filtered on the parent folder, e.g. ‘myfolder/’

**file name macro** ***(string)*** The name of the macro which will contain the name of the current file for each file the loop iterates over, e.g. ‘myfolder/myfile.txt’

**file path macro** ***(string)*** The name of the macro which will contain the full path of the current file for each file the loop iterates over, e.g. e.g. ‘mybucket/myfolder/myfile.txt’

**parent path macro** ***(string)*** The name of the macro which will contain the parent folder of the current file for each file the loop iterates over, e.g. e.g. ‘myfolder/’

**recursive** ***(boolean)*** If selected, the loop will also iterate over all files in all subfolders of the selected folder. Otherwise, it will only iterate over the files in the selected folder.

# Read Amazon S3


This operator downloads the specified file from the Amazon S3 cloud storage.

## Description

After you have configured your Amazon S3 account, you can load the Amazon S3 file with this operator.

Be aware that the operator cannot read the file as example set. For this reason, you must connect the Read Amazon S3 operator to another appropriate operator to read the file. For example, if you want to load an Excel file from your Amazon S3, you must connect the Read Amazon S3 operator with the Read Excel operator to see the result.

## Output Ports

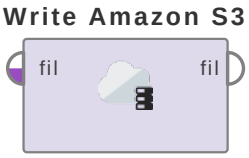
**file *(fil)*** The downloaded file object is returned here. Must be connected to a appropriate Read Operator, for example Read Excel or Read CSV.

## Parameters

**connection *(configurable)*** The connection details for the Amazon S3 connection have to be specified. If you have already configured a Amazon S3 connection, you can select it from the drop-down list. If you have not configured a Amazon S3 connection yet, select the icon to the right of the drop-down list. Create a new Amazon S3 connection in the Manage connections box. The access key, secret key and the region are required. Note: It is very important to select the correct region for your connection. Otherwise an error occurs.

**file *(selection)*** Select the Amazon S3 file you want to download. Note that the concept of folders does not exist in Amazon S3, so the default delimiter ('/') is used to represent them. If your file was stored as 'name1/name2/my_file.xls' on Amazon S3, the file 'my_file.xls' would be displayed as residing in the folder 'name1/name2/'.

# Write Amazon S3




This operator uploads the input file to the Amazon S3 cloud storage.

## Description

Before you can upload the input file to the selected Amazon S3 cloud storage, you must load it with an Open file operator.

Ensure that the correct bucket is selected, otherwise an error occurs! Buckets are container for the Amazon S3 objects. Each Bucket name is unique across all of Amazon S3.

## Input Ports

**file *(fil)*** The file object which should be uploaded to Amazon S3 cloud storage. The file must be provided by an Open file operator.

## Output Ports

**file *(fil)*** The input file object is passed through and returned here.

## Parameters

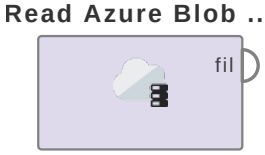
**connection *(configurable)*** The connection details for the Amazon S3 connection have to be specified. If you have already configured a Amazon S3 connection, you can select it from the drop-down list. If you have not configured a Amazon S3 connection yet, select the icon to the right of the drop-down list. Create a new Amazon S3 connection in the Manage connections box. The access key, secret key and the region are required. Note: It is very important to select the correct region for your connection. Otherwise an error occurs.

**file *(selection)*** Enter the name of the file as it should be stored on Amazon S3, e.g., /mybucket/my_file.xls.

**content type *(string)*** This option is optional. Enter the MIME type of the upload file, e.g., text/xml.

# 1.5.2 Azure Blob Storage

## Loop Azure Blob Storage




This operator loops over all files in the specified container/folder from the Microsoft Azure Blob Storage.

### Description

After you have configured your Azure Blob Storage account, you can process all Azure Blob Storage files within the selected folder.

Be aware that the operator cannot read the file as example set. For this reason, you must connect the file input in the inner process of this operator to another appropriate operator to process the file. For example, if you want to load Excel files from your Azure Blob Storage folder, you must connect the file input in the inner process with the Read Excel operator.

### Input Ports

**in** ***(in )*** Optional input data which is delivered to the inner process.

### Output Ports

**out** ***(out)*** Output data of the inner process.

### Parameters

**connection** ***(configurable)*** The connection details for the Azure Blob Storage connection have to be specified. If you have already configured an Azure Blob Storage connection, you can select it from the drop-down list. If you have not configured an Azure Blob Storage yet, select the icon to the right of the drop-down list. Create a new Azure Blob Storage connection in the Manage connections box. The account name and account key are required.

**folder** ***(selection)*** Provide the name of the Azure Blob Storage 'folder' over which you want to loop. Note that the concept of folders does not exist in Azure Blob Storage, so the default delimiter ('/') is used to represent them. If your file was stored as 'name1/name2/my_file.xls' on Azure Blob Storage, the file 'my_file.xls' would be displayed as residing in the folder 'name1/name2/'.

**filter** ***(string)*** Optional filter via a regular expression which is used to exclude files from looping over them, e.g. 'a.*b' for all files starting with 'a' and ending with 'b'. Ignored if empty.

**filtered string** ***(selection)*** Indicates which part of the file name is matched against the filter expression.

- **file_name** Filtered on the name, e.g. 'myfolder/myfile.txt'
- **full_path** Filtered on the full path, e.g. 'mycontainer/myfolder/myfile.txt'
- **parent_path** Filtered on the parent folder, e.g. 'myfolder/'

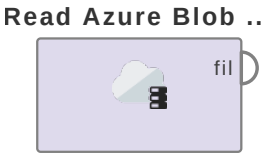
**file name macro** ***(string)*** The name of the macro which will contain the name of the current file for each file the loop iterates over, e.g. ‘myfolder/myfile.txt’

**file path macro** ***(string)*** The name of the macro which will contain the full path of the current file for each file the loop iterates over, e.g. e.g. ‘mycontainer/myfolder/myfile.txt’

**parent path macro** ***(string)*** The name of the macro which will contain the parent folder of the current file for each file the loop iterates over, e.g. e.g. ‘myfolder/’

**recursive** ***(boolean)*** If selected, the loop will also iterate over all files in all subfolders of the selected folder. Otherwise, it will only iterate over the files in the selected folder.

# Read Azure Blob Storage




This operator downloads the specified file from the Microsoft Azure Blob Storage cloud storage.

## Description

After you have configured your Azure Blob Storage account, you can load the Azure Blob Storage file with this operator.

Be aware that the operator cannot read the file as example set. For this reason, you must connect the Read Azure Blob Storage operator to another appropriate operator to read the file. For example, if you want to load an Excel file from your Azure Blob Storage, you must connect the Read Azure Blob Storage operator with the Read Excel operator to see the result.

## Output Ports

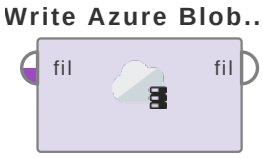
**file *(fil)*** The downloaded file object is returned here. Must be connected to a appropriate Read Operator, for example Read Excel or Read CSV.

## Parameters

**connection *(configurable)*** The connection details for the Azure Blob Storage connection have to be specified. If you have already configured an Azure Blob Storage connection, you can select it from the drop-down list. If you have not configured an Azure Blob Storage yet, select the icon to the right of the drop-down list. Create a new Azure Blob Storage connection in the Manage connections box. The account name and account key are required.

**file *(selection)*** Select the Azure Blob Storage file you want to download. Note that the concept of folders does not exist in Azure Blob Storage, so the default delimiter ('/') is used to represent them. If your file was stored as 'name1/name2/my_file.xls' on Azure Blob Storage, the file 'my_file.xls' would be displayed as residing in the folder 'name1/name2/'.

# Write Azure Blob Storage




This operator uploads the input file to the Azure Blob Storage cloud storage.

## Description

Before you can upload the input file to the selected Azure Blob Storage cloud storage, you must load it with an Open file operator.

Ensure that the correct bucket is selected, otherwise an error occurs! Buckets are container for the Azure Blob Storage objects. Each Bucket name is unique across all of Azure Blob Storage.

## Input Ports

**file *(fil)*** The file object which should be uploaded to Azure Blob Storage cloud storage. The file must be provided by an Open file operator.

## Output Ports

**file *(fil)*** The input file object is passed through and returned here.

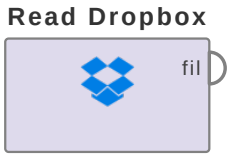
## Parameters

**connection *(configurable)*** The connection details for the Azure Blob Storage connection have to be specified. If you have already configured an Azure Blob Storage connection, you can select it from the drop-down list. If you have not configured an Azure Blob Storage yet, select the icon to the right of the drop-down list. Create a new Azure Blob Storage connection in the Manage connections box. The account name and account key are required.

**file *(selection)*** Enter the name of the file as it should be stored on Azure Blob Storage, e.g., /mycontainer/my_file.xls.

# 1.5.3 Dropbox

## Read Dropbox




This operator loads the specified file from the Dropbox cloud storage.

### Description

After you have created a Dropbox account, you can load the Dropbox file with this operator.

Be aware that the operator cannot read the file as example set. For this reason, you must connect the Read Dropbox operator to another appropriate operator to parse the file. For example, if you want to load an Excel file from your Dropbox, you must connect the Read Dropbox operator with the Read Excel operator to see the result.

### Output Ports

**file *(fil)*** The downloaded file object is returned here. Must be connected to a appropriate Read Operator, for example Read Excel or Read CSV.

### Parameters

**connection *(configurable)*** The connection details for the Dropbox connection have to be specified. If you have already configured a Dropbox connection, you can select it from the drop-down list. If you have not configured a Dropbox connection yet, select the Dropbox icon to the right of the drop-down list. Create a new Dropbox connection in the Manage connections box. An access token is required. If you don't have a valid access token, you must authenticate RapidMiner via OAuth and copy the generated token to the acess token field. Test the connection and click the Save all changes button.

**path *(selection)*** Select the Dropbox folder from the drop-down list. All available folders are displayed in the drop-down list.

**file name *(selection)*** Select the file you want to download. The available files are displayed in the drop-down list.

# Write Dropbox


This operator uploads the input file to the Dropbox cloud storage.

## Description

Before you can upload the input file to the selected Dropbox cloud storage, you must load it with an Open file operator.

## Input Ports

**file *(fil)*** The file object that should be uploaded to Dropbox cloud storage. The file must be provided by an Open file operator.

## Output Ports

**file *(fil)*** The input file object is passed through and returned here.

## Parameters

**connection *(configurable)*** The connection details for the Dropbox connection have to be specified. If you have already configured a Dropbox connection, you can select it from the drop-down list. If you have not configured a Dropbox connection yet, select the Dropbox icon to the right of the drop-down list. Create a new Dropbox connection in the Manage connections box. An access token is required. If you don't have a valid access token, you must authenticate RapidMiner via OAuth and copy the generated token to the acess token field. Test the connection and click the Save all changes button.

**path *(selection)*** Select the Dropbox folder from the drop-down list. All available folders are displayed in the drop-down list.

**file name *(string)*** The file name of the file that is written to Dropbox cloud storage. This entry is optional. If you don't enter a name, the original input file name is taken.

**overwrite *(boolean)*** If the checkbox is activated, the input file will overwrite existing files with the same file name. If the checkbox is not activated, existing files with the same name are not overwritten. The file name will be enhanced by a counter. By default the option is deactivated. Note that uploading a file that has no changes in comparison to the destination will neither update the time stamp nor create a new file with a counter.

# 1.5.4 Google Storage

## Loop Google Storage


This operator loops over all files in the specified bucket/folder from the Google Cloud Storage.

### Description

After you have configured your Google account, you can process all Google Storage files within the selected folder.

Be aware that the operator cannot read the file as example set. For this reason, you must connect the file input in the inner process of this operator to another appropriate operator to process the file. For example, if you want to load Excel files from your Google Storage folder, you must connect the file input in the inner process with the Read Excel operator.

### Input Ports

**in** ***(in )*** Optional input data which is delivered to the inner process.

### Output Ports

**out** ***(out)*** Output data of the inner process.

### Parameters

**connection** ***(configurable)*** The connection details for the Google Storage connection have to be specified. If you have already configured a Google Storage connection, you can select it from the drop-down list. If you have not configured a Google Storage connection yet, select the icon to the right of the drop-down list. Create a new Google Storage connection in the Manage connections box. The access token / private key and project ID are required.

**folder** ***(selection)*** Provide the name of the Google Storage folder over which you want to loop.

**filter** ***(string)*** Optional filter via a regular expression which is used to exclude files from looping over them, e.g. 'a.*b' for all files starting with 'a' and ending with 'b'. Ignored if empty.

**filtered string** ***(selection)*** Indicates which part of the file name is matched against the filter expression.

- **file_name** Filtered on the name, e.g. 'myfile.txt'
- **full_path** Filtered on the full path, e.g. 'mybucket/myfolder/myfile.txt'
- **parent_path** Filtered on the parent folder, e.g. 'myfolder/'

**file name macro** ***(string)*** The name of the macro which will contain the name of the current file for each file the loop iterates over, e.g. 'myfile.txt'

**file path macro** ***(string)*** The name of the macro which will contain the full path of the current file for each file the loop iterates over, e.g. 'mybucket/myfolder/myfile.txt'

**parent path macro *(string)*** The name of the macro which will contain the parent folder of the current file for each file the loop iterates over, e.g. ‘myfolder/’

**recursive *(boolean)*** If selected, the loop will also iterate over all files in all subfolders of the selected folder. Otherwise, it will only iterate over the files in the selected folder.

# Read Google Storage


This operator downloads the specified file from the Google Cloud Storage.

## Description

After you have configured your Google Storage account, you can load the Google Storage file with this operator.

Be aware that the operator cannot read the file as example set. For this reason, you must connect the Read Google Storage operator to another appropriate operator to read the file. For example, if you want to load an Excel file from your Google Storage, you must connect the Read Google Storage operator with the Read Excel operator to see the result.

## Output Ports

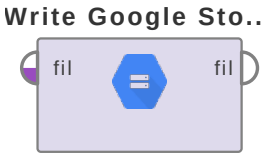
**file *(fil)*** The downloaded file object is returned here. Must be connected to a appropriate Read Operator, for example Read Excel or Read CSV.

## Parameters

**connection *(configurable)*** The connection details for the Google Storage connection have to be specified. If you have already configured a Google Storage connection, you can select it from the drop-down list. If you have not configured a Google Storage connection yet, select the icon to the right of the drop-down list. Create a new Google Storage connection in the Manage connections box. The access token / private key and project ID are required.

**file *(selection)*** Select the Google Storage file you want to download.

# Write Google Storage




This operator uploads the input file to the Google Cloud Storage.

## Description

Before you can upload the input file to the selected Google Storage account, you must load it with an Open file operator.

Be aware that the operator cannot write the example set as file. For this reason, you must connect before the Write Google Storage operator an appropriate operator to write the file. For example, if you want to save an Excel file to your Google Storage, you must connect before the Write Google Storage operator a Write Excel operator to see the result.

## Input Ports

**file *(fil)*** The file object which should be uploaded to Google Storage. The file must be provided by an Open file operator.

## Output Ports

**file *(fil)*** The input file object is passed through and returned here.

## Parameters

**connection *(configurable)*** The connection details for the Google Storage connection have to be specified. If you have already configured a Google Storage connection, you can select it from the drop-down list. If you have not configured a Google Storage connection yet, select the icon to the right of the drop-down list. Create a new Google Storage connection in the Manage connections box. The access token / private key and project ID are required.

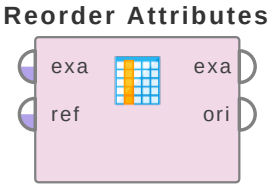
**file *(selection)*** Enter the name of the file as it should be stored on Google Storage, e.g., "/mybucket/myfolder/my_file.xls".

**content type *(string)*** This option is optional. Enter the MIME type of the upload file, e.g., "text/xml".

# 2Blending

## 2.1 Attributes

### Reorder Attributes




This operator allows to reorder regular Attributes of an ExampleSet. Reordering can be done alphabetically, by user specification (including Regular Expressions) or with a reference ExampleSet.

#### Description

This operator allows to change the ordering of *regular* Attributes of an ExampleSet. Therefore, two different order modes may be selected in the parameter sort_mode. If sort mode alphabetically is chosen attributes are sorted alphabetically according to the selected sort_direction. If sort mode user specified is chosen the user can specify rules that define how attributes should be ordered. If sort mode reference data is chosen the input ExampleSet will be sorted according to the order of reference ExampleSet.

Note that *special* attributes will *not* be considered by this operator. If they also should be reordered set them to regular with *Set Role* operator before.

#### Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with data.

**reference data *(ref)*** This input port expects an ExampleSet. If sort mode is set to reference data and this port is connected, the ExampleSet from first port sorted will be sorted according to the order of attributes from this ExampleSet.

#### Output Ports

**example set *(exa)*** The ExampleSet with reordered attributes is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

#### Parameters

**sort mode *(selection)*** This parameter allows you to select the method you want to use for reordering attributes. It has the following options:

- **user specified** This option allows to specify rules that define how the attributes should be reordered. When this option is selected another parameter (attribute ordering) becomes visible in the Parameters panel. This is the default option.
- **alphabetically** This option simply reorders all regular attributes alphabetically according to the selected sort direction.
- **reference data** This option allows to reorder all *regular* attributes according to the order of all *regular* attributes of the reference ExampleSet. If special attributes should also be considered, set them to regular before using this operator.

**sort direction** ***(selection)*** The direction of matched attribute groups to be sorted. If sort mode is alphabetically all regular attributes are sorted according to this direction. If sort mode is user specified, attributes that match a Regular Expression and all unmachted attributes are sorted according to this parameter. Moreover if sort mode is set to reference data all attributes that could not be found in the reference ExampleSet are sorted according to this parameter.

- **ascending** Sort attribute names ascending. This is the default option.
- **descending** Sort attribute names descending.
- **none** Apply no sorting at all.

**attribute ordering** ***(string)*** This parameter allows the user to specify rules that define how attributes should be ordered. If the parameter use regular expressions is checked all specified rules are treated as Regular Expressions.

**handle unmachted** ***(selection)*** Defines how unmachted attributes should be handled. Unmachted attributes can occur if one or more Attribute do not match the rules that the user did provide with the attribute ordering parameter or if one or more Attribute cannot be found in the reference ExampleSet. If they are kept (prepend,append) they will be sorted according to the selected sort direction.

- **append** Append all attributes that are not covered by the provided sorting rules.
- **prepend** Prepend all attributes that are not covered by the provided sorting rules.
- **remove** Remove all attributes that are not covered by the provided sorting rules.

**use regular expressions** ***(boolean)*** If this parameter is checked all rules created with the attribute ordering parameter are treated as Regular Expressions.

## Tutorial Processes

### Selecting attributes by specifying regular expressions matching their names

In the given Example process the Labor-Negotiations ExampleSet is loaded using the Retrieve operator. Then Reorder Attribute operator is applied on it. Have a look at the Parameters panel of the Reorder Attributes operator. Here is a stepwise explanation of this process.

The sort mode parameter is set to ‘user specified’. This allows the user to specify exact rules on how the attributes should be ordered.

The attribute ordering parameter has two rules set. First rule is ‘contrib-.*’ and second rule is ‘.*-.*’

The first rule ’contrib-.*’ that attributes starting with ‘contrib-’ should be ordered in front.

Since this expression matches two attributes both are sorted in descending order (see sort direction). ’.*-.*’ means all attributes that have a ’-’ in their name without those that already have been matched be the first rule.

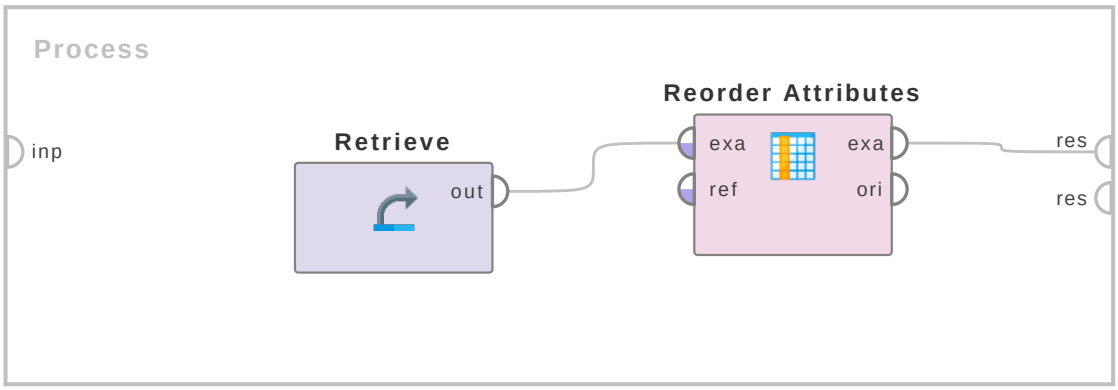
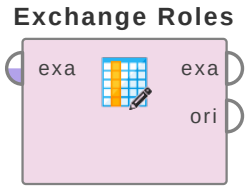



Figure 2.1: Tutorial process ‘Selecting attributes by specifying regular expressions matching their names’.

Only duration, pension and vacation do not match these two rules.
They are also sorted according to the sort direction and appended like it is defined with the handle unmachted parameter.

# 2.1.1 Names and Roles

## Exchange Roles




This operator exchanges the roles of two attributes.

### Description

The Exchange Roles operator exchanges the roles of the two specified attributes i.e. it assigns the role of the first attribute to the second attribute and vice versa. This can be useful, for example, to exchange the roles of a label with a regular attribute (or vice versa), or a label with a batch attribute, a label with a cluster etc. For more information about roles please study the description of the Set Role operator.

### Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

### Output Ports

**example set output *(exa)*** The roles of the specified attributes are exchanged and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

### Parameters

**first attribute *(string)*** This parameter specifies the name of the first attribute for the attribute role exchange.

**second attribute *(string)*** This parameter specifies the name of the second attribute for the attribute role exchange.

### Tutorial Processes

#### Exchanging roles of attributes of the Golf data set

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the roles of the Play and Outlook attributes are label and regular respectively. The Exchange Roles operator is applied on the 'Golf' data set to exchange the roles of these attributes. the first attribute and second attribute parameters are set to 'Play' and 'Outlook' respectively. The resultant ExampleSet can be seen in the Results

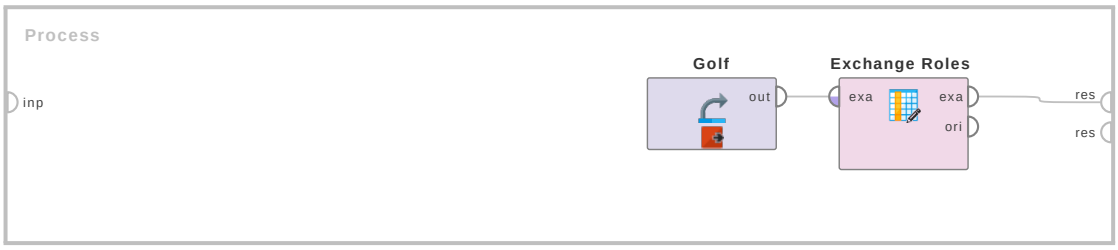
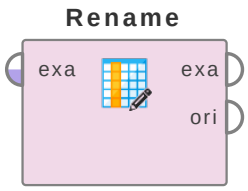



Figure 2.2: Tutorial process ‘Exchanging roles of attributes of the Golf data set’.

Workspace. You can see that now the role of the Play attribute is regular and the role of the Outlook attribute is label.

# Rename




This operator can be used to rename one or more attributes of an ExampleSet.

## Description

The Rename operator is used for renaming one or more attributes of the input ExampleSet. Please keep in mind that attribute names must be unique. The Rename operator has no impact on the type or role of an attribute. For example if you have an attribute named 'alpha' of *integer* type and *regular* role. Renaming the attribute to 'beta' will just change its name. It will retain its type *integer* and role *regular*. To change the role of an operator, use the Set Role operator. Many type conversion operators are available for changing the type of an attribute at 'Data Transformation/Type Conversion'.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with data for input because attributes are specified in its meta data. The Retrieve operator provides meta data along-with data.

## Output Ports

**example set *(exa)*** The ExampleSet with renamed attributes is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**old name *(string)*** This parameter is used to select the attribute whose name is to be changed.

**new name *(string)*** The new name of the attribute is specified through this parameter. Name can also include special characters.

**rename additional attributes *(string)*** To rename more than one attributes click on the *Edit List* button. Here you can select attributes and assign new names to them.

## Tutorial Processes

### Renaming multiple attributes

The 'Golf ' data set is used in this Example Process.The 'Play' attribute is renamed to 'Game' and the 'Wind' attribute is renamed to '#*#'. The 'Wind' attribute is renamed to '#*#', just to show

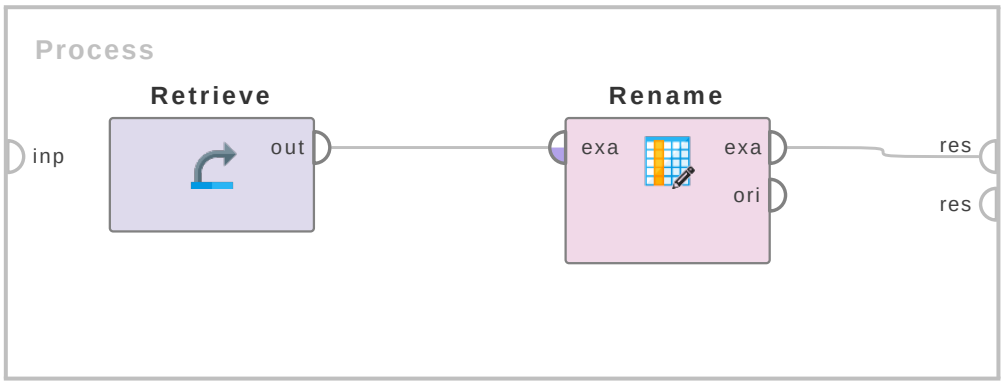
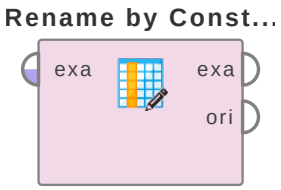



Figure 2.3: Tutorial process ‘Renaming multiple attributes’.

that special characters can also be used to rename attributes. However, attribute names should always be meaningful and should be relevant to the type of information stored in them.

# Rename by Constructions




This operator renames the regular attributes of an ExampleSet by their construction descriptions if available.

## Description

The Rename by Constructions operator replaces the names of regular attributes of the given ExampleSet by their corresponding construction descriptions if the attribute was constructed at all. Please study the attached Example Process for better understanding.

Please keep in mind that attribute names must be unique. The Rename by Constructions operator has no impact on the type or role of an attribute. For example if you have an attribute named 'alpha' of *integer* type and *regular* role. Renaming the attribute to 'beta' will just change its name. It will retain its type *integer* and role *regular*. To change the role of an operator, use the Set Role operator. Many type conversion operators are available for changing the type of an attribute at 'Data Transformation/Type Conversion'.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Write Constructions operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.

## Output Ports

**example set output *(exa)*** The ExampleSet with renamed attributes is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Tutorial Processes

### Renaming attributes by their construction descriptions

This Example Process shows how the Rename by Constructions operator can be used for renaming attributes. The 'Sonar' data set is loaded using the Retrieve operator. The Rename by Generic names operator is applied on this ExampleSet to rename the attributes with the generic stem 'att'. This ExampleSet is provided as input to the Write Constructions operator. The attribute constructions file parameter is set to 'D:\attributes' thus a file named 'attributes' is created (if it does not already exist) in the 'D' drive of your computer. You can open the written file and make changes in it (if required). A breakpoint is inserted here so that you can have a look at the constructions file. You can see that each line in the file holds the construction description of one attribute. You can see that the attribute names are of the form att1, att2 and so on. The attribute constructions are of the form attribute_1, attribute_2 and so on. The Rename by Constructions

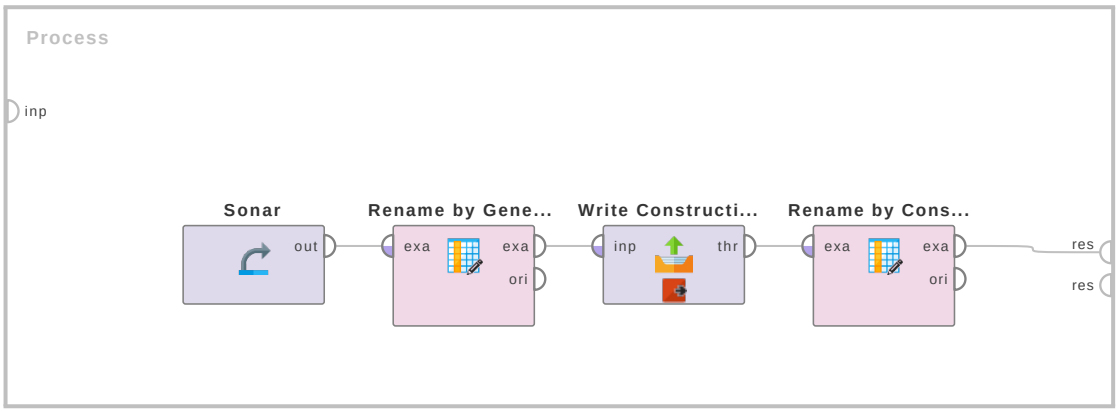
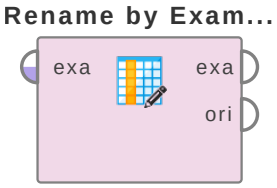



Figure 2.4: Tutorial process 'Renaming attributes by their construction descriptions'.

operator is applied on this ExampleSet. This operator will replace the attribute names by the attribute constructions. Which means that the attributes that are currently named as att1, att2 etc will be renamed to attribute_1, attribute_2 etc. You can verify this by viewing the resultant ExampleSet in the Results Workspace.

# Rename by Example Values




This operator renames the attributes of an ExampleSet by assigning the values of a specified example as attribute names and deleting that example from the ExampleSet.

## Description

The Rename by Example Values operator uses the values of the specified example of the ExampleSet as new attribute names. The *row number* parameter specifies which row should be used as attribute names. Please note that all regular and special attributes are renamed. Moreover, the example is deleted from the ExampleSet. This operator can be useful in cases when an example holds the names of the attributes.

Please keep in mind that attribute names must be unique. The Rename by Example Values operator has no impact on the type or role of an attribute. For example if you have an attribute named 'alpha' of *integer* type and *regular* role. Renaming the attribute to 'beta' will just change its name. It will retain its type *integer* and role *regular*. To change the role of an operator, use the Set Role operator. Many type conversion operators are available for changing the type of an attribute at 'Data Transformation/Type Conversion'.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.

## Output Ports

**example set output *(exa)*** The ExampleSet with renamed attributes is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**row number *(integer)*** This parameter specifies which row values should be used as attribute names. Please note that counting starts with 1.

## Tutorial Processes

### Renaming all attributes by example values

This Example Process starts with the Subprocess operator. The Subprocess operator delivers an ExampleSet. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that currently the attributes names are 'label', 'att1' and 'att2'. The first example has the

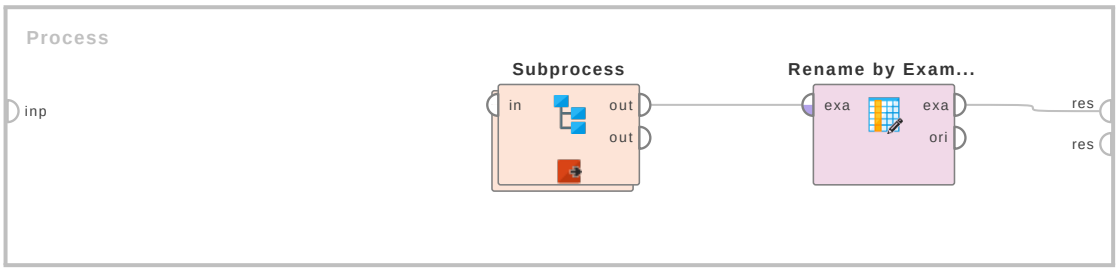
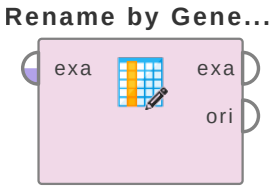



Figure 2.5: Tutorial process ‘Renaming all attributes by example values’.

values ‘new_label’, ‘new_name1’ and ‘new_name2’. The Rename by Example Values operator is applied on this ExampleSet to set the values of the first example as attribute names. The row number parameter is set to 1. After execution of the process you will see that the attributes have been renamed accordingly. Moreover the first example has been removed from the ExampleSet.

# Rename by Generic Names




This operator renames the selected attributes of the given ExampleSet to a set of generic names like att1, att2, att3 etc.

## Description

The Rename by Generic Names operator renames the selected attributes of the given ExampleSet to a set of generic names like att1, att2, att3 etc. The *generic name stem* parameter specifies the name stem which should be used for building generic names. For example, using 'att' as stem would lead to 'att1', 'att2', etc. as attribute names.

The Rename by Generic Names operator has no impact on the type or role of an attribute. For example if you have an attribute named 'alpha' of *integer* type and *regular* role. Renaming the attribute to 'beta' will just change its name. It will retain its type *integer* and role *regular*. To change the role of an operator, use the Set Role operator. Many type conversion operators are available for changing the type of an attribute at 'Data Transformation/Type Conversion'.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.

## Output Ports

**example set output *(exa)*** The ExampleSet with renamed attributes is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet.This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute *(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *parameter* attribute if the meta data is known.

**attributes *(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list, which is the list of selected attributes.

**regular expression *(string)*** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression *(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter *(except regular expression*) becomes visible in the Parameters panel.

**except regular expression *(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the *regular expression* parameter).

**value type *(selection)*** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception *(boolean)*** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type *(selection)*** The attributes matching this type will not be selected even if they match the previously mentioned type i.e. *value type* parameter's value.

**block type *(selection)*** The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception *(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type *(selection)*** The attributes matching this block type will be not be selected even if they match the previously mentioned block type i.e. *block type* parameter's value.

**numeric condition *(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '> 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '> 6 && < 11' or '<= 5 || < 0'. But && and || cannot be used together in one numeric condition. Conditions like '(> 0 && < 2) || (>10 && < 12)' are not allowed because they use both && and ||. Use a blank space after '>', '=' and '<' e.g. '<5' will not work, so use '< 5' instead.

**include special attributes *(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection *(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**generic name stem *(string)*** This parameter specifies the name stem which should be used for building generic names. For example, using 'att' as stem would lead to 'att1', 'att2', etc. as attribute names.

## Tutorial Processes

### Renaming attributes of the Sonar data set

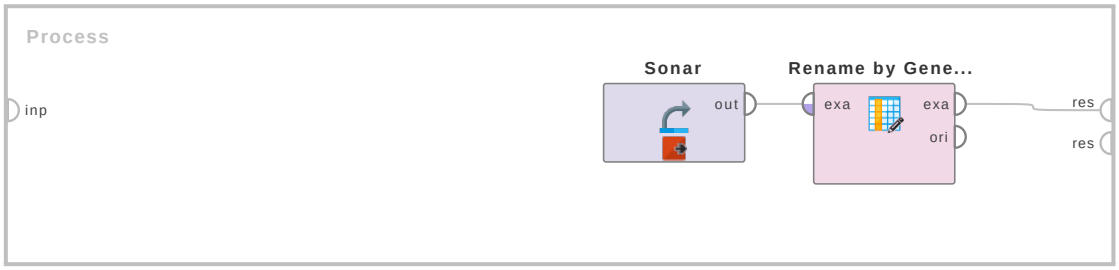
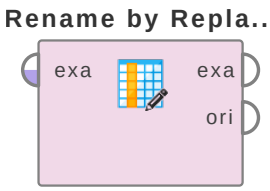



Figure 2.6: Tutorial process 'Renaming attributes of the Sonar data set'.

The ‘Sonar’ data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the ExampleSet. You can see that the ExampleSet has 60 regular attributes with names like attribute_1, atribute_2 etc. The Rename by Generic Names operator is applied on it. The attribute filter type parameter is set to ‘all’ thus all attributes can be renamed by this operator. The generic name stem parameter is set to ‘att’. Thus the attributes are renamed to format att1, att2 and so on. This can be verified by seeing the results in the Results Workspace. You can see that the label attribute is not renamed. This is so because the include special attributes parameter was not set to true.

# Rename by Replacing




This operator can be used to rename a set of attributes by replacing parts of the attribute names by a specified replacement.

## Description

The Rename by Replacing operator replaces parts of the attribute names by the specified replacement. This operator is used mostly for removing unwanted parts of attribute names like whitespaces, parentheses, or other unwanted characters. The *replace what* parameter defines that part of the attribute name that should be replaced. It can be defined as a regular expression which is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. The *replace by* parameter can be defined as an arbitrary string. Empty strings are also allowed. Capturing groups of the regular expression of the *replace what* parameter can be accessed with $1, $2, $3 etc. Please study the attached Example Process for more understanding.

Please keep in mind that attribute names must be unique. The Rename by Replacing operator has no impact on the type or role of an attribute. For example if you have an attribute named 'alpha' of *integer* type and *regular* role. Renaming the attribute to 'beta' will just change its name. It will retain its type *integer* and role *regular*. To change the role of an operator, use the Set Role operator. Many type conversion operators are available for changing the type of an attribute at 'Data Transformation/Type Conversion'.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.

## Output Ports

**example set output *(exa)*** The ExampleSet with renamed attributes is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet.This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.

- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute *(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *parameter* attribute if the meta data is known.

**attributes *(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list, which is the list of selected attributes.

**regular expression *(string)*** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression *(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter *(except regular expression*) becomes visible in the Parameters panel.

**except regular expression *(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the *regular expression* parameter).

**value type *(selection)*** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will not be selected even if they match the previously mentioned type i.e. *value type* parameter's value.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be not be selected even if they match the previously mentioned block type i.e. *block type* parameter's value.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '> 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '> 6 && < 11' or '<= 5 || < 0'. But && and || cannot be used together in one numeric condition. Conditions like '(> 0 && < 2) || (>10 && < 12)' are not allowed because they use both && and ||. Use a blank space after '>', '=' and '<' e.g. '<5' will not work, so use '< 5' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**replace what** ***(string)*** The *replace what* parameter defines that part of the attribute name that should be replaced. It can be defined as a *regular expression*. Capturing groups of the regular expression of the *replace what* parameter can be accessed in the *replace by* parameter with $1, $2, $3 etc.

**replace by** ***(string)*** The *replace by* parameter can be defined as an arbitrary string. Empty strings are also allowed. Capturing groups of the regular expression of the *replace what* parameter can be accessed with $1, $2, $3 etc.

## Tutorial Processes

### Renaming attributes of the Sonar data set

The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the ExampleSet. You can see that the ExampleSet has 60 regular attributes with

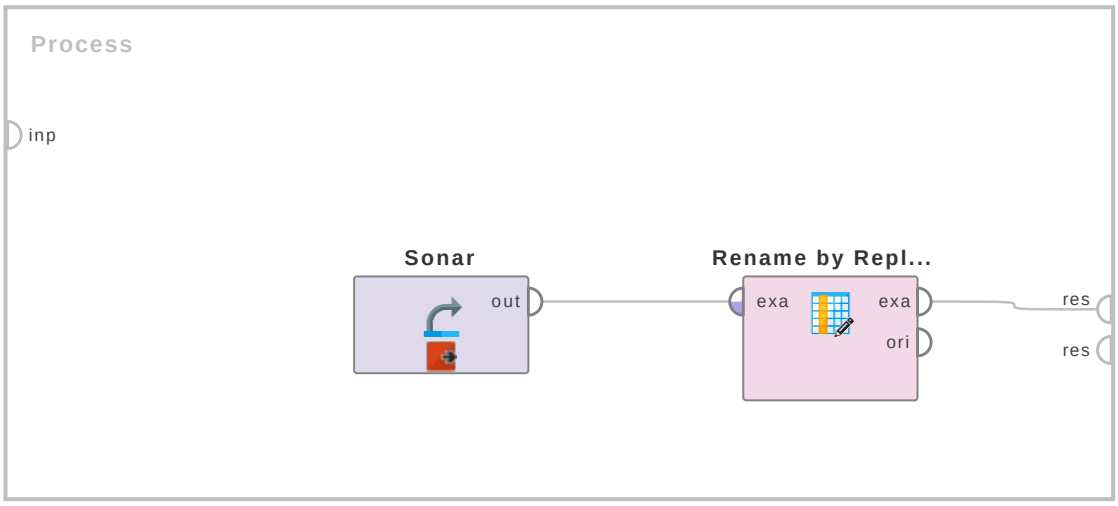
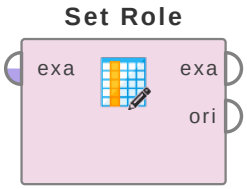



Figure 2.7: Tutorial process ‘Renaming attributes of the Sonar data set’.

names like attribute_1, atribute_2 etc. The Rename by Replacing operator is applied on it. The attribute filter type parameter is set to ‘all’ thus all attributes can be renamed by this operator. The replace what parameter is set to the regular expression: ‘(att)ribute_’. The brackets are used for specifying the capturing group which can be accessed in the replace by parameter with $1. The replace by parameter is set to ‘$1-’. Wherever ‘attribute_’ is found in names of the ‘Sonar’ attributes, it is replaced by the first capturing group and a dash i.e. ‘att-’. Thus attributes are renamed to format att-1, att-2 and so on. This can be verified by seeing the results in the Results Workspace.

# Set Role




This Operator is used to change the role of one or more Attributes.

## Description

The role of an Attribute describes how other Operators handle this Attribute. The default role is regular, other roles are classified as special. An ExampleSet can have many special Attributes, but each special role can only appear once. If a special role is assigned to more than one Attribute, all roles will be changed to regular except for the last Attribute. The different types of roles are explained below in the parameter section.

## Differentiation

*Renaming Operators*
There are several Operator for renaming Attributes (e.g., *Rename* , *Rename by Replacing*, ...). Those only change the name of the Attribute and not its role.

- **Generate ID**

  This Operator creates a new Attribute with the special role *id*. In contrast to *Set Role* this Operator will overwrite an existing Attribute with the *id* role. Applying *Set Role* to change the role to *regular* will keep the original Attribute.

  See page 232 for details.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet.

## Output Ports

**example set *(exa)*** The ExampleSet with modified role(s) is output of this port

**original *(ori)*** The ExampleSet, that was given as input is passed through without changes.

## Parameters

**attribute name** The name of the Attribute which role should be changed. The name can be selected from the dropdown menu or manual typed.

**target role** The target role of the selected Attribute is the new role assigned to it. Following target roles are possible:

- **regular** Attributes without a special role. Regular Attributes are used as input variables for learning tasks.

- **id** This is a special role. An Attribute with the *id* role acts as an identifier for the Examples. It should be unique for all Examples. Different Blending Operators (*Join, Union, Transpose, Pivot, ...*) uses the *id* Attribute to perform their tasks.
- **label** This is a special role. An Attribute with the *label* role acts as a target Attribute for learning Operators. The *label* is also often called 'target variable' or 'class'.
- **prediction** This is a special role. An Attribute with the *prediction* role is the result of an application of a learning model. The Apply Model Operator adds for example a *prediction* Attribute to the ExampleSet. To evaluate the performance of a model, a *label* and a *prediction* Attribute is necessary.
- **cluster** This is a special role. An Attribute with the cluster role indicates the membership of an ExampleSet to a particular cluster. For example the k-Means Operator adds an Attribute with the cluster role.
- **weight** This is a special role. An Attribute with the weight role indicates the weight of the Examples with regard to the label. Weights are used in learning processes to set the importance of Examples. Weights can also be used to evaluate the performance of models; there they assign a severness for misclassification of single Examples.
- **batch** This is a special role. An Attribute with the batch role indicates the memeberships to a specific batch.
- **user defined** Any role can be assigned to an Attribute by typing in the textbox. User defined roles are special roles, so one specific role cannot be assigned to more than one Attribute. Attributes with user defined roles are ignored in learning processes. So an Attribute with a user defined role is ignored in a learning processes but remains in the ExampleSet.

**set additional roles** This parameter is used to set the role of more than one Attribute at once. A click on *Edit List* opens a menu with Attribute name and target role pairs. They can be used in the same way as the above described parameters.

## Tutorial Processes

### Set Role on Titanic Data Set

This tutorial Process shows the basic usage of the Set Role Operator. First the Titanic data set is retrieved from the Samples folder. Then the role of several Attributes are set. An explanation is given of which Attribute is set to which role and the reason for that is given in the comments.

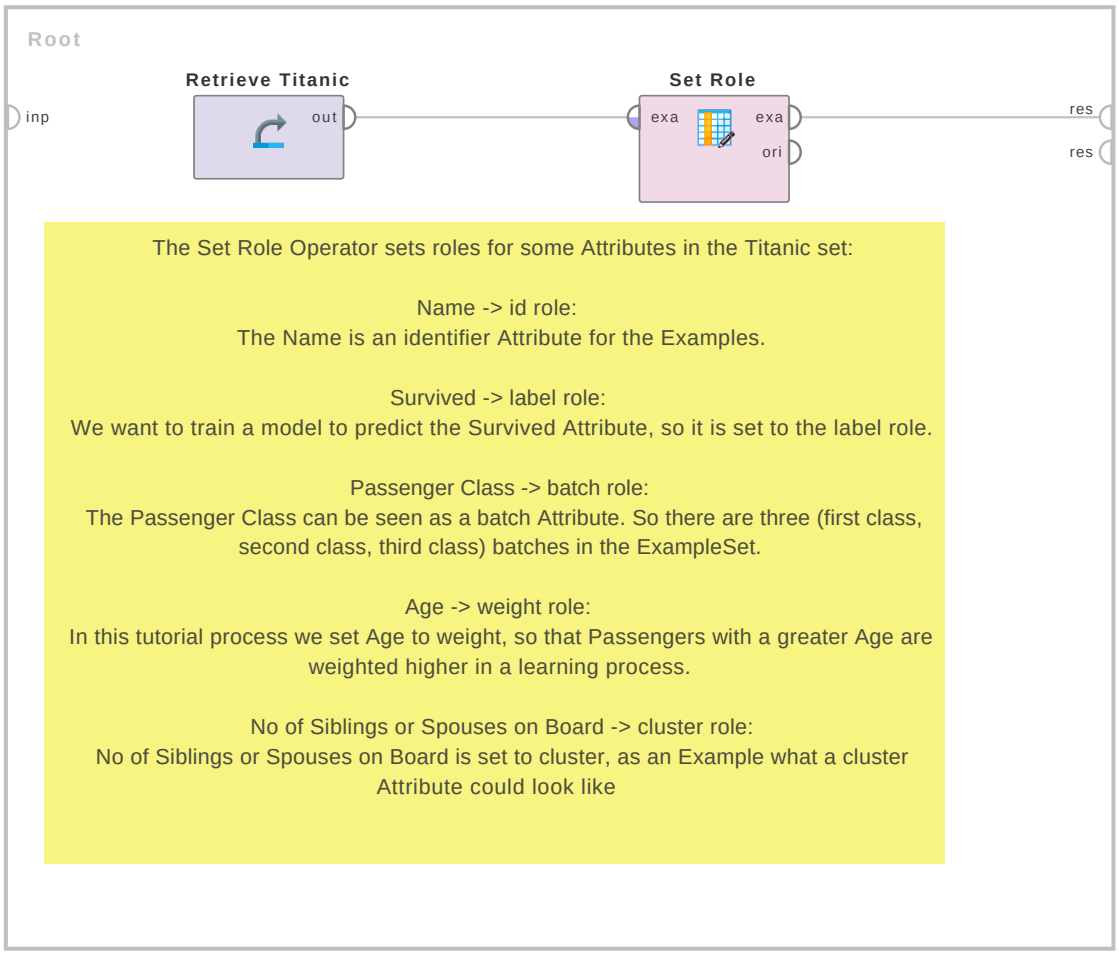
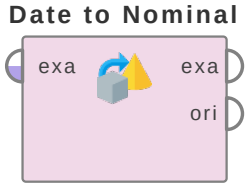



Figure 2.8: Tutorial process ‘Set Role on Titanic Data Set’.

## 2.1.2 Types

### Date to Nominal




This operator parses the date values of the specified date attribute with respect to the given date format string and transforms the values into nominal values.

#### Description

The Date to Nominal operator transforms the specified date attribute and writes a new nominal attribute in a user specified format. This conversion is done with respect to the specified date format string that is specified by the *date format* parameter. This operator might be useful for

time base OLAP to change the granularity of the time stamps from day to week or month. The date attribute is selected by the *attribute name* parameter. The old date attribute will be removed and replaced by a new nominal attribute if the *keep old attribute* parameter is not set to true. The understanding of Date and Time patterns is very important for using this operator properly.

### Date and Time Patterns

This section explains the date and time patterns. Understanding of date and time patterns is necessary especially for specifying the *date format* string in the date format parameter. Within date and time pattern strings, unquoted letters from 'A' to 'Z' and from 'a' to 'z' are interpreted as pattern letters that represent the components of a date or time. Text can be quoted using single quotes (') to avoid interpretation as date or time components. All other characters are not interpreted as date or time components; they are simply matched against the input string during parsing.

Here is a brief description of the defined pattern letters. The *format types* like 'Text', 'Number', 'Year', 'Month' etc are described in detail after this section.

- G: This pattern letter is the era designator. For example: AD, BC etc. It follows the rules of 'Text' format type.
- y: This pattern letter represents year. yy represents year in two digits e.g. 96 and yyyy represents year in four digits e.g. 1996. This pattern letter follows the rules of the 'Year' format type.
- M: This pattern letter represents the month of the year. It follows the rules of the 'Month' format type. Month can be represented as; for example; March, Mar or 03 etc.
- w: This pattern letter represents the week number of the year. It follows the rules of the 'Number' format type. For example, the first week of January can be represented as 01 and the last week of December can be represented as 52.
- W: This pattern letter represents the week number of the month. It follows the rules of the 'Number' format type. For example, the first week of January can be represented as 01 and the forth week of December can be represented as 04.
- D: This pattern letter represents the day number of the year. It follows the rules of the 'Number' format type. For example, the first day of January can be represented as 01 and last day of December can be represented as 365 (or 366 in case of a leap year).
- d: This pattern letter represents the day number of the month. It follows the rules of the 'Number' format type. For example, the first day of January can be represented as 01 and the last day of December can be represented as 31.
- F: This pattern letter represents the day number of the week. It follows the rules of the 'Number' format type.
- E: This pattern letter represents the name of the day of the week. It follows the rules of the 'Text' format type. For example, Tuesday or Tue etc.
- a: This pattern letter represents the AM/PM portion of the 12-hour clock. It follows the rules of the 'Text' format type.
- H: This pattern letter represents the hour of the day (from 0 to 23). It follows the rules of the 'Number' format type.

- k: This pattern letter represents the hour of the day (from 1 to 24). It follows the rules of the ‘Number’ format type.
- K: This pattern letter represents the hour of the day for 12-hour clock (from 0 to 11). It follows the rules of the ‘Number’ format type.
- h: This pattern letter represents the hour of the day for 12-hour clock (from 1 to 12). It follows the rules of the ‘Number’ format type.
- m: This pattern letter represents the minutes of the hour (from 0 to 59). It follows the rules of the ‘Number’ format type.
- s: This pattern letter represents the seconds of the minute (from 0 to 59). It follows the rules of the ‘Number’ format type.
- S: This pattern letter represents the milliseconds of the second (from 0 to 999). It follows the rules of the ‘Number’ format type.
- z: This pattern letter represents the time zone. It follows the rules of the ‘General Time Zone’ format type. Examples include Pacific Standard Time, PST, GMT-08:00 etc.
- Z: This pattern letter represents the time zone. It follows the rules of the ‘RFC 822 Time Zone’ format type. Examples include -08:00 etc.

Please note that all other characters from ‘A’ to ‘Z’ and from ‘a’ to ‘z’ are reserved. Pattern letters are usually repeated, as their number determines the exact presentation. Here is the explanation of various *format types*:

- Text: For formatting, if the number of pattern letters is 4 or more, the full form is used; otherwise a short or abbreviated form is used (if available). For parsing, both forms are acceptable independent of the number of pattern letters.
- Number: For formatting, the number of pattern letters is the minimum number of digits. The numbers that are shorter than this minimum number of digits are zero-padded to this amount. For example if the minimum number of digits is 3 then the number 5 will be changed to 005. For parsing, the number of pattern letters is ignored unless it is needed to separate two adjacent fields.
- Year: If the underlying calendar is the Gregorian calendar, the following rules are applied:
  - For formatting, if the number of pattern letters is 2, the year is truncated to 2 digits; otherwise it is interpreted as a ‘Number’ format type.
  - For parsing, if the number of pattern letters is more than 2, the year is interpreted literally, regardless of the number of digits. So using the pattern ‘MM/dd/yyyy’, the string ‘01/11/12’ parses to ‘Jan 11, 12 A.D’.
  - For parsing with the abbreviated year pattern (’y’ or ‘yy’), this operator must interpret the abbreviated year relative to some century. It does this by adjusting dates to be within 80 years before and 20 years after the time the operator is created. For example, using a pattern of ‘MM/dd/yy’ and the operator created on Jan 1, 1997, the string ‘01/11/12’ would be interpreted as Jan 11, 2012 while the string ‘05/04/64’ would be interpreted as May 4, 1964. During parsing, only strings consisting of exactly two digits will be parsed into the default century. Any other numeric string, such as a one digit string, a three or more digit string, or a two digit string that is not all digits (for example, ‘-1’), is interpreted literally. So ‘01/02/3’ or ‘01/02/003’ are parsed, using the same pattern, as ‘Jan 2, 3 AD’. Likewise, ‘01/02/-3’ is parsed as ‘Jan 2, 4 BC’.

Otherwise, if the underlying calendar is not the Gregorian calendar, calendar system specific forms are applied. If the number of pattern letters is 4 or more, a calendar specific long form is used. Otherwise, a calendar short or abbreviated form is used.

- Month: If the number of pattern letters is 3 or more, the month is interpreted as 'Text' format type otherwise, it is interpreted as a 'Number' format type.
- General time zone: Time zones are interpreted as 'Text' format type if they have names. It is possible to define time zones by representing a GMT offset value. RFC 822 time zones are also acceptable.
- RFC 822 time zone: For formatting, the RFC 822 4-digit time zone format is used. General time zones are also acceptable.

This operator also supports localized date and time pattern strings by defining the *locale* parameter. In these strings, the pattern letters described above may be replaced with other, locale-dependent pattern letters.

The following examples show how date and time patterns are interpreted in the U.S. locale. The given date and time are 2001-07-04 12:08:56 local time in the U.S. Pacific Time time zone.

- 'yyyy.MM.dd G 'at' HH:mm:ss z': 2001.07.04 AD at 12:08:56 PDT
- 'EEE, MMM d, yy': Wed, Jul 4, '01
- 'h:mm a': 12:08 PM
- 'hh 'oclock' a, zzzz': 12 oclock PM, Pacific Daylight Time
- 'K:mm a, z': 0:08 PM, PDT
- 'yyyy.MMMMM.dd GGG hh:mm aaa': 2001.July.04 AD 12:08 PM
- 'EEE, d MMM yyyy HH:mm:ss Z': Wed, 4 Jul 2001 12:08:56 -0700
- 'yyMMddHHmmssZ': 010704120856-0700
- 'yyyy-MM-dd'T'HH:mm:ss.SSSZ': 2001-07-04T12:08:56.235-0700

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input. The ExampleSet should have at least one date attribute because if there is no such attribute, the use of this operator does not make sense.

## Output Ports

**example set output *(exa)*** The selected date attribute is converted to a nominal attribute according to the specified date format string and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute name *(string)*** The name of the date attribute is specified here. The attribute name can be selected from the drop down box of the *attribute name* parameter if the meta data is known.

**date format** This is the most important parameter of this operator. It specifies the date time format of the desired nominal attribute. This date format string specifies what portion of the date attribute should be stored in the nominal attribute. Date format strings are discussed in detail in the description of this operator.

**locale *(selection)*** This is an expert parameter. A long list of locales is provided; users can select any of them.

**keep old attribute *(boolean)*** This parameter indicates if the original date attribute should be kept or it should be discarded.

## Tutorial Processes

### Introduction to the Date to Nominal operator

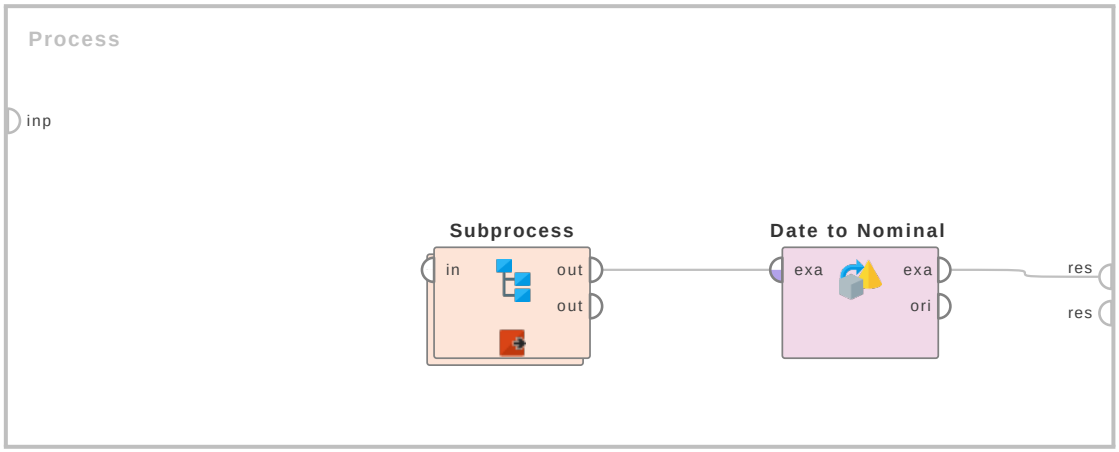


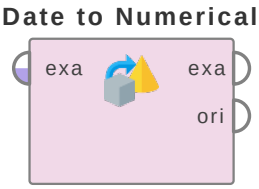

Figure 2.9: Tutorial process 'Introduction to the Date to Nominal operator'.

This Example Process starts with a Subprocess operator. The subprocess delivers an ExampleSet with just a single attribute. The name of the attribute is 'deadline_date'. The type of the attribute is date. A breakpoint is inserted here so that you can view the ExampleSet. As you can see, all the examples of this attribute have both date and time information. The Date to Nominal operator is applied on this ExampleSet to change the type of the 'deadline_date' attribute from date to nominal type. Have a look at the parameters of the Date to Nominal operator. The attribute name parameter is set to 'deadline_date'. The date format parameter is set to 'EEEE', here is an explanation of this date format string:

'E' is the pattern letter used for the representation of the name of the day of the week. As explained in the description, if the number of pattern letters is 4 or more, the full form is used. Thus 'EEEE' is used for representing the day of the week in full form e.g. Monday, Tuesday etc. Thus the date attribute is changed to a nominal attribute which has only name of days as possible

values. Please note that this date format string is used for specifying the format of the nominal values of the new nominal attribute of the input ExampleSet.

# Date to Numerical




This operator changes the type of the selected date attribute to a numeric type. It also maps all values of this attribute to numeric values. You can specify exactly which component of date or time should be extracted. You can also specify relative to which date or time component information should be extracted.

## Description

The Date to Numerical operator provides a lot of flexibility when it comes to selecting a component of date or time. The following components can be selected: millisecond, second, minute, hour, day, week, month, quarter, half year, and year. The most important thing is that these components can be selected relative to other components. For example it is possible to extract the day relative to the week, relative to the month or relative to the year. Suppose the date is 15/Feb/2012. Then the day relative to the month would be 15 because it is the 15th day of the month. And the day relative to the year would be 46 because this is the 46th day of the year. All date and time components can be extracted relative to the most common parent components e.g. month can be calculated relative to the quarter or the year. Similarly second can be calculated relative to the minute, the hour or the day. All date and time components can be extracted relative to the Epoch where Epoch is defined as the date: '01-01-1970 00:00'. If the date attribute has no time information then all calculations on time components will result to 0. All these things can be understood easily by studying the attached Example Process.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Generate Data operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in the meta data. The Generate Data operator provides meta data along-with the data. The ExampleSet should have at least one date attribute because if there is no such attribute, the use of this operator does not make sense.

## Output Ports

**example set *(exa)*** The ExampleSet with selected date attribute converted to numeric type is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute name *(string)*** This parameter specifies the attribute of the input ExampleSet that should be converted from date to numerical form.

**time unit *(selection)*** This parameter specifies the unit in which the time is measured. In other words, this parameter specifies the component of the date that should be extracted. The

following components can be extracted: millisecond, second, minute, hour, day, week, month, quarter, half year, and year.

**millisecond relative to *(selection)*** This parameter is only available when the *time unit* parameter is set to 'millisecond'. This parameter specifies the component relative to which the milliseconds should be extracted. The following options are available: second, epoch.

**second relative to *(selection)*** This parameter is only available when the *time unit* parameter is set to 'second'. This parameter specifies the component relative to which the seconds should be extracted. The following options are available: minute, hour, day, epoch.

**minute relative to *(selection)*** This parameter is only available when the *time unit* parameter is set to 'minute'. This parameter specifies the component relative to which the minutes should be extracted. The following options are available: hour, day, epoch.

**hour relative to *(selection)*** This parameter is only available when the *time unit* parameter is set to 'hour'. This parameter specifies the component relative to which the hours should be extracted. The following options are available: day, epoch.

**day relative to *(selection)*** This parameter is only available when the *time unit* parameter is set to 'day'. This parameter specifies the component relative to which the days should be extracted. The following options are available: week, month, year, epoch.

**week relative to *(selection)*** This parameter is only available when the *time unit* parameter is set to 'week'. This parameter specifies the component relative to which the weeks should be extracted. The following options are available: month, year, epoch.

**month relative to *(selection)*** This parameter is only available when the *time unit* parameter is set to 'month'. This parameter specifies the component relative to which the months should be extracted. The following options are available: quarter, year, epoch.

**quarter relative to *(selection)*** This parameter is only available when the *time unit* parameter is set to 'quarter'. This parameter specifies the component relative to which the quarters should be extracted. The following options are available: year, epoch.

**half year relative to *(selection)*** This parameter is only available when the *time unit* parameter is set to 'half year'. This parameter specifies the component relative to which the half years should be extracted. The following options are available: year, epoch.

**year relative to *(selection)*** This parameter is only available when the *time unit* parameter is set to 'year'. This parameter specifies the component relative to which the years should be extracted. The following options are available: epoch, era.

**keep old attribute *(selection)*** This is an expert parameter. This parameter indicates if the original date attribute of the input ExampleSet should be kept. This parameter is set to false by default thus the original date attribute is removed from the input ExampleSet.

## Tutorial Processes

### Introduction to the Date to Numerical operator

The Generate Data by User Specification operator is used in this Example Process to create a date type attribute. The attribute is named 'Date' and it is defined by the expression 'date_parse("04/21/2012")'. Thus an attribute named 'Date' is created with just a single example. The value of the date is 21/April/2012. Please note that no information about time is given. The

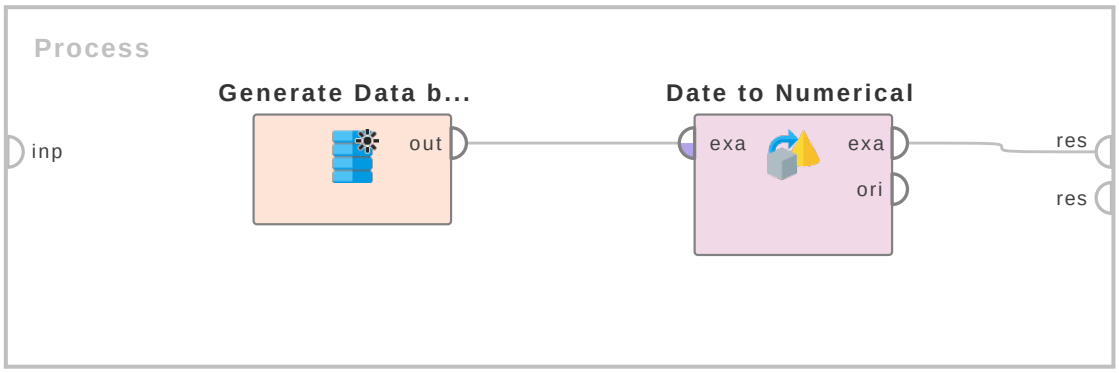
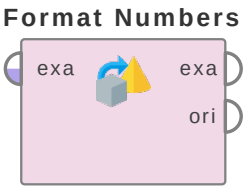



Figure 2.10: Tutorial process 'Introduction to the Date to Numerical operator'.

Date to Numerical operator is applied on this ExampleSet. The 'Date' attribute is selected in the attribute name parameter.

If the time unit parameter is set to 'year' and the year relative to parameter is set to 'era' then the result is 2012. This is so because this is the 2012th year relative to the era.If the time unit parameter is set to 'year' and the year relative to parameter is set to 'epoch' then the result is 42. This is so because the year of epoch date is 1970 and difference between 2012 and 1970 is 42.If the time unit parameter is set to 'half year' and the half year relative to parameter is set to 'year' then the result is 1. This is so because April is in the first half of the year.If the time unit parameter is set to 'quarter' and the quarter relative to parameter is set to 'year' then the result is 2. This is so because April is the 4th month of the year and it comes in the second quarter of the year.If the time unit parameter is set to 'month' and the month relative to parameter is set to 'year' then the result is 4. This is so because April is the fourth month of the year.If the time unit parameter is set to 'month' and the month relative to parameter is set to 'quarter' then the result is 1. This is so because April is the first month of the second quarter of the year.If the time unit parameter is set to 'week' and the week relative to parameter is set to 'year' then the result is 16. This is so because 21st April comes in the 16th week of the year.If the time unit parameter is set to 'week' and the week relative to parameter is set to 'month' then the result is 3. This is so because 21st day of month comes in the 3rd week of the month.If the time unit parameter is set to 'day' and the day relative to parameter is set to 'year' then the result is 112. This is so because 21st April is the 112th day of the year.If the time unit parameter is set to 'day' and the day relative to parameter is set to 'month' then the result is 21. This is so because 21st April is the 21st day of the month.If the time unit parameter is set to 'day' and the day relative to parameter is set to 'week' then the result is 7. This is so because 21st April 2012 is on Saturday. Saturday is the seventh day of the week. Sunday is the first day of the week.If the time unit parameter is set to 'hour' and the hour relative to parameter is set to 'day' then the result is 0. This is so because no time information was provided for this date attribute and all time information was assumed to be 00 by default.

# Format Numbers




This operator reformats the selected numerical attributes according to the specified format and changes the attributes to nominal.

## Description

This operator parses numerical values and formats them into the specified format. The format is specified by the *format type* parameter. It supports different kinds of number formats including integers (e.g. 123), fixed-point numbers (e.g. 123.4), scientific notation (e.g. 1.23E4), percentages (e.g. 12%), and currency amounts (e.g. $123). Please note that this operator only works on numerical attributes and the result will be in any case a nominal attribute even if the resulting format is a number which can be parsed again.

If the *format type* parameter is set to 'pattern', the *pattern* parameter is used for defining the format. If two different formats for positive and negative numbers should be used, those formats can be defined by separating them by a semi-colon ';'. The *pattern* parameter provides a lot of flexibility for defining the pattern. Important structures that can be used for defining a pattern are listed below. The structures in brackets are optional.

- pattern := subpattern{;subpattern}
- subpattern := {prefix}integer{.fraction}{suffix}
- prefix := any character combination including whitespace
- suffix := any character combination including whitespace
- integer := #* 0* 0
- fraction := 0* #*

0* and #* stand for multiple 0 or # respectively. 0 and # perform similar functions but 0 ensures that length of all numbers is same i.e. if a digit is missing it is replaced by 0. For example 54 will be formatted to 0054 with pattern '0000' and it will be formatted to 54 with pattern '####'.

The following placeholders can be used within the *pattern* parameter:

- . placeholder for decimal separator.
- , placeholder for grouping separator.
- E separates mantissa and exponent for exponential formats.
- - default negative prefix.
- % multiply by 100 and show as percentage.
- ' used to quote special characters in a prefix or suffix.

The *locale* parameter is ignored when the *format type* parameter is set to 'pattern'. In other cases it plays its role e.g. if the *format type* parameter is set to 'currency' then the *locale* parameter specifies the notation for that currency (i.e. dollar, euro etc).

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Generate Data operator in the attached Example Process. The output of other operators can also be used as input. The ExampleSet should have at least one numerical attribute because if there is no such attribute, the use of this operator does not make sense.

## Output Ports

**example set output *(exa)*** The selected numerical attributes are reformatted and converted to nominal and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression** ***(string)*** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6 \,\&\& < 11$' or '$<= 5 \,||\, < 0$'. But && and || cannot be used

together in one numeric condition. Conditions like '(> 0 && < 2) || (>10 && < 12)' are not allowed because they use both && and ||. Use a blank space after '>', '=' and '<' e.g. '<5' will not work, so use '< 5' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**format type** ***(selection)*** This parameter specifies the type of formatting to perform on the selected numerical attributes.

**pattern** ***(string)*** This parameter is only available when the *format type* parameter is set to 'pattern'. This parameter specifies the pattern for formatting the numbers. Various structures and replacement patterns for this parameter have been discussed in the description of this operator.

**locale** ***(selection)*** This is an expert parameter. A long list of locales is provided; users can select any of them.

**use grouping** ***(boolean)*** This parameter indicates if a grouping character should be used for larger numbers.

## Tutorial Processes

### Changing numeric values to currency format

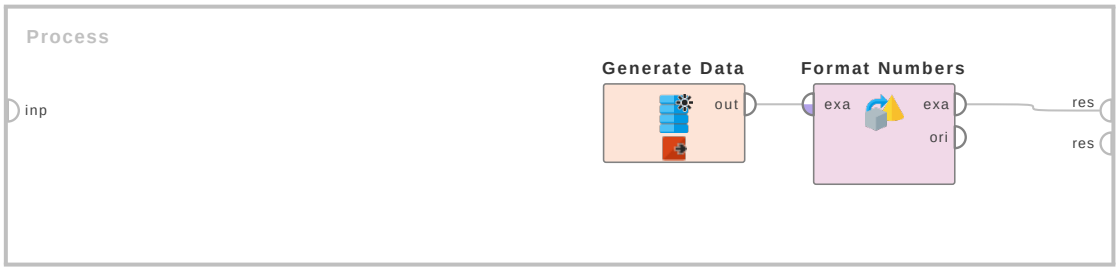
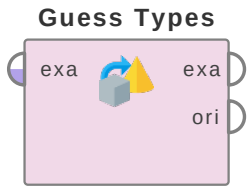



Figure 2.11: Tutorial process 'Changing numeric values to currency format'.

This process starts with the Generate Data operator which generates a random ExampleSet with a numeric attribute named 'att1'. A breakpoint is inserted here so that you can have a look at the ExampleSet. The Format Numbers operator is applied on it to change the format of this attribute to a currency format. The attribute filter type parameter is set to 'single' and the attribute parameter is set to 'att1' to select the required attribute. The format type parameter is set to 'currency'. Run the process and switch to the Results Workspace. You can see that the 'att1' attribute has been changed from numeric to nominal type and its values have a '$' sign in the beginning because they have been converted to a currency format. The locale parameter specifies

the required currency. In this process the locale parameter was set to ‘English (United States)’ therefore the numeric values were converted to the currency of United States (i.e. dollar).

# Guess Types




This operator (re-)guesses the value types of all attributes of the input ExampleSet and changes them accordingly.

## Description

The Guess Types operator can be used to (re-)guess the value types of the attributes of the input ExampleSet. This might be useful after some preprocessing transformations and purification of some of the attributes. This operator can be useful especially if nominal attributes can be handled as numerical attributes after some preprocessing. It is not necessary to (re-)guess the type of all the attributes with this operator. You can select the attributes whose type is to be (re-)guessed. Please study the attached Example Process for more information. Please note that this operator has no impact on the values of the ExampleSet.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set output *(exa)*** The type of the selected attributes of the input ExampleSet is (re-)guessed and the resultant ExampleSet is delivered through this output port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of *parameter* attribute if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list. Attributes can be shifted to the right list, which is the list of selected attributes.

**regular expression** ***(string)*** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression** ***(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter *(except regular expression*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will not be selected even if they match the previously mentioned type i.e. *value type* parameter's value.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be not be selected even if they match the previously mentioned block type i.e. *block type* parameter's value.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '>', '=' and '<' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**first character index** ***(integer)*** This parameter specifies the index of the first character of the substring which should be kept. Please note that the counting starts with 1.

**last character index** ***(integer)*** This parameter specifies the index of the last character of the substring which should be kept. Please note that the counting starts with 1.

**decimal point character** ***(char)*** The character specified by this parameter is used as the decimal character.

**number grouping character** ***(char)*** The character specified by this parameter is used as the grouping character. This character is used for grouping the numbers. If this character is found between numbers, the numbers are combined and this character is ignored. For example if "22-14" is present in the ExampleSet and "-" is set as the *number grouping character*, then the number will be considered to be "2214".

## Tutorial Processes

### Guessing the type of an attribute after preprocessing

The 'Iris' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. Please note the 'id' attribute. The 'id' attribute is of nominal type and it has the values of the format 'id_1', 'id_2' and so on. The Cut operator is applied on the ExampleSet to remove the substring 'id_' from the start of the 'id' attribute values. A breakpoint is inserted after the Cut operator. You can see that now the values in the 'id' attribute are of the form '1', '2', '3' and so on but the type of this attribute is still nominal. The Guess Types operator

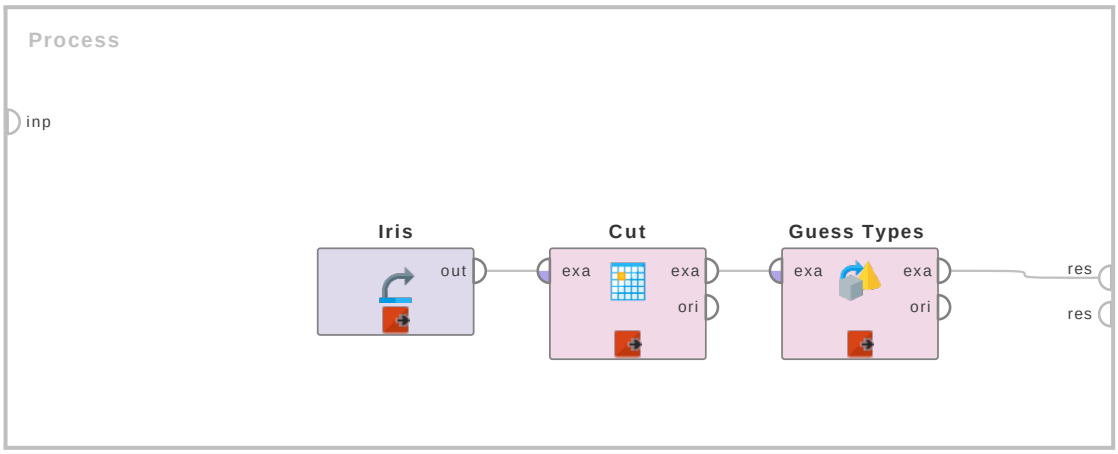
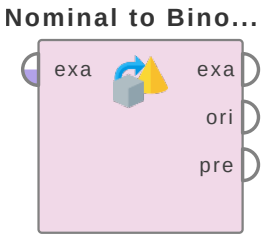



Figure 2.12: Tutorial process ‘Guessing the type of an attribute after preprocessing’.

is applied on this ExampleSet. The attribute filter type parameter is set to ‘single’, the attribute parameter is set to ‘id’ and the include special attributes parameter is also set to ‘true’. Thus the Guess Types operator will re-guess the type of the ‘id’ attribute. A breakpoint is inserted after the Guess Type operator. You can see that the type of the ‘id’ attribute has now been changed to integer.

# Nominal to Binominal




This operator changes the type of selected nominal attributes to a binominal type. It also maps all values of these attributes to binominal values.

## Description

The Nominal to Binominal operator is used for changing the type of nominal attributes to a binominal type. This operator not only changes the type of selected attributes but it also maps all values of these attributes to binominal values i.e. true and false. For example, if a nominal attribute with name 'costs' and possible nominal values 'low', 'moderate', and 'high' is transformed, the result is a set of three binominal attributes 'costs = low', 'costs = moderate', and 'costs = high'. Only the value of one of these attributes is true for a specific example, the value of the other attributes is false. Examples of the original ExampleSet where the 'costs' attribute had value 'low', in the new ExampleSet these examples will have attribute 'costs=low' value set to 'true', value of 'cost=moderate' and ' cost=high' attributes will be 'false'. Numeric attributes of the input ExampleSet remain unchanged.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in its meta data. The Retrieve operator provides meta data alongwith data. The ExampleSet should have at least one nominal attribute because if there is no such attribute, use of this operator does not make sense.

## Output Ports

**example set *(exa)*** The ExampleSet with selected nominal attributes converted to binominal type is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model *(pre)*** This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

## Parameters

**create view *(boolean)*** It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes that you want to convert to binominal form. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (*block type, use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are not selected.
- **numeric_value_filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute *(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of *parameter* attribute if the meta data is known.

**attributes *(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list. Attributes can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to binominal will take place; all other attributes will remain unchanged.

**regular expression *(string)*** The attributes whose name match this expression will be selected. Regular expression is very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (*except regular expression*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type)* becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Nominal to Binominal operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Nominal to Binominal operator and only those attributes are selected that satisfy the conditions.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is removed prior to selection of this parameter. After selection of this parameter 'att1' will be removed and 'att2' will be selected.

**transform binominal** ***(boolean)*** This parameter indicates if attributes which are already binominal should be dichotomized i.e. they should be split in two columns with values true and false.

**use underscore in name *(boolean)*** This parameter indicates if underscores should be used in the new attribute names instead of empty spaces and '='. Although the resulting names are harder to read for humans it might be more appropriate to use these if the data should be written into a database system.

## Tutorial Processes

### Nominal to Binominal conversion of attributes of Golf data set

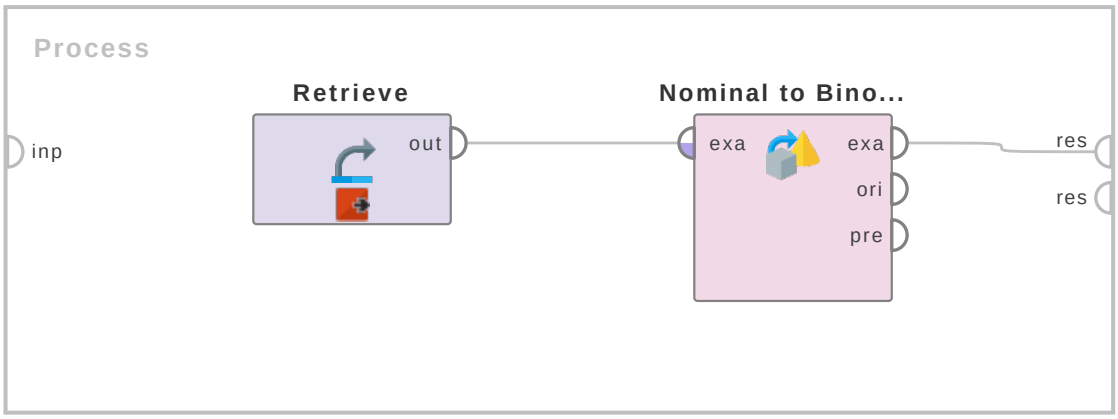


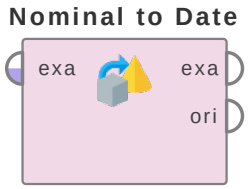

Figure 2.13: Tutorial process 'Nominal to Binominal conversion of attributes of Golf data set'.

This Example Process mostly focuses on the transform binominal parameter. All remaining parameters are mostly for selecting the attributes. The Select Attributes operator also has many similar parameters for selection of attributes. You can study the Example Process of the Select Attributes operator if you want an understanding of these parameters.

The Retrieve operator is used to load the Golf data set. A breakpoint is inserted at this point so that you can have look at the data set before application of the Nominal to Binominal operator. You can see that the 'Outlook' attribute has three possible values i.e. 'sunny', 'rain' and 'overcast'. The 'Wind' attribute has two possible values i.e. 'true' and 'false'. All parameters of the Nominal to Binominal operator are used with default values. Run the process. First you will see the Golf data set. Press the run button again and you will see the final results. You can see that the 'Outlook' attribute is replaced by three binominal attributes, one for each possible value of the original 'Outlook' attribute. These attributes are ' Outlook = sunny', ' Outlook = rain', and ' Outlook = overcast'. Only the value of one of these attributes is true for a specific example, the value of the other attributes is false. Examples whose 'Outlook ' attribute had the value 'sunny' in the original ExampleSet, will have the attribute ' Outlook =sunny' value set to 'true'in the new ExampleSet, the value of the 'Outlook =overcast' and 'Outlook =rain' attributes will be 'false'. The numeric attributes of the input ExampleSet remain unchanged.

The 'Wind' attribute was not replaced by two binominal attributes, one for each possible value of the 'Wind' attribute because this attribute is already binominal. Still if you want to break it into two separate binominal attributes, this can be done by setting the transform binominal parameter to true.

# Nominal to Date




This operator converts the selected nominal attribute into the selected date time type. The nominal values are transformed into date and/or time values. This conversion is done with respect to the specified date format string.

## Description

The Nominal to Date operator converts the selected nominal attribute of the input ExampleSet into the selected date and/or time type. The attribute is selected by the *attribute name* parameter. The type of the resultant date and/or time attribute is specified by the *date type* parameter. The nominal values are transformed into date and/or time values. This conversion is done with respect to the specified date format string that is specified by the *date format* parameter. The old nominal attribute will be removed and replaced by a new date and/or time attribute if the *keep old attribute* parameter is not set to true.

### Date and Time Patterns

This section explains the date and time patterns. Understanding of date and time patterns is necessary specially for specifying the *date format* string in the date format parameter. Within date and time pattern strings, unquoted letters from 'A' to 'Z' and from 'a' to 'z' are interpreted as pattern letters that represent the components of a date or time. Text can be quoted using single quotes (') to avoid interpretation as date or time components. All other characters are not interpreted as date or time components; they are simply matched against the input string during parsing.

Here is a brief description of the defined pattern letters. The *format types* like 'Text', 'Number', 'Year', 'Month' etc are described in detail after this section.

- G: This pattern letter is the era designator. For example: AD, BC etc. This pattern letter follows the rules of 'Text' format type.
- y: This pattern letter represents year. yy represents year in two digits e.g. 96 and yyyy represents year in four digits e.g. 1996. This pattern letter follows the rules of the 'Year' format type.
- M: This pattern letter represents the month of the year. This pattern letter follows the rules of the 'Month' format type. Month can be represented as; for example; March, Mar or 03 etc.
- w: This pattern letter represents the week number of the year. This pattern letter follows the rules of the 'Number' format type. For example, the first week of January can be represented as 01 and the last week of December can be represented as 52.
- W: This pattern letter represents the week number of the month. This pattern letter follows the rules of the 'Number' format type. For example, the first week of January can be represented as 01 and the forth week of December can be represented as 04.
- D: This pattern letter represents the day number of the year. This pattern letter follows the rules of the 'Number' format type. For example, the first day of January can be represented as 01 and last day of December can be represented as 365 (or 366 in case of a leap year).

- d: This pattern letter represents the day number of the month. This pattern letter follows the rules of the 'Number' format type. For example, the first day of January can be represented as 01 and the last day of December can be represented as 31.
- F: This pattern letter represents the day number of the week. This pattern letter follows the rules of the 'Number' format type.
- E: This pattern letter represents the name of the day of the week. This pattern letter follows the rules of the 'Text' format type. For example, Tuesday or Tue etc.
- a: This pattern letter represents the AM/PM portion of the 12-hour clock. This pattern letter follows the rules of the 'Text' format type.
- H: This pattern letter represents the hour of the day (from 0 to 23). This pattern letter follows the rules of the 'Number' format type.
- k: This pattern letter represents the hour of the day (from 1 to 24). This pattern letter follows the rules of the 'Number' format type.
- K: This pattern letter represents the hour of the day for 12-hour clock (from 0 to 11). This pattern letter follows the rules of the 'Number' format type.
- h: This pattern letter represents the hour of the day for 12-hour clock (from 1 to 12). This pattern letter follows the rules of the 'Number' format type.
- m: This pattern letter represents the minutes of the hour (from 0 to 59). This pattern letter follows the rules of the 'Number' format type.
- s: This pattern letter represents the seconds of the minute (from 0 to 59). This pattern letter follows the rules of the 'Number' format type.
- S: This pattern letter represents the milliseconds of the second (from 0 to 999). This pattern letter follows the rules of the 'Number' format type.
- z: This pattern letter represents the time zone. This pattern letter follows the rules of the 'General Time Zone' format type. Examples include Pacific Standard Time, PST, GMT-08:00 etc.
- Z: This pattern letter represents the time zone. This pattern letter follows the rules of the 'RFC 822 Time Zone' format type. Examples include -08:00 etc.

Please note that all other characters from 'A' to 'Z' and from 'a' to 'z' are reserved. Pattern letters are usually repeated, as their number determines the exact presentation. Here is the explanation of various *format types*:

- Text: For formatting, if the number of pattern letters is 4 or more, the full form is used; otherwise a short or abbreviated form is used (if available). For parsing, both forms are acceptable independent of the number of pattern letters.
- Number: For formatting, the number of pattern letters is the minimum number of digits. The numbers that are shorter than this minimum number of digits are zero-padded to this amount. For example if the minimum number of digits is 3 then the number 5 will be changed to 005. For parsing, the number of pattern letters is ignored unless it is needed to separate two adjacent fields.
- Year: If the underlying calendar is the Gregorian calendar, the following rules are applied:

  - For formatting, if the number of pattern letters is 2, the year is truncated to 2 digits; otherwise it is interpreted as a ‘Number’ format type.
  - For parsing, if the number of pattern letters is more than 2, the year is interpreted literally, regardless of the number of digits. So using the pattern ‘MM/dd/yyyy’, the string ‘01/11/12’ parses to ‘Jan 11, 12 A.D’.
  - For parsing with the abbreviated year pattern (’y’ or ‘yy’), this operator must interpret the abbreviated year relative to some century. It does this by adjusting dates to be within 80 years before and 20 years after the time the operator is created. For example, using a pattern of ‘MM/dd/yy’ and the operator created on Jan 1, 1997, the string ‘01/11/12’ would be interpreted as Jan 11, 2012 while the string ‘05/04/64’ would be interpreted as May 4, 1964. During parsing, only strings consisting of exactly two digits will be parsed into the default century. Any other numeric string, such as a one digit string, a three or more digit string, or a two digit string that is not all digits (for example, ‘-1’), is interpreted literally. So ‘01/02/3’ or ‘01/02/003’ are parsed, using the same pattern, as ‘Jan 2, 3 AD’. Likewise, ‘01/02/-3’ is parsed as ‘Jan 2, 4 BC’.

  Otherwise, if the underlying calendar is not the Gregorian calendar, calendar system specific forms are applied. If the number of pattern letters is 4 or more, a calendar specific long form is used. Otherwise, a calendar short or abbreviated form is used.

- Month: If the number of pattern letters is 3 or more, the month is interpreted as ‘Text’ format type otherwise, it is interpreted as a ‘Number’ format type.

- General time zone: Time zones are interpreted as ‘Text’ format type if they have names. It is possible to define time zones by representing a GMT offset value. RFC 822 time zones are also acceptable.

- RFC 822 time zone: For formatting, the RFC 822 4-digit time zone format is used. General time zones are also acceptable.

This operator also supports localized date and time pattern strings by defining the *locale* parameter. In these strings, the pattern letters described above may be replaced with other, locale-dependent pattern letters.

The following examples show how date and time patterns are interpreted in the U.S. locale. The given date and time are 2001-07-04 12:08:56 local time in the U.S. Pacific Time time zone.

- ’yyyy.MM.dd G ‘at’ HH:mm:ss z’: 2001.07.04 AD at 12:08:56 PDT
- ’EEE, MMM d, yy’: Wed, Jul 4, ‘01
- ’h:mm a’: 12:08 PM
- ’hh ‘oclock’ a, zzzz’: 12 oclock PM, Pacific Daylight Time
- ’K:mm a, z’: 0:08 PM, PDT
- ’yyyy.MMMMM.dd GGG hh:mm aaa’: 2001.July.04 AD 12:08 PM
- ’EEE, d MMM yyyy HH:mm:ss Z’: Wed, 4 Jul 2001 12:08:56 -0700
- ’yyMMddHHmmssZ’: 010704120856-0700
- ’yyyy-MM-dd’T’HH:mm:ss.SSSZ’: 2001-07-04T12:08:56.235-0700

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The ExampleSet should have at least one nominal attribute because if there is no such attribute, the use of this operator does not make sense.

## Output Ports

**example set *(exa)*** The selected nominal attribute is converted to date type and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute name *(string)*** The name of the nominal attribute that is to be converted to date type is specified here.

**date type *(selection)*** This parameter specifies the type of the resultant attribute.

- **date** If the *date type* parameter is set to 'date', the resultant attribute will be of date type. The time portion (if any) of the nominal attribute will be ignored.
- **time** If the *date type* parameter is set to 'time', the resultant attribute will be of time type. The date portion (if any) of the nominal attribute will be ignored.
- **date_time** If the *date type* parameter is set to 'date_time', the resultant attribute will be of date_time type.

**date format** This is the most important parameter of this operator. It specifies the date time format of the selected nominal attribute. Date format strings are discussed in detail in the description of this operator.

**time zone *(selection)*** This is an expert parameter. A long list of time zones is provided; users can select any of them.

**locale *(selection)*** This is an expert parameter. A long list of locales is provided; users can select any of them.

**keep old attribute *(boolean)*** This parameter indicates if the original nominal attribute should be kept or if it should be discarded.

## Tutorial Processes

### Introduction to the Nominal to Date operator

This Example Process starts with a subprocess. The subprocess delivers an ExampleSet with just a single attribute. The name of the attribute is 'deadline_date'. The type of the attribute is nominal. A breakpoint is inserted here so that you can view the ExampleSet. As you can see, all the examples of this attribute have both date and time information. The Nominal to Date operator

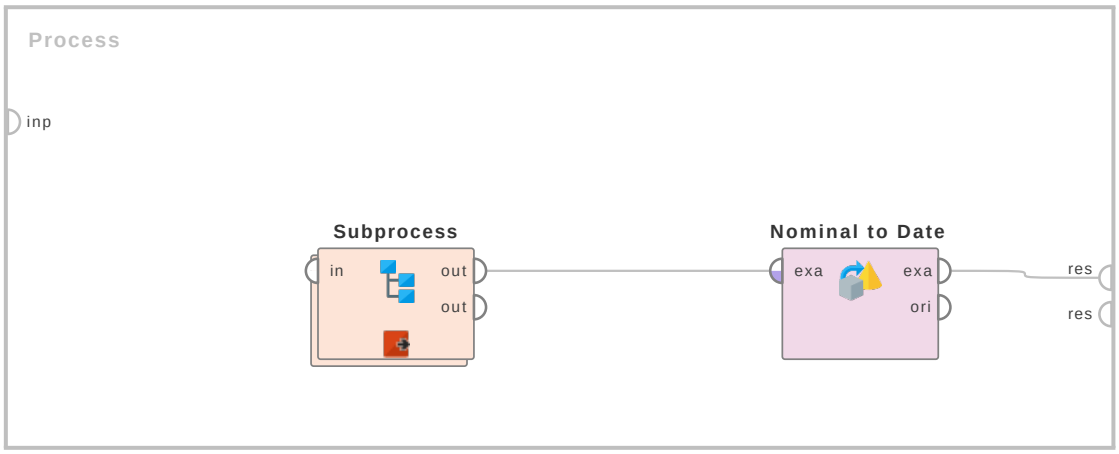
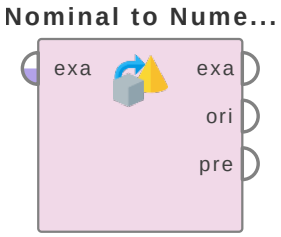



Figure 2.14: Tutorial process ‘Introduction to the Nominal to Date operator’.

is applied on this ExampleSet to change the type of the ‘deadline_date’ attribute from nominal to date type. Have a look at the parameters of the Nominal to Date operator. The attribute name parameter is set to ‘deadline_date’. The date type parameter is set to ‘date’. Thus the ‘deadline_date’ attribute will be converted from nominal to date type (not date_time) therefore the time portion of the value will not be available in the resultant attribute. The date format parameter is set to ‘EEEE, MMMM d, yyyy h:m:s a z’, here is an explanation of this date format string: ’E’ is the pattern letter used for the representation of the name of the day of the week. As explained in the description, if the number of pattern letters is 4 or more, the full form is used. Thus ‘EEEE’ is used for representing the day of the week in full form e.g. Monday, Tuesday etc.’M’ is the pattern letter used for the representation of the name of the month of the year. As explained in the description, if the number of pattern letters is 4 or more, the full form is used. Thus ‘MMMM’ is used for representing the month of the year in full form e.g. January, February etc.’y’ is the pattern letter used for the representation of the year portion of the date. ‘yyyy’ represents year of date in four digits like 2011, 2012 etc.’h’ is the pattern letter used for the representation of the hour portion of the time. ‘h’ can represent multiple digit hours as well e.g. 10, 11 etc. The difference between ‘hh’ and ‘h’ is that ‘hh’ represents single digit hours by appending a 0 in start e.g. 01, 02 and so on. But ‘h’ represents single digits without any modifications e.g. 1, 2 and so on.’m’ is the pattern letter used for the representation of the minute portion of the time. ‘m’ can represent multiple digit minutes as well e.g. 51, 52 etc. The difference between ‘mm’ and ‘m’ is that ‘mm’ represents single digit minutes by appending a 0 in start e.g. 01, 02 and so on. But ‘m’ represents single digits without any modifications e.g. 1, 2 and so on.’s’ is the pattern letter used for the representation of the second portion of the time. ‘s’ can represent multiple digit seconds as well e.g. 40, 41 etc. The difference between ‘ss’ and ‘s’ is that ‘ss’ represents single digit seconds by appending a 0 in start e.g. 01, 02 and so on. But ‘s’ represents single digits without any modifications e.g. 1, 2 and so on.’a’ is the pattern letter used for the representation of the ‘AM/PM’ portion of the 12-hour date and time.’z’ is the pattern letter used for the representation of the time zone.

Please note that this date format string represents the date format of the nominal values of the selected nominal attribute of the input ExampleSet. The date format string helps RapidMiner to understand which portions of the nominal value represent which component of the date or time e.g. year, month etc.

# Nominal to Numerical




This operator changes the type of selected non-numeric attributes to a numeric type. It also maps all values of these attributes to numeric values.

## Description

The Nominal to Numerical operator is used for changing the type of non-numeric attributes to a numeric type. This operator not only changes the type of selected attributes but it also maps all values of these attributes to numeric values. *Binary* attribute values are mapped to 0 and 1. Numeric attributes of input the ExampleSet remain unchanged. This operator provides three modes for conversion from nominal to numeric. This mode is selected by the *coding type* parameter. Explanation of these coding types is given in the parameters and they are also explained in the example process.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with data for input because attributes are specified in its meta data. The Retrieve operator provides meta data along-with data. The ExampleSet should have at least one non-numeric attribute because if there is no such attribute, the use of this operator does not make sense.

## Output Ports

**example set *(exa)*** The ExampleSet with selected non-numeric attributes converted to numeric types is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model *(pre)*** This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

## Parameters

**create view *(boolean)*** It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes on which you want to apply nominal to numeric conversion. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose all examples satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute *(string)*** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes *(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression *(string)*** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression *(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except regular expression *(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type *(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception *(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type *(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type *(selection)*** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception *(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type *(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition *(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes *(boolean)*** The special attributes are attributes with special roles. The special attributes are those attributes which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection *(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**coding type *(selection)*** This parameter indicates the coding which will be used for transforming nominal attributes to numerical attributes. There are three available options i.e. unique integers, dummy coding, effect coding. You can easily understand these options by studying the attached Example Process.

- **unique_integers** If this option is selected, the values of nominal attributes can be seen as equally ranked, therefore the nominal attribute will simply be turned into a real valued attribute, the old values result in equidistant real values.
- **dummy_coding** If this option is selected, for all values of the nominal attribute, excluding the *comparison group*, a new attribute is created. The *comparison group* can be defined using the *comparison groups* parameter. In every example, the new attribute which corresponds to the actual nominal value of that example gets value 1 and all

other new attributes get value 0. If the value of the nominal attribute of this example corresponds to the *comparison group*, all new attributes are set to 0. Note that the *comparison group* is an optional parameter with 'dummy coding'. If no *comparison group* is defined, in every example the new attribute which corresponds to the actual nominal value of that example gets value 1 and all other new attributes get value 0. In this case, there will be no example where all new attributes get value 0.This can be easily understood by studying the attached example process.

- **effect_coding** If this option is selected; for all values of the nominal attribute, excluding the *comparison group*, a new attribute is created. The *comparison group* can be defined using the *comparison groups* parameter. In every example, the new attribute which corresponds to the actual nominal value of that example gets value 1 and all other new attributes get value 0. If the value of the nominal attribute of this example corresponds to the *comparison group*, all new attributes are set to -1. Note that the *comparison group* is a mandatory parameter with 'effect coding'. This can be easily understood by studying the attached example process.

**use comparison groups *(boolean)*** This parameter is available only when the *coding type* parameter is set to *dummy coding*. If checked, for each selected attribute in the ExampleSet a value has to be specified in the *comparison group* parameter. A separate new column for this value will not appear in the final result set. If not checked, all values of the selected attributes will result in an indicator attribute in the resultant ExampleSet.

**comparison groups** This parameter defines the *comparison group* for each selected non-numeric attribute. Only one *comparison group* can be specified for one attribute. When the *coding type* parameter is set to 'effect coding', it is compulsory to define a *comparison group* for all selected attributes.

**use underscore in name *(boolean)*** This parameter indicates if underscores should be used in the names of new attributes instead of empty spaces and '='. Although the resulting names are harder to read for humans but it might be more appropriate to use these if the data is to be written into a database system.

## Tutorial Processes

### Nominal to Numeric conversion through different coding types

This Example Process mostly focuses on the coding type and comparison groups parameters. All remaining parameters are mostly for selecting the attributes. The Select Attributes operator also has many similar parameters for the selection of attributes. You can study its Example Process if you want an understanding of these parameters.

The Retrieve operator is used to load the 'Golf 'data set. The Nominal to Numerical operator is applied on it. The 'Outlook' and 'Wind' attributes are selected for this operator for changing them to numeric attributes. Initially, the coding type parameter is set to 'unique integers'. Thus, the nominal attributes will simply be turned into real valued attributes; the old values will result in equidistant real values. As you can see in the Results Workspace, all occurrences of value 'sunny' for the 'Outlook' attribute are replaced by 2. Similarly, 'overcast' and 'rain' are replaced by 1 and 0 respectively. In the same way, all occurrences of 'false' value in the 'Wind' attribute are replaced by 1 and occurrences of 'true' are replaced by 0.

Now, change the coding type parameter to 'dummy coding' and run the process again. As dummy coding is selected, for all values of the nominal attribute a new attribute is created. In every example, the new attribute which corresponds to the actual nominal value of that example gets value 1 and all other new attributes get value 0. As you can see in the Results Workspace,

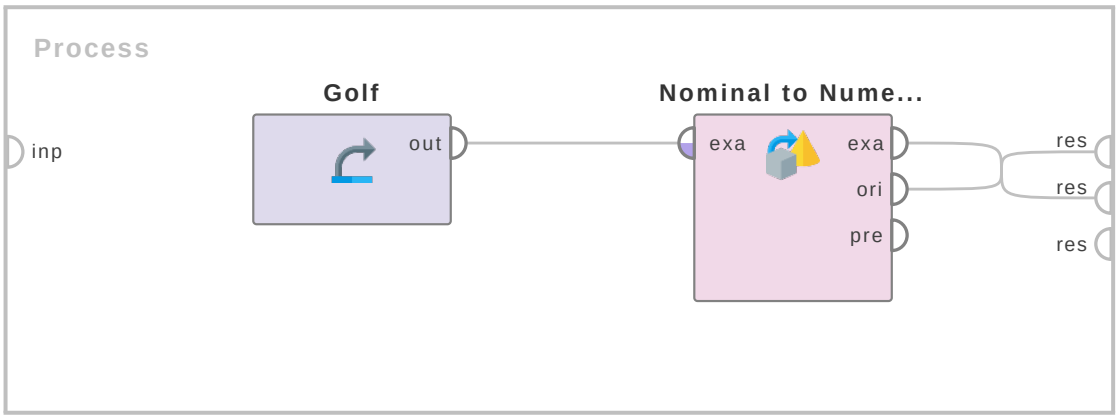


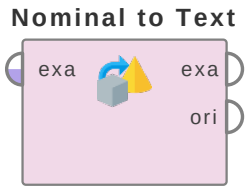

Figure 2.15: Tutorial process 'Nominal to Numeric conversion through different coding types'.

'Wind=true' and 'Wind=false' attributes are created in place of the 'Wind' attribute. In all examples where the 'Wind' attribute had value 'true', the 'Wind=true' attributes gets 1 and 'Wind=false' attribute gets 0. Similarly, all examples where the 'Wind' attribute had value 'false', the 'Wind=true' attribute gets value 0 and 'Wind= false' attribute gets value 1. The same principle applies to the 'Outlook' attribute.

Now, keep the coding type parameter as 'dummy coding' and also set the use comparison groups parameter to true. Run the process again. You can see in the comparison groups parameter that 'sunny' and 'true' are defined as comparison groups for the 'Outlook' and 'Wind' attributes respectively. As dummy coding is used and the comparison groups are also used thus for all values of the nominal attribute, excluding the comparison group, a new attribute is created. In every example, the new attribute which corresponds to the actual nominal value of that example gets value 1 and all other new attributes get value 0. If the value of the nominal attribute of this example corresponds to the comparison group, all new attributes are set to 0. This is why 'Outlook=rain' and 'Outlook=overcast' attributes are created but 'Outlook=sunny' attribute is not created this time. In examples where the 'Outlook' attribute had value 'sunny', all new Outlook attributes get value 0. You can see this in the Results Workspace. The same rule is applied on the 'Wind' attribute.

Now, change the coding type parameter to 'effect coding' and run the process again. You can see in the comparison groups parameter that 'sunny' and 'true' are defined as comparison groups for the 'Outlook' and 'Wind' attributes respectively. As effect coding is selected thus for all values of the nominal attribute, excluding the comparison group, a new attribute is created. In every example, the new attribute which corresponds to the actual nominal value of that example gets value 1 and all other new attributes get value 0. If the value of the nominal attribute of this example corresponds to the comparison group, all new attributes are set to -1. This is why 'Outlook=rain' and 'Outlook = overcast' attributes are created but an 'Outlook=sunny' attribute is not created this time. In examples where the 'Outlook' attribute had value 'sunny', all new Outlook attributes get value -1. You can see this in the Results Workspace. The same rule is applied on the 'Wind' attribute.

# Nominal to Text




This operator changes the type of selected nominal attributes to text. It also maps all values of these attributes to corresponding string values.

## Description

The Nominal to Text operator converts all nominal attributes to string attributes. Each nominal value is simply used as a string value of the new attribute. If the value is missing in the nominal attribute, the new value will also be missing.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The ExampleSet should have at least one nominal attribute because if there is no such attribute, the use of this operator does not make sense.

## Output Ports

**example set *(exa)*** The ExampleSet with selected nominal attributes converted to text is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes on which you want to apply nominal to text conversion. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression** ***(string)*** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression** ***(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter *(except regular expression*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will not be selected even if they match the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be not be selected even if they match the previously mentioned block type i.e. *block type* parameter's value.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '> 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '> 6 && < 11' or '<= 5 || < 0'. But && and || cannot be used together in one numeric condition. Conditions like '(> 0 && < 2) || (>10 && < 12)' are not allowed because they use both && and ||. Use a blank space after '>', '=' and '<' e.g. '<5' will not work, so use '< 5' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

## Tutorial Processes

### Applying the Nominal to Text operator on the Golf data set

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted after the Retrieve operator so that you can have a look at the 'Golf' data set before application of the Nominal to Text operator. You can see that the 'Golf' data set has three nominal attributes i.e. 'Play', 'Outlook' and 'Wind'. The Nominal to Text operator is applied on this data set. The attribute filter type parameter is set to 'single' and the attribute parameter is set to 'Outlook'. Thus this operator converts the type of the 'Outlook' attribute to text. You can verify this by seeing the results in the Meta Data View in the Results Workspace.

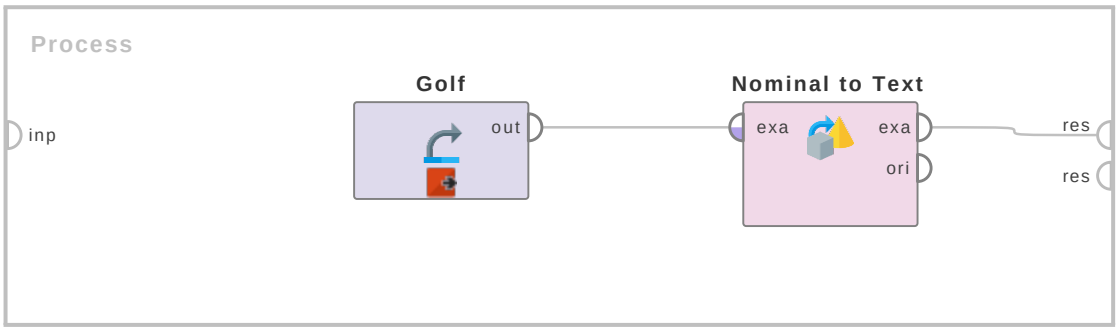
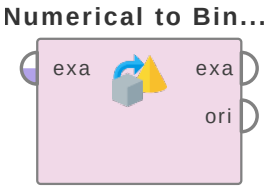



Figure 2.16: Tutorial process 'Applying the Nominal to Text operator on the Golf data set'.

## Numerical to Binominal




This operator changes the type of the selected numeric attributes to a binominal type. It also maps all values of these attributes to corresponding binominal values.

### Description

The Numerical to Binominal operator changes the type of numeric attributes to a binominal type (also called binary). This operator not only changes the type of selected attributes but it also maps all values of these attributes to corresponding binominal values. Binominal attributes can have only two possible values i.e. 'true' or 'false'. If the value of an attribute is between the specified minimal and maximal value, it becomes 'false', otherwise 'true'. Minimal and maximal values can be specified by the *min* and *max* parameters respectively. If the value is missing, the new value will be missing. The default boundaries are both set to 0.0, thus only 0.0 is mapped to 'false' and all other values are mapped to 'true' by default.

### Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data alongwith data. The ExampleSet should have at least one numeric attribute because if there is no such attribute, use of this operator does not make sense.

### Output Ports

**example set *(exa)*** The ExampleSet with selected numeric attributes converted to binominal type is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators

or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes that you want to convert to binominal form. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (*block type, use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are not selected.
- **numeric_value_filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute *(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of *parameter* attribute if the meta data is known.

**attributes *(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list. Attributes can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to binominal will take place; all other attributes will remain unchanged.

**regular expression** ***(string)*** The attributes whose name match this expression will be selected. Regular expression is very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (*except regular expression*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type)* becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Nominal to Binominal operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Nominal to Binominal operator and only those attributes are selected that satisfy the conditions.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is

removed prior to selection of this parameter. After selection of this parameter 'att1' will be removed and 'att2' will be selected.

**min** ***(real)*** This parameter is used to set the lower bound of the range. The *max* parameter is used to set the upper bound of the range. The attribute values that fell in this range are mapped to 'false'. The attribute values that do not fell in this range are mapped to 'true'.

**max** ***(real)*** This parameter is used to set the upper bound of the range. The *min* parameter is used to set the lower bound of the range. The attribute values that fell in this range are mapped to 'false'. The attribute values that do not fell in this range are mapped to 'true'.

## Tutorial Processes

### Converting numeric attributes of the Sonar data set to binominal attributes

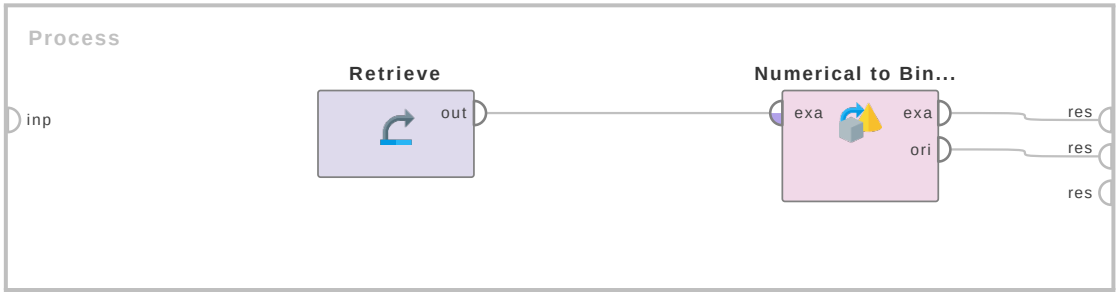
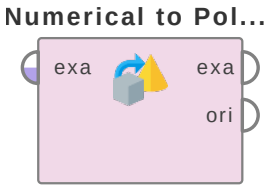



Figure 2.17: Tutorial process 'Converting numeric attributes of the Sonar data set to binominal attributes'.

This Example Process mostly focuses on the min and max parameters. All remaining parameters are mostly for selecting the attributes. The Select Attributes operator also has many similar parameters for selection of attributes. You can study the Example Process of the Select Attributes operator if you want an understanding of these parameters.

The 'Sonar' data set is loaded using the Retrieve operator. The Numerical to Binominal operator is applied on it. The min parameter is set to 0.0 and the max parameter is set to 0.01. All other parameters are used with default values. The attribute filter type parameter is set to 'all', thus all numeric attributes of the 'Sonar' data set will be converted to binominal type. As you can see in the Results Workspace, before application of the Numerical to Binominal operator, all attributes were of real type. After application of this operator they are now all changed to binominal type. All attribute values that fell in the range from 0.0 to 0.01 are mapped to 'false', all the other values are mapped to 'true'.

# Numerical to Polynominal




This operator changes the type of selected numeric attributes to a polynominal type. It also maps all values of these attributes to corresponding polynominal values. This operator simply changes the type of selected attributes; if you need a more sophisticated normalization method please use the discretization operators.

## Description

The Numerical to Polynominal operator is used for changing the type of numeric attributes to a polynominal type. This operator not only changes the type of selected attributes but it also maps all values of these attributes to corresponding polynominal values. It simply changes the type of selected attributes i.e. every new numerical value is considered to be another possible value for the polynominal attribute. In other words, each numerical value is simply used as nominal value of the new attribute. As numerical attributes can have a huge number of different values even in a small range, converting such a numerical attribute to polynominal form will generate a huge number of possible values for the new attribute. Such a polynominal attribute may not be a very useful one and it may increase memory usage significantly. If you need a more sophisticated normalization method please use the discretization operators. The Discretization operators are at: "Data Transformation/ Type Conversion/ Discretization".

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data alongwith data. The ExampleSet should have at least one numeric attribute because if there is no such attribute, use of this operator does not make sense.

## Output Ports

**example set *(exa)*** The ExampleSet with selected numeric attributes converted to nominal type is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes that you want to convert to polynominal form. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.

- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (*block type, use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are not selected.
- **numeric_value_filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of *parameter* attribute if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list. Attributes can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to polynominal will take place; all other attributes will remain unchanged.

**regular expression** ***(string)*** The attributes whose name match this expression will be selected. Regular expression is very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (*except regular expression*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type)* becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Nominal to Polynominal operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Nominal to Polynominal operator and only those attributes are selected that satisfy the conditions.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is removed prior to selection of this parameter. After selection of this parameter 'att1' will be removed and 'att2' will be selected.

## Tutorial Processes

### Converting numeric attributes of the Sonar data set to polynominal attributes

This Example Process mostly focuses on the working of this operator. All parameters of this operator are mostly for selecting the attributes. The Select Attributes operator also has many

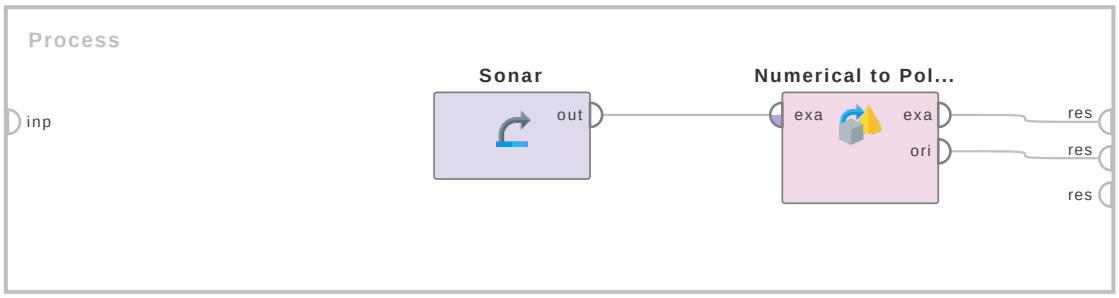
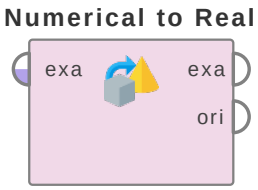



Figure 2.18: Tutorial process 'Converting numeric attributes of the Sonar data set to polynominal attributes'.

similar parameters for selection of attributes. You can study the Example Process of the Select Attributes operator if you want an understanding of these parameters.

The 'Sonar' data set is loaded using the Retrieve operator. The Numerical to Polynominal operator is applied on it. All parameters are used with default values. The attribute filter type parameter is set to 'all', thus all numeric attributes of the 'Sonar' data set will be converted to nominal type. As you can see in the Results Workspace, before application of the Numerical to Polynominal operator, all attributes were of real type. After application of this operator they are now all changed to nominal type. But if you have a look at the examples, they are exactly the same i.e. just the type of the values has been changed not the actual values. Every new numerical value is considered to be another possible value for the polynominal attribute. In other words, each numerical value is simply used as nominal value of the new attribute. As there is a very large number of different values for almost all attributes in the 'Sonar' data set, converting these attributes to polynominal form generates a huge number of possible values for the new attributes. These new polynominal attributes may not be very useful and they may increase memory usage significantly. In such a scenario it is always better to use a more sophisticated normalization method i.e. the discretization operators.

# Numerical to Real




This operator changes the type of the selected numerical attributes to real type. It also maps all values of these attributes to real values.

## Description

The Numerical to Real operator converts selected numerical attributes (especially the integer attributes) to real valued attributes. Each integer value is simply used as a real value of the new attribute. If the value is missing, the new value will be missing.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. The ExampleSet should have at least one non-real numerical attribute because if there is no such attribute, the use of this operator does not make sense.

## Output Ports

**example set output *(exa)*** The ExampleSet with selected numerical attributes converted to real type is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes on which you want to apply numerical to real conversion. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to *numeric* type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When it is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression** ***(string)*** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type *(selection)*** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception *(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type *(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition *(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '> 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '> 6 && < 11' or '<= 5 || < 0'. But && and || cannot be used together in one numeric condition. Conditions like '(> 0 && < 2) || (>10 && < 12)' are not allowed because they use both && and ||. Use a blank space after '>', '=' and '<' e.g. '<5' will not work, so use '< 5' instead.

**include special attributes *(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection *(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

## Tutorial Processes

### Integer to real conversion of attributes of the Golf data set

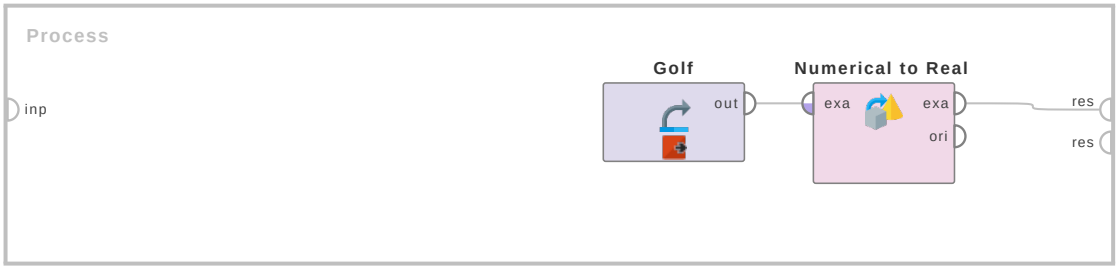
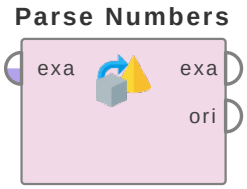



Figure 2.19: Tutorial process 'Integer to real conversion of attributes of the Golf data set'.

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the type of the Humidity and Temperature attributes is integer. The Numerical to Real operator is applied on the 'Golf' data set to convert the type of these integer attributes to real. All parameters are used with default values. The resultant ExampleSet can be seen in the Results Workspace. You can see that now the type of these attributes is real.

# Parse Numbers




This operator changes the type of selected nominal attributes to a numeric type. It also maps all values of these attributes to numeric values by parsing the numbers if possible.

## Description

The Parse Numbers operator is used for changing the type of nominal attributes to a numeric type. This operator not only changes the type of selected attributes but it also maps all values of these attributes to numeric values by parsing the numbers if possible. In contrast to the Nominal to Numerical operator, this operator directly parses numbers from the afore wrongly encoded as nominal values. The Nominal to Numeric operator is used when the values are actually nominal but you want to change them to numerical values. On the other hand the Parse Numbers operator is used when the values should actually be numerical but they are wrongly stored as nominal values. Please note that this operator will first check the stored nominal mappings for all attributes. If (old) mappings are still stored which actually are nominal (without the corresponding data being part of the ExampleSet), the attribute will not be converted. Please use the Guess Types operator in these cases.

## Differentiation

- **Nominal to Numerical** The Nominal to Numerical operator provides various coding types to convert nominal attributes to numerical attributes. On the other hand the Parse Numbers operator is used when the values should actually be numerical but they are wrongly stored as nominal values. See page 159 for details.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input. The ExampleSet should have at least one nominal attribute because if there is no such attribute, the use of this operator does not make sense.

## Output Ports

**example set output *(exa)*** The ExampleSet with selected nominal attributes converted to numeric types is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to *numeric* type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression** ***(string)*** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**decimal character** ***(char)*** This character is used as the decimal character.

**grouped digits** ***(boolean)*** This option decides whether grouped digits should be parsed or not. If this option is set to true, *grouping character* parameter should be specified.

**grouping character** ***(char)*** This character is used as the grouping character. If this character is found between numbers, the numbers are combined and this character is ignored. For example if "22-14" is present in the nominal attribute and "-" is set as *grouping character*, then "2214" will be stored in the corresponding numerical attribute.

**infinity string *(string)*** This parameter can be set to parse a specific infinity representation (e.g. "Infinity"). If it is not set, the local specific infinity representation will be used.

**unparsable value handling *(selection)*** This selects the method for handling occurrences of values which are not parsable to numbers. The unparsable value can either be skipped, treated as an error or replaced with a missing value.

## Related Documents

- **Nominal to Numerical** (page 159)

## Tutorial Processes

### Nominal to Numeric conversion by the Parse Numbers operator

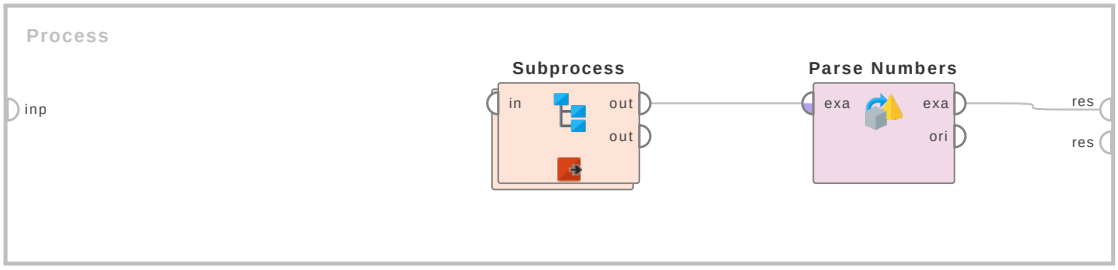
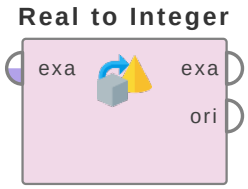



Figure 2.20: Tutorial process 'Nominal to Numeric conversion by the Parse Numbers operator'.

This Example Process starts with a Subprocess operator. The Subprocess operator provides an ExampleSet as its output. The ExampleSet has some nominal attributes. But these nominal attributes actually wrongly store numerical values as nominal values. A breakpoint is inserted here so that you can have a look at the ExampleSet. The type of these attributes should be numerical. To convert these nominal attributes to numerical attributes the Parse Numbers operator is applied. All parameters are used with default values. The resultant ExampleSet can be seen in the Results Workspace. You can see that the type of all attributes has been changed from nominal to numerical type.

# Real to Integer




This operator changes the type of the selected real attributes to integer type. It also maps all values of these attributes to integer values.

## Description

The Real to Integer operator converts selected real attributes to integer valued attributes. Each real value is either cut or rounded off and then used as an integer value of the new attribute. This option is controlled by the *round values* parameter. If it is set to false, the decimal portion of the real value is simply truncated otherwise it is rounded off. If the real value is missing, the new integer value will be missing.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. The ExampleSet should have at least one real attribute because if there is no such attribute, the use of this operator does not make sense.

## Output Ports

**example set output *(exa)*** The ExampleSet with selected real attributes converted to integer type is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When it is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression** ***(string)*** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type *(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type *(selection)*** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception *(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type *(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition *(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '> 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '> 6 && < 11' or '<= 5 || < 0'. But && and || cannot be used together in one numeric condition. Conditions like '(> 0 && < 2) || (>10 && < 12)' are not allowed because they use both && and ||. Use a blank space after '>', '=' and '<' e.g. '<5' will not work, so use '< 5' instead.

**include special attributes *(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection *(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**round values *(boolean)*** This parameter indicates if the values should be rounded off for conversion from real to integer. If not set to true, then the decimal portion of real values is simply truncated to convert the real values to integer values.

## Tutorial Processes

### Real to integer conversion of attributes of the Iris data set

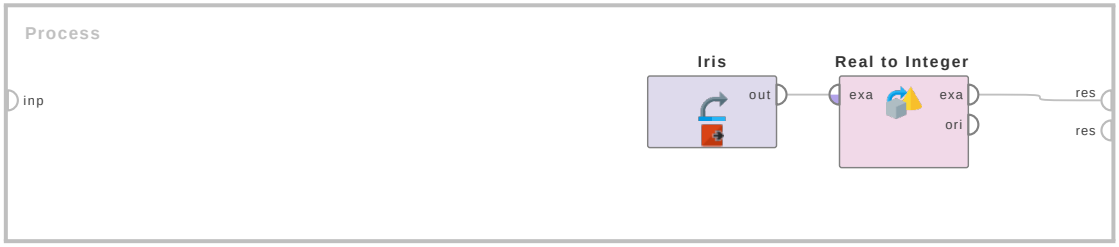
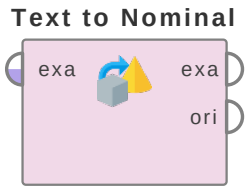



Figure 2.21: Tutorial process 'Real to integer conversion of attributes of the Iris data set'.

The ‘Iris’ data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has four real attributes i.e. a1, a2, a3 and a4. The Real to Integer operator is applied on the ‘Iris’ data set to convert the type of these real attributes to integer. All parameters are used with default values. The resultant ExampleSet can be seen in the Results Workspace. You can see that now the type of these attributes is integer.

# Text to Nominal




This operator changes the type of selected text attributes to nominal. It also maps all values of these attributes to corresponding nominal values.

## Description

The Text to Nominal operator converts all text attributes to nominal attributes. Each text value is simply used as a nominal value of the new attribute. If the value is missing in the text attribute, the new value will also be missing.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The ExampleSet should have at least one text attribute because if there is no such attribute, the use of this operator does not make sense.

## Output Ports

**example set output *(exa)*** The selected text attributes are converted to nominal and the resultant ExampleSet is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes on which you want to apply text to nominal conversion. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression** ***(string)*** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression** ***(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter *(except regular expression*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will not be selected even if they match the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be not be selected even if they match the previously mentioned block type i.e. *block type* parameter's value.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '> 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '> 6 && < 11' or '<= 5 || < 0'. But && and || cannot be used together in one numeric condition. Conditions like '(> 0 && < 2) || (>10 && < 12)' are not allowed because they use both && and ||. Use a blank space after '>', '=' and '<' e.g. '<5' will not work, so use '< 5' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

## Tutorial Processes

### Introduction to the Text to Nominal operator

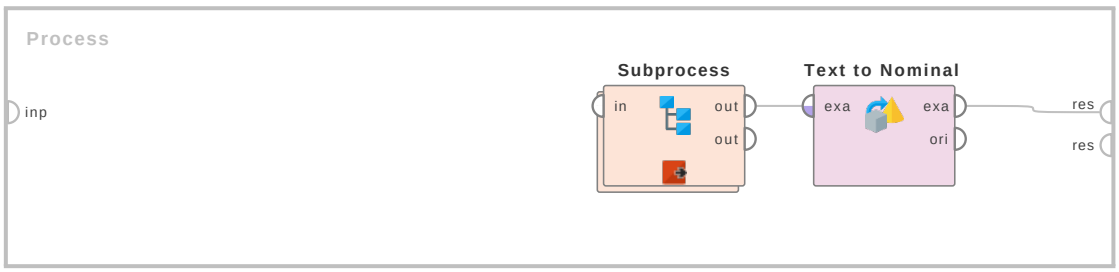
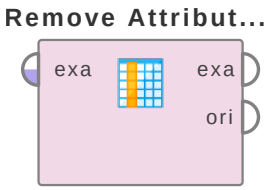



Figure 2.22: Tutorial process 'Introduction to the Text to Nominal operator'.

This Example Process starts with the Subprocess operator which provides an ExampleSet. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has three text attributes i.e. 'att1', 'att2' and 'att3'. The Text to Nominal operator is applied on this data set. The attribute filter type parameter is set to 'single' and the attribute parameter is set to 'att1'. Thus this operator converts the type of the 'att1' attribute from text to nominal. You can verify this by seeing the results in the Meta Data View in the Results Workspace.

## 2.1.3 Selection

### Remove Attribute Range




This operator removes a range of attributes from the given ExampleSet.

#### Description

The Remove Attribute Range operator removes the attributes within the specified range. The first and last attribute of the range are specified by the *first attribute* and *last attribute* parameters. All attributes in this range (including first and last attribute) will be removed from the ExampleSet. It is important to note that the attribute range starts from 1. This is a little different from the way attributes are counted in the Table Index where counting starts from 0. So, first and last attributes should be specified carefully.

#### Differentiation

- **Select Attributes** Provides a lot of options for selecting desired attributes e.g. on the basis of type, block, numerical value even regular expressions. See page 199 for details.
- **Remove Correlated Attributes** Selects attributes on the basis of correlations of the attributes. See page 192 for details.
- **Remove Useless Attributes** Selects attributes on the basis of usefulness. Different usefulness measures are available e.g. numerical attributes with minimum deviation etc. See page 195 for details.

#### Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

#### Output Ports

**example set output *(exa)*** The ExampleSet with selected attributes removed from the original ExampleSet is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**first attribute *(integer)*** The first attribute of the attribute range which should be removed is specified through this parameter. The counting of attributes starts from 1.

**last attribute *(integer)*** The last attribute of the attribute range which should be removed is specified through this parameter. The counting of attributes starts from 1.

## Related Documents

- **Select Attributes** (page 199)
- **Remove Correlated Attributes** (page 192)
- **Remove Useless Attributes** (page 195)

## Tutorial Processes

### Removing the first two attributes of the Golf data set

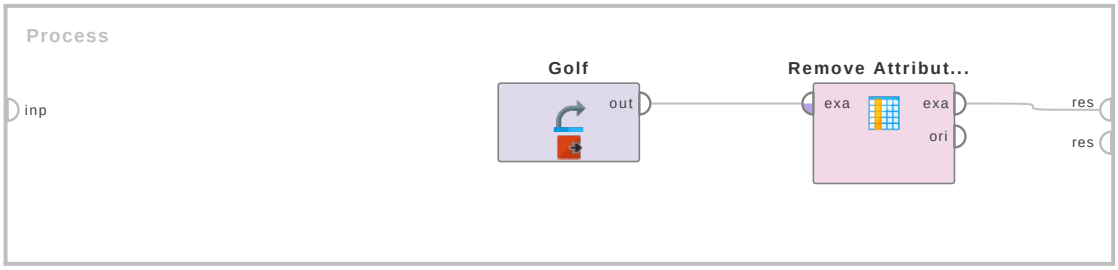
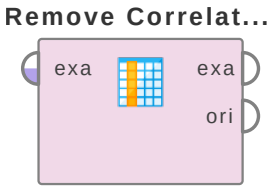



Figure 2.23: Tutorial process 'Removing the first two attributes of the Golf data set'.

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the Table Index of the Outlook attribute is 0. The Table Index column can be seen if the Show column 'Table Index' option is selected in the Meta Data View tab. The Table Index of the Temperature attribute is 1. The Remove Attribute Range operator is applied on the 'Golf' data set to remove the first two attributes. The first attribute and second attribute parameters are set to 1 and 2 respectively to remove the first two attributes. The first attribute and second attribute parameters were not set to 0 and 1 respectively because here attribute counting starts from 1 (instead of 0). The resultant ExampleSet can be seen in the Results Workspace. You can see that the Outlook and Temperature attributes have been removed from the ExampleSet.

# Remove Correlated Attributes




This operator removes correlated attributes from an ExampleSet. The correlation threshold is specified by the user. Correlation is a statistical technique that can show whether and how strongly pairs of attributes are related.

## Description

A correlation is a number between -1 and +1 that measures the degree of association between two attributes (call them X and Y). A positive value for the correlation implies a positive association. In this case large values of X tend to be associated with large values of Y and small values of X tend to be associated with small values of Y. A negative value for the correlation implies a negative or inverse association. In this case large values of X tend to be associated with small values of Y and vice versa.

Suppose we have two attributes X and Y, with means X' and Y' respectively and standard deviations S(X) and S(Y) respectively. The correlation is computed as summation from 1 to n of the product *(X(i)-X').(Y(i)-Y')* and then dividing this summation by the product *(n-1).S(X).S(Y)* where *n* is the total number of examples and *i* is the increment variable of summation. There can be other formulas and definitions but let us stick to this one for simplicity.

As discussed earlier a positive value for the correlation implies a positive association. Suppose that an X value was above average, and that the associated Y value was also above average. Then the product *(X(i)-X').(Y(i)-Y')* would be the product of two positive numbers which would be positive. If the X value and the Y value were both below average, then the product above would be of two negative numbers, which would also be positive. Therefore, a positive correlation is evidence of a general tendency that large values of X are associated with large values of Y and small values of X are associated with small values of Y.

As discussed earlier a negative value for the correlation implies a negative or inverse association. Suppose that an X value was above average, and that the associated Y value was instead below average. Then the product *(X(i)-X').(Y(i)-Y')* would be the product of a positive and a negative number which would make the product negative. If the X value was below average and the Y value was above average, then the product above would also be negative. Therefore, a negative correlation is evidence of a general tendency that large values of X are associated with small values of Y and small values of X are associated with large values of Y.

This operator can be used for removing correlated or uncorrelated attributes depending on the setting of parameters specially the *filter relation* parameter. The procedure is quadratic in number of attributes i.e. for *m* attributes an *m x m* matrix of correlations is calculated. Please note that this operator might fail in some cases when the attributes should be filtered out. For example, it might not be able to remove for example all negative correlated attributes because for the complete *m x m* - matrix of correlation the correlations will not be recalculated and hence not checked if one of the attributes of the current pair was already marked for removal. This means that for three attributes X, Y, and Z that it might be that Y was already ruled out by the negative correlation with X and is now not able to rule out Z any longer. The used correlation function in this operator is the Pearson correlation. In order to get more stable results the original, random, and reverse order of attributes is available.

Correlated attributes are usually removed because they are similar in behavior and will have similar impact in prediction calculations, so keeping attributes with similar impacts is redundant. Removing correlated attributes saves space and time of calculation of complex algorithms. Moreover, it also makes processes easier to design, analyze, understand and comprehend.

## Input Ports

**example set input** ***(exa)*** This input port expects an ExampleSet. It is the output of the Filter Examples operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set output** ***(exa)*** The (un-)correlated attributes are removed from the ExampleSet and this ExampleSet is delivered through this output port.

**original** ***(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**correlation** ***(real)*** This parameter specifies the correlation for filtering attributes. A correlation is a number between -1 and +1 that measures the degree of association between two attributes (call them X and Y). A positive value for the correlation implies a positive association. In this case large values of X tend to be associated with large values of Y and small values of X tend to be associated with small values of Y. A negative value for the correlation implies a negative or inverse association. In this case large values of X tend to be associated with small values of Y and vice versa.

**filter relation** ***(selection)*** Correlations of two attributes are compared at a time. One of the two attributes is removed if their correlation fulfills the relation specified by this parameter.

**attribute order** ***(selection)*** The algorithm takes this attribute order to calculate correlations and for filtering the attributes.

**use absolute correlation** ***(boolean)*** This parameter indicates if the absolute value of the correlations should be used for comparison.

## Tutorial Processes

### Removing correlated attributes from the Sonar data set

The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the ExampleSet before further operators are applied on it. You can see that the 'Sonar' data set has 60 numerical attributes. The Correlation Matrix operator is applied on it. This operator is applied so that you can view the correlation matrix of the 'Sonar' data set otherwise this operator was not required here. The Remove Correlated Attributes operator is applied on the 'Sonar' data set. The correlation parameter is set to 0.8. The filter relation parameter is set to 'greater' and the attribute order parameter is set to 'original'. Run the process and you will see in the Results Workspace that 19 out of 60 numerical attributes of the 'Sonar' data set have been removed. Now have a look at the correlation matrix generated by the Correlation Matrix operator. You can see that most of the attributes with correlations above 0.8 have been removed from the data set. Some such attributes are not removed because this operator might fail in some cases when the attributes should be filtered out. It might not be able to remove all correlated attributes because for the complete m x m matrix of correlation the correlations will not be recalculated and hence not checked if one of the attributes of the current pair was already marked

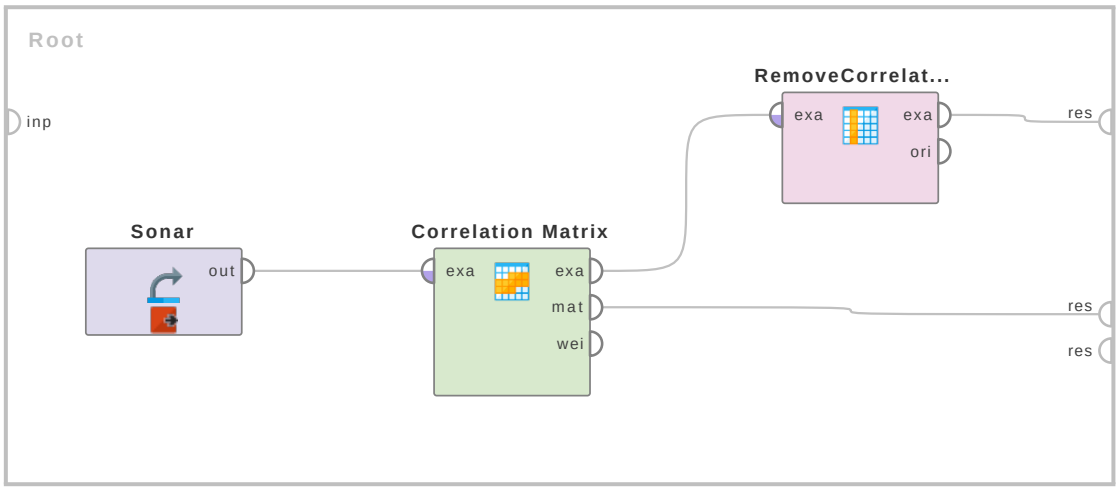
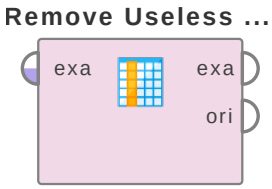



Figure 2.24: Tutorial process ‘Removing correlated attributes from the Sonar data set’.

for removal. Change the value of the attribute order parameter to ‘random’ and run the process again. Compare these results with the previous ones. This time a different set of attributes is removed from the data set. So, the order in which correlation operator is applied may change the output.

# Remove Useless Attributes




This operator removes useless attributes from an ExampleSet. The thresholds for useless attributes are specified by the user.

## Description

The Remove Useless Attributes operator removes four kinds of useless attributes:

1. Such nominal attributes where the most frequent value is contained in more than the specified ratio of all examples. The ratio is specified by the *nominal useless above* parameter. This ratio is defined as the number of examples with most frequent attribute value divided by the total number of examples. This property can be used for removing such nominal attributes where one value dominates all other values.
2. Such nominal attributes where the most frequent value is contained in less than the specified ratio of all examples. The ratio is specified by the *nominal useless below* parameter. This ratio is defined as the number of examples with most frequent attribute value divided by the total number of examples. This property can be used for removing nominal attributes with too many possible values.
3. Such numerical attributes where the Standard Deviation is less than or equal to a given deviation threshold. The *numerical min deviation* parameter specifies the deviation threshold. The Standard Deviation is a measure of how spread out values are. Standard Deviation is the square root of the Variance which is defined as the average of the squared differences from the Mean.
4. Such nominal attributes where the value of all examples is unique. This property can be used to remove id-like attributes.

Please note that this is not an intelligent operator i.e. it cannot figure out at its own whether an attribute is useless or not. It simply removes those attributes that satisfy the criteria for uselessness defined by the user.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Filter Examples operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set output *(exa)*** The attributes that satisfy the user-defined criteria for useless attributes are removed from the ExampleSet and this ExampleSet is delivered through this output port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**numerical min deviation *(real)*** The *numerical min deviation* parameter specifies the deviation threshold. Such numerical attributes where Standard Deviation is less than or equal to this deviation threshold are removed from the input ExampleSet. The Standard Deviation is a measure of how spread out values are. Standard Deviation is the square root of the Variance which is defined as the average of the squared differences from the Mean.

**nominal useless above *(real)*** The *nominal useless above* parameter specifies the ratio of the number of examples with most frequent value to the total number of examples. Such nominal attributes where the ratio of the number of examples with most frequent value to the total number of examples is more than this ratio are removed from the input ExampleSet. This property can be used to remove such nominal attributes where one value dominates all other values.

**nominal remove id like *(boolean)*** If this parameter is set to true, all such nominal attributes where the value of all examples is unique are removed from the input ExampleSet. This property can be used to remove id-like attributes.

**nominal useless below *(real)*** The nominal useless below parameter specifies the ratio of the number of examples with most frequent value to the total number of examples. Such nominal attributes where the ratio of the number of examples with most frequent value to the total number of examples is less than this ratio are removed from the input ExampleSet. This property can be used to remove nominal attributes with too many possible values.

## Tutorial Processes

### Removing useless nominal attributes from an ExampleSet

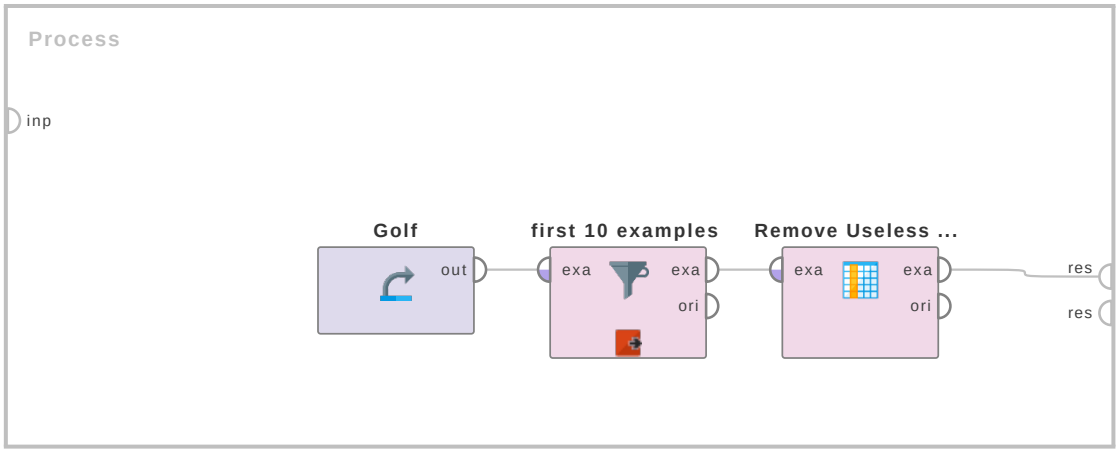



Figure 2.25: Tutorial process ‘Removing useless nominal attributes from an ExampleSet’.

This Example Process explains how the nominal useless above and nominal useless below parameters can be used to remove useless nominal attributes. Please keep in mind that the Remove Useless Attributes operator removes those attributes that satisfy the user-defined criteria for useless attributes.

The ‘Golf’ data set is loaded using the Retrieve operator. The Filter Examples operator is applied on it to filter the first 10 examples. This is done to just simplify the calculations for understanding this process. A breakpoint is inserted after the Filter Examples operator so that you can see the ExampleSet before application of the Remove Useless Attributes operator. You can see that the ExampleSet has 10 examples. There are 2 regular nominal attributes: ‘Outlook’ and ‘Wind’. The most frequent values in the ‘Outlook’ attribute are ‘rain’ and ‘sunny’, they occur in 4 out of 10 examples. Thus their ratio is 0.4. The most frequent value in the ‘Wind’ attribute is ‘false’, it occurs in 7 out of 10 examples. Thus its ratio is 0.7.

The Remove Useless Attributes operator is applied on the ExampleSet. The nominal useless above parameter is set to 0.6. Thus attributes where the ratio of most frequent value to total number of examples is above 0.6 are removed from the ExampleSet. As the ratio of most frequent value in the Wind attribute is greater than 0.6, it is removed from the ExampleSet.

The nominal useless below parameter is set to 0.5. Thus attributes where the ratio of most frequent value to total number of examples is below 0.5 are removed from the ExampleSet. As the ratio of most frequent value in the Outlook attribute is below 0.5, it is removed from the ExampleSet.

This can be verified by seeing the results in the Results Workspace.

### Removing useless numerical attributes from an ExampleSet

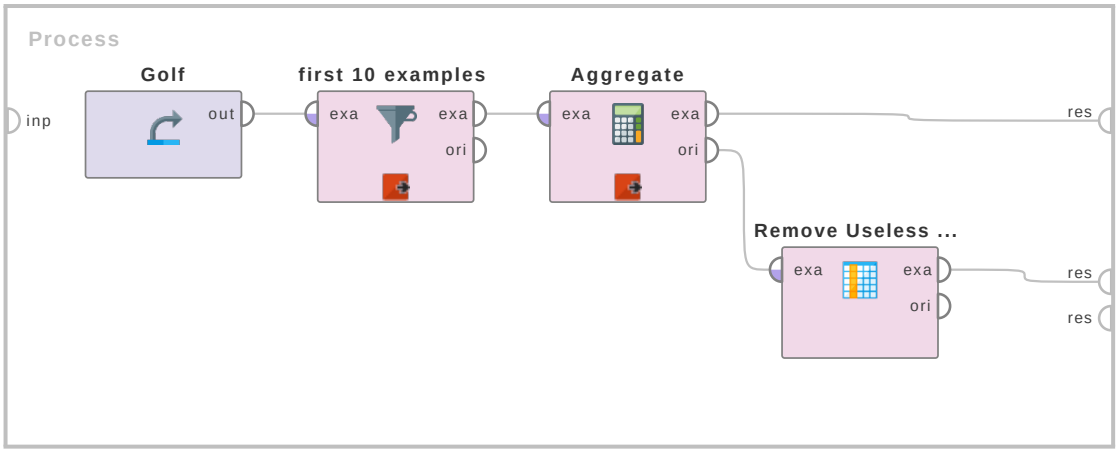



Figure 2.26: Tutorial process ‘Removing useless numerical attributes from an ExampleSet’.

This Example Process explains how the numerical min deviation parameter can be used to remove useless numerical attributes. The numerical min deviation parameter specifies the deviation threshold. Such numerical attributes where the Standard Deviation is less than or equal to this deviation threshold are removed from the input ExampleSet. The Standard Deviation is a measure of how spread out values are. Standard Deviation is the square root of the Variance which is defined as the average of the squared differences from the Mean. Please keep in mind that the Remove Useless Attributes operator removes those attributes that satisfy the user-defined criteria for useless attributes.

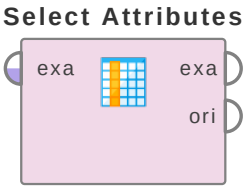
The ‘Golf’ data set is loaded using the Retrieve operator. The Filter Examples operator is applied on it to filter the first 10 examples. This is done to just simplify the calculations for understanding this process. A breakpoint is inserted after the Filter Examples operator so that you see the ExampleSet before application of the Remove Useless Attributes operator. You can see

that it has 10 examples. There are 2 regular numerical attributes: 'Temperature' and 'Humidity'. The Aggregate operator is applied on the ExampleSet to calculate and display the Standard Deviations of both numerical attributes. This operator is inserted here so that you can see that Standard Deviations without actually calculating them, otherwise this operator is not required here. You can see that the Standard Deviation of the 'Temperature' and 'Humidity' attributes is 7.400 and 10.682 respectively.

The Remove Useless Attributes operator is applied on the original ExampleSet (the ExampleSet with the first 10 examples of the 'Golf' data set). The numerical min deviation parameter is set to 9.0. Thus the numerical attributes where the Standard Deviation is less than 9.0 are removed from the ExampleSet. As the Standard Deviation of the Temperature attribute is less than 9.0, it is removed from the ExampleSet.

This can be verified by seeing the results in the Results Workspace.

# Select Attributes




This Operator selects a subset of Attributes of an ExampleSet and removes the other Attributes.

## Description

The Operator provides different filter types to make Attribute selection easy. Possibilities are for example: Direct selection of Attributes. Selection by a regular expression or selecting only Attributes without missing values. See parameter *attribute filter type* for a detailed description of the different filter types.

The *invert selection* parameter reverses the selection. Special Attributes (Attributes with Roles, like id, label, weight) are by default ignored in the selection. They will always remain in the resulting output ExampleSet. The parameter *include special attributes* changes this.

Only the selected Attributes are delivered to the output port. The rest is removed from the ExampleSet.

## Differentiation

*Select by <...> Operators*

There are several Operators which selects Attributes according to different input. For example the Select by Weights selects Attributes whose weights match a specified criterion. The Select by Random Operator selects a random subset of Attributes. The Remove Attribute Range removes a range of Attributes according to the index of the Attributes. The Remove Useless Attributes Operator removes Attributes which can be considered to be useless according to some specified criteria. The Remove Correlated Attributes Operator removes Attributes which are correlated to each other.

- **Work on Subset**

  This Operator is a combination of the Select Attributes Operator and the Subprocess Operator. It applies the Operators in its inner process to an ExampleSet with only the Attributes which are selected by the attribute filter type. The inner result is merged back to the whole input ExampleSet.

  See page 210 for details.

- **Forward Selection**

  This is an implementation of the forward selection feature selection method. It selects the most relevant Attributes according to an model which is trained inside the Operator. For details see the documentation of the Forward Selection Operator.

  See page 685 for details.

- **Backward Elimination**

  This is an implementation of the backward elimination feature selection method. It selects the most relevant Attributes according to an model which is trained inside the Operator. For details see the documentation of the Forward Selection Operator.

  See page 682 for details.

- **Filter Examples**

  This Operator does not select Attributes, but filters (or select) Examples. Thus it is the similar operation as the Select Attributes but applied on Examples instead of Attributes.

  See page 241 for details.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet for which you want to select Attributes from.

## Output Ports

**example set *(exa)*** The ExampleSet with only the selected Attributes is delivered to this output port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port.

## Parameters

**attribute filter type** This parameter allows you to select the Attribute selection filter; the method you want to use for selecting Attributes. It has the following options:

- **all** This option selects all the Attributes of the ExampleSet, no Attributes are removed. This is the default option.
- **single** This option allows the selection of a single Attribute. The required Attribute is selected by the *attribute* parameter.
- **subset** This option allows the selection of multiple Attributes through a list (see parameter *attributes*). If the meta data of the ExampleSet is known all Attributes are present in the list and the required ones can easily be selected.
- **regular_expression** This option allows you to specify a regular expression for the Attribute selection. The regular expression filter is configured by the parameters *regular expression, use except expression and except expression*.
- **value_type** This option allows selection of all the Attributes of a particular type. It should be noted that types are hierarchical. For example real and integer types both belong to the numeric type. The value type filter is configured by the parameters *value type, use value type exception, except value type*.
- **block_type** This option allows the selection of all the Attributes of a particular block type. It should be noted that block types may be hierarchical. For example value_series_start and value_series_end block types both belong to the value_series block type. The block type filter is configured by the parameters *block type, use block type exception, except block type*.
- **no_missing_values** This option selects all Attributes of the ExampleSet which do not contain a missing value in any Example. Attributes that have even a single missing value are removed.
- **numeric_value_filter** All numeric Attributes whose Examples all match a given numeric condition are selected. The condition is specified by the *numeric condition* parameter. Please note that all nominal Attributes are also selected irrespective of the given numerical condition.

**attribute** The required Attribute can be selected from this option. The Attribute name can be selected from the drop down box of the parameter if the meta data is known.

**attributes** The required Attributes can be selected from this option. This opens a new window with two lists. All Attributes are present in the left list. They can be shifted to the right list, which is the list of selected Attributes that will make it to the output port.

**regular expression** Attributes whose names match this expression will be selected. The expression can be specified through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression** If enabled, an exception to the first regular expression can be specified. This exception is specified by the *except regular expression* parameter.

**except regular expression** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in *regular expression* parameter).

**value type** This option allows to select a type of Attribute. One of the following types can be chosen: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date, time.

**use value type exception** If enabled, an exception to the selected type can be specified. This exception is specified by the *except value type* parameter.

**except value type** The Attributes matching this type will be removed from the final output even if they matched the before selected type, specified by the *value type* parameter. One of the following types can be selected here: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date, time.

**block type** This option allows to select a block type of Attribute. One of the following types can be chosen: single_value, value_series, value_series_start, value_series_end, value_matrix, value_matrix_start, value_matrix_end, value_matrix_row_start.

**use block type exception** If enabled, an exception to the selected block type can be specified. This exception is specified by the *except block type* parameter.

**except block type** The Attributes matching this block type will be removed from the final output even if they matched the before selected type by the *block type* parameter. One of the following block types can be selected here: single_value, value_series, value_series_start, value_series_end, value_matrix, value_matrix_start, value_matrix_end, value_matrix_row_start.

**numeric condition** The numeric condition used by the numeric condition filter type. A numeric Attribute is kept if all Examples match the specified condition for this Attribute. For example the numeric condition '$> 6$' will keep all numeric Attributes having a value of greater than 6 in every Example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Nominal Attributes are always kept, regardless of the specified numeric condition.

**include special attributes** Special Attributes are Attributes with special roles. These are: id, label, prediction, cluster, weight and batch. Also custom roles can be assigned to Attributes.

By default all special Attributes are delivered to the output port irrespective of the conditions in the Select Attribute Operator. If this parameter is set to true, special Attributes are also tested against conditions specified in the Select Attribute Operator and only those Attributes are selected that match the conditions.

**invert selection** If this parameter is set to true the selection is reversed. In that case all Attributes matching the specified condition are removed and the other Attributes remain in the output ExampleSet. Special Attributes are kept independent of the *invert selection* parameter as along as the *include special attributes* parameter is not set to true. If so the condition is also applied to the special Attributes and the selection is reversed if this parameter is checked.

## Tutorial Processes

### Selecting Attributes from the Titanic Data Sample

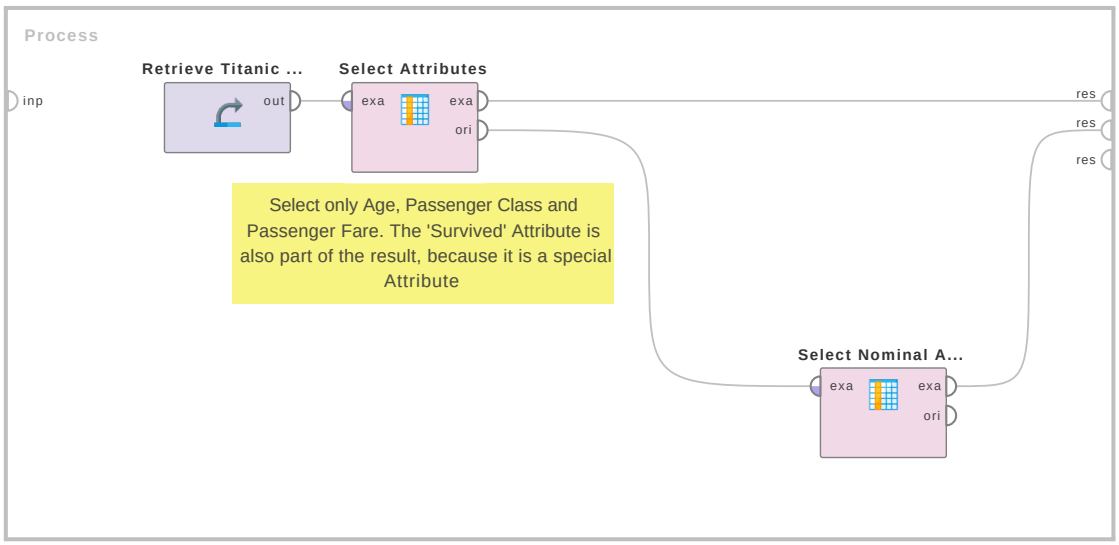



Figure 2.27: Tutorial process 'Selecting Attributes from the Titanic Data Sample'.

This tutorial Process show the basic usage of the Select Attributes Operator. First the 'Titanic' data is retrieved from the Samples folder. The first Select Attributes Operator selects a subset of the Attributes. The subset is specified by the attributes parameter.

The original output port is connected to the input port of the second Select Attributes Operator. There only nominal Attributes are selected.

### Different usages of the Select Attributes Operator

This tutorial Process demonstrates different usages of the Select Attributes Operator. A demo ExampleSet is created inside a Subprocess Oeprator. It has 3 special Attributes (id, label, weight) and 5 regular Attributes (att1, att2, att3, att4, att5). Also different attribute types are used (numeric: id; binominal: label; numeric: weight; real: att1, att2, att4, att5; nominal: att3). After the Subprocess Operator a Breakpoint is inserted, to investigate the demo ExampleSet.

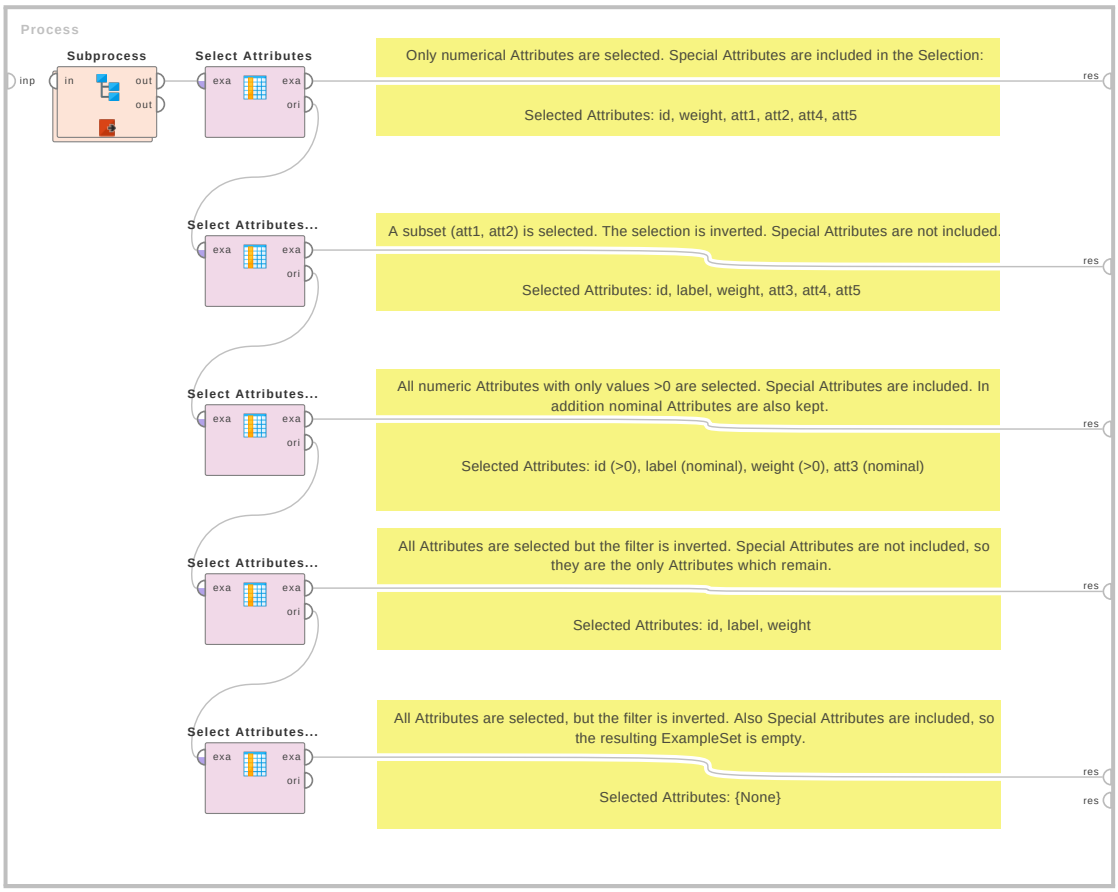



Figure 2.28: Tutorial process 'Different usages of the Select Attributes Operator'.

Next several Select Attributes Operators are used to show the different attribute filter types and the combinations with the parameters invert selection and include special attributes.

See the comments in the process for more details.

### Selecting Attributes by using a regular expression

This tutorial Process illustrates the usage of a regular expression to select Attributes from the Labor-Negotiations data sample. The regular expression specified is: w.*|.*y.*

This means all Attributes starting with a 'w' (w.*) or (|) all Attributes whose name contains a 'y' in their name (.*y.*) matches the expression. The following Attributes of the Labor-Negotiations data set match this expression:

wage-inc-1st, wage-inc-2nd, wage-inc-3rd, working-hours, standby-pay, statutory-holidays, longterm-disability-assistance.

The use except expression parameter is also set to true. Thus Attributes that match the condition in the except regular expression parameter will be removed. The specified except regular expression is: .*\[0-9\].*. This means all Attributes whose name contains a digit are removed.

Finally the following four Attributes are selected: working-hours, standby-pay, statutory-holidays, longterm-disability-assistance. Besides the special Attribute class is also kept.

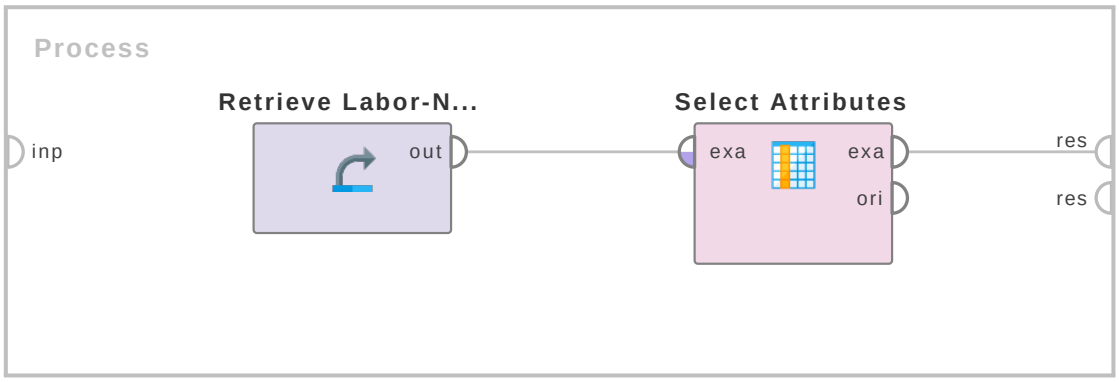
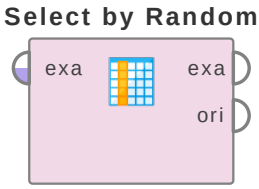



Figure 2.29: Tutorial process ‘Selecting Attributes by using a regular expression’.

For more details about regular expression see the configuration of the regular expression parameter.

# Select by Random




This operator selects a random subset of attributes of the given ExampleSet.

## Description

The Select by Random operator selects attributes randomly from the input ExampleSet. If the *use fixed number of attributes* parameter is set to true, then the required number of attributes is specified through the *number of attributes* parameter. Otherwise, a random number of attributes is selected. The randomization can be changed by changing the seed value in the corresponding parameters. This operator can be useful in combination with the Loop Parameters operator or can be used as a baseline for significance test comparisons for feature selection techniques.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with the data.

## Output Ports

**example set *(exa)*** The ExampleSet with selected attributes is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**use fixed number of attributes *(boolean)*** This parameter specifies if a fixed number of attributes should be selected.

**number of attributes *(integer)*** This parameter is only available when the *use fixed number of attributes* parameter is set to true. This parameter specifies the number of attributes which should be randomly selected.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same ExampleSet. Changing the value of the local seed changes the randomization, thus the ExampleSet will have a different set of attributes.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Selecting random attributes from Sonar data set

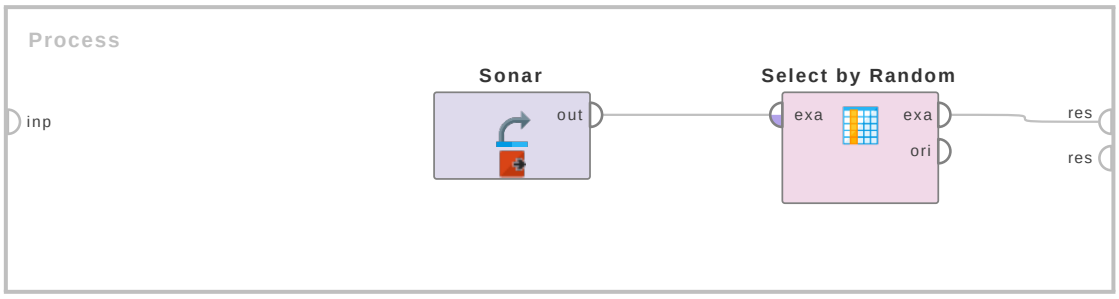
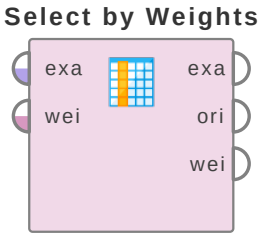



Figure 2.30: Tutorial process ‘Selecting random attributes from Sonar data set’.

The ‘Sonar’ data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see the ExampleSet has 60 attributes. The Select by Random operator is applied on this ExampleSet. The use fixed number of attributes parameter is set to true and the number of attributes parameter is set to 10. Thus 10 attributes will be selected randomly from the ‘Sonar’ data set. The resultant ExampleSet can be seen in the Results Workspace.

# Select by Weights




This operator selects only those attributes of an input ExampleSet whose weights satisfy the specified criterion with respect to the input weights.

## Description

This operator selects only those attributes of an input ExampleSet whose weights satisfy the specified criterion with respect to the input weights. Input weights are provided through the *weights* input port. The criterion for attribute selection by weights is specified by the *weight relation* parameter.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with data

**weights *(wei)*** This port expects the attribute weights. There are numerous operators that provide the attribute weights. The Weight by Correlation operator is used in the Example Process.

## Output Ports

**example set *(exa)*** The ExampleSet with selected attributes is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**weights *(wei)*** The Attributes weights that were provided at the *weights* input port are delivered through this output port.

## Parameters

**weight relation** Only those attributes are selected whose weights satisfy this relation.

- **greater** Attributes whose weights are greater than the *weight* parameter are selected.
- **greater_equals** Attributes whose weights are equal or greater than the *weight* parameter are selected.
- **equals** Attributes whose weights are equal to the *weight* parameter are selected.
- **less_equals** Attributes whose weights are equal or less than the *weight* parameter are selected.

- **less** Attributes whose weights are less than the *weight* parameter are selected.
- **top_k** The k attributes with highest weights are selected. k is specified by the *k* parameter.
- **bottom_k** The k attributes with lowest weights are selected. k is specified by the *k* parameter.
- **all_but_top_k** All attributes other than the k attributes with highest weights are selected. k is specified by the *k* parameter.
- **all_but_bottom_k** All attributes other than k attributes with lowest weights are selected. k is specified by the *k* parameter.
- **top_p%** The top *p* percent attributes with highest weights are selected. p is specified by the *p* parameter.
- **bottom_p%** The bottom *p* percent attributes with lowest weights are selected. p is specified by the *p* parameter.

**weight** This parameter is available only when the weight relation parameter is set to 'greater', 'greater equals', 'equals', 'less equals' or 'less'. This parameter is used to compare weights.

**k** This parameter is available only when the weight relation parameter is set to 'top k', 'bottom k', 'all but top k' or 'all but bottom k'. It is used to count the number of attributes to select.

**p** This parameter is available only when the weight relation parameter is set to 'top p%' or 'bottom p%'. It is used to specify the percentage of attributes to select.

**deselect unknown** This is an expert parameter. This parameter indicates if attributes whose weight is unknown should be removed from the ExampleSet.

**use absolute weights** This is an expert parameter. This parameter indicates if the absolute values of the weights should be used for comparison.

## Tutorial Processes

### Selecting attributes from Sonar data set

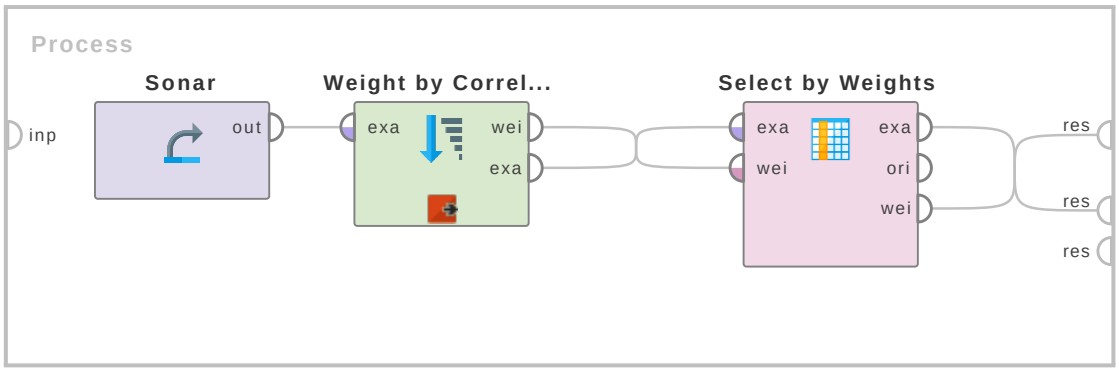
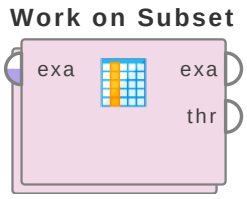



Figure 2.31: Tutorial process 'Selecting attributes from Sonar data set'.

The 'Sonar' data set is loaded using the Retrieve operator. The Weight by Correlation operator is applied on it to generate attribute weights. A breakpoint is inserted here. You can see the

attributes with their weights here. The Select by Weights operator is applied next. The 'Sonar' data set is provided at the exampleset port and weights calculated by the Weight by Correlation operator are provided at the weights input port. The weight relation parameter is set to 'bottom k' and the k parameter is set to 4. Thus 4 attributes with minimum weights are selected. As you can see the 'attribute_57', 'attribute_17', 'attribute_30' and 'attribute_16' have lowest weights, thus these four attributes are selected. Also note that the label attribute 'class' is also selected. This is because the attributes with special roles are selected irrespective of weights condition.

# Work on Subset




This operator selects a subset (one or more attributes) of the input ExampleSet and applies the operators in its subprocess on the selected subset.

## Description

The Work on the Subset operator can be considered as the blend of the Select Attributes and Subprocess operator to some extent. The attributes are selected in the same way as selected by the Select Attributes operator and the subprocess of this operator works in the same way as the Subprocess operator works. A subprocess can be considered as small unit of a process where all operators and a combination of operators can be applied in a subprocess. That is why a subprocess can also be defined as a chain of operators that is subsequently applied. For more information about subprocess please study the Subprocess operator. Although the Work on Subset operator has similarities with the Select Attributes and Subprocess operators however, this operator provides some functionality that cannot be performed by the combination of the Select Attributes and Subprocess operator. Most importantly, this operator can merge the results of its subprocess with the input ExampleSet such that the original subset is overwritten by the subset received after processing of the subset in the subprocess. This merging can be controlled by the *keep subset only* parameter. This parameter is set to false by default. Thus merging is done by default. If this parameter is set to true, then only the result of the subprocess is returned by this operator and no merging is done. In such a case this operator behaves very similar to the combination of the Select Attributes and Subprocess operator. This can be understood easily by studying the attached Example Process.

This operator can also deliver the additional results of the subprocess if desired. This can be controlled by the *deliver inner results* parameter. Please note that this is a very powerful operator. It can be used to create new preprocessing schemes by combining it with other preprocessing operators. However, there are two major restrictions:

- Since the result of the subprocess will be combined with the rest of the input ExampleSet, the number of examples is not allowed to be changed inside the subprocess.
- The changes in the role of an attribute will not be delivered outside the subprocess.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set *(exa)*** The result of the subprocess will be combined with the rest of the input ExampleSet and delivered through this port. However if the *keep subset only* parameter is set to true then only the result of the subprocess will be delivered.

**through *(thr)*** This operator can also deliver the additional results of the subprocess if desired. This can be controlled by the *deliver inner results* parameter. This port is used for delivering the additional results of the subprocess.The Work on Subset operator can have multiple *through* ports. When one *through* port is connected, another *through* port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object passed at the first *through* port inside the subprocess of the Work on Subset operator is delivered at the first *through* port of the operator.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose all examples satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute *(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *parameter* attribute if the meta data is known.

**attributes *(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list, which is the list of selected attributes.

**regular expression** ***(string)*** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression** ***(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter *(except regular expression*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will not be selected even if they match the previously mentioned type i.e. *value type* parameter's value.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be not be selected even if they match the previously mentioned block type i.e. *block type* parameter's value.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles. Special attributes are those attributes which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**name conflict handling** ***(selection)*** This parameter decides how to handle a conflict with names when the Operator merges the Subset back to the ExampleSet. There are three possible behaviors:

- **error** The Operator will show an Error if there is any conflict.
- **keep new** If there is an conflict, the Operator will keep the one from the Subset.The other one will be deleted.
- **keep original** If there is an conflict, the Operator will keep the one which is not in the Subset. The other one will be deleted.

**role conflict handling** ***(selection)*** This parameter decides how to handle a conflict with roles when the Operator merges the Subset back to the ExampleSet. There are three possible behaviors:

- **error** The Operator will show an Error if there is any conflict.
- **keep new** If there is an conflict, the Operator will keep the one from the Subset.The other one will be deleted.
- **keep original** If there is an conflict, the Operator will keep the one which is not in the Subset. The other one will be deleted.

**keep subset only** ***(boolean)*** The Work on Subset operator can merge the results of its subprocess with the input ExampleSet such that the original subset is overwritten by the subset received after processing of the subset in the subprocess. This merging can be controlled by the *keep subset only* parameter. This parameter is set to false by default. Thus merging is done by default. If this parameter is set to true, then only the result of the subprocess is returned by this operator and no merging is done.

**deliver inner results** ***(boolean)*** This parameter indicates if the additional results (other than the input ExampleSet) of the subprocess should also be returned. If this parameter is set to true then the additional results are delivered through the *through* ports.

**remove roles** ***(boolean)*** This parameter decides if the role of the Special Attributes in the Subset will be removed by entering the Subset or not.

## Tutorial Processes

### Working on a subset of Golf data set

The 'Golf' data set is loaded using the Retrieve operator. Then the Work on Subset operator is applied on it. The attribute filter type parameter is set to subset. The attributes parameter is used for selecting the 'Temperature' and 'Humidity' attributes. Double-click on the Work on Subset operator to see its subprocess. All the operations in the subprocess will be performed only on the selected attributes i.e. the 'Temperature' and 'Humidity' attributes. The Normalize operator is applied in the subprocess. The attribute filter type parameter of the Normalize operator is set to 'all'. Please note that the Normalize operator will not be applied on 'all' the attributes of the input ExampleSet rather it would be applied on 'all' selected attributes of the input ExampleSet i.e. the 'Temperature' and 'Humidity' attributes. Run the process. You will see that the normalized 'Humidity' and 'Temperature' attribute are combined with the rest of the input ExampleSet. Now set the keep subset only parameter to true and run the process again. Now you will see that only the results of the subprocess are delivered by the Work on Subset operator. This Example Process just explains the basic usage of this operator. This operator can be used for creating new preprocessing schemes by combining it with other preprocessing operators.

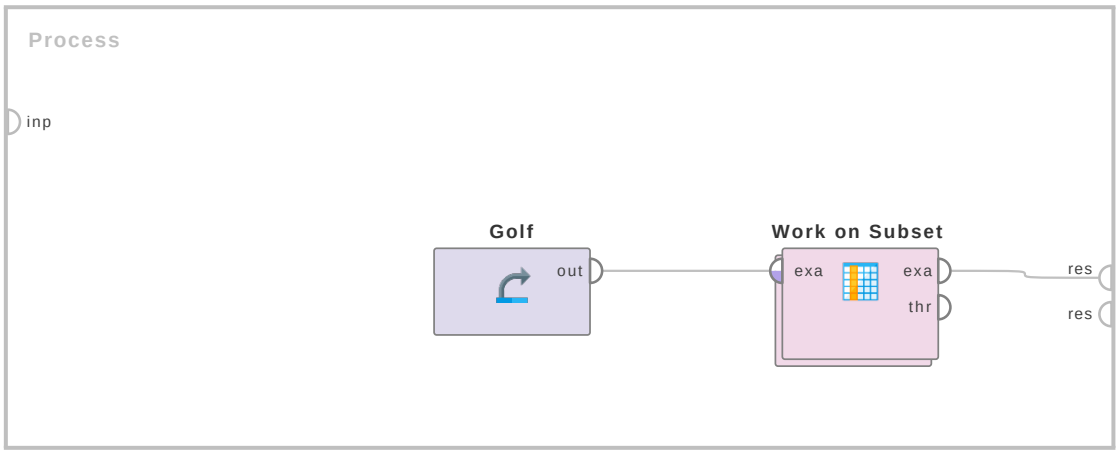
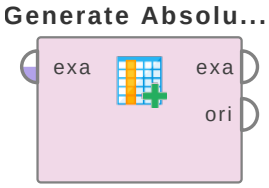



Figure 2.32: Tutorial process 'Working on a subset of Golf data set'.

## 2.1.4 Generation

### Generate Absolutes




This operator replaces all values of the selected numerical attributes by their corresponding absolute values.

#### Description

The Generate Absolutes operator replaces all values of the selected numerical attributes by their absolute values. The absolute value of a real number is the numerical value of that number without regard to its sign. For example, the absolute value of 7 is 7, and the absolute value of −7 is also 7. The absolute value of a number may be thought of as its distance from zero.

#### Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

#### Output Ports

**example set output *(exa)*** The values of the selected numerical attributes are replaced by their corresponding absolute values and the resultant ExampleSet is returned through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When it is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute *(string)*** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes *(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression *(string)*** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression *(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except regular expression *(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type *(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception *(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type *(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type *(selection)*** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception *(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type *(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition *(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**invert selection *(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**include special attributes *(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

## Tutorial Processes

### Absolute values of the Ripley-Set data set

The 'Ripley-Set' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has two real

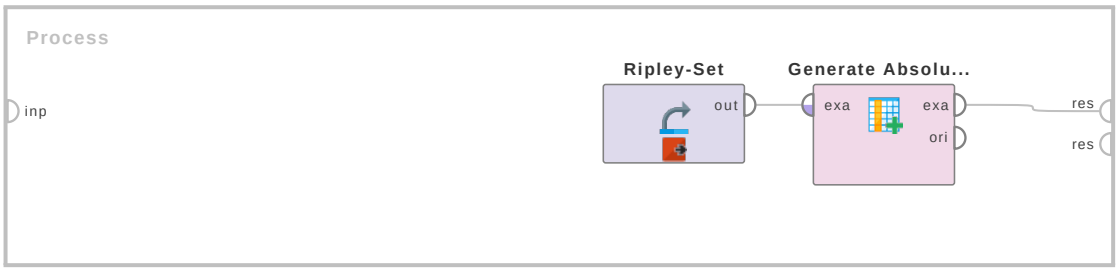
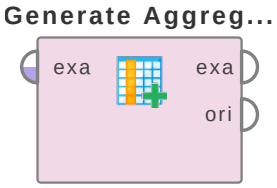



Figure 2.33: Tutorial process 'Absolute values of the Ripley-Set data set'.

attributes i.e. att1 and att2. Note that both these attributes have both positive and negative values. The Generate Absolutes operator is applied on this ExampleSet to replace these values by their corresponding absolute values. The resultant ExampleSet can be seen in the Results Workspace.

# Generate Aggregation




This operator generates a new attribute by performing the specified aggregation function on every example of the selected attributes.

## Description

This operator can be considered to be a blend of the Generate Attributes operator and the Aggregate operator. This operator generates a new attribute which consists of a function of several other attributes. These 'other' attributes can be selected by the *attribute filter type* parameter and other associated parameters. The aggregation function is selected through the *aggregation function* parameter. Several aggregation functions are available e.g. count, minimum, maximum, average, mode etc. The *attribute name* parameter specifies the name of the new attribute. If you think this operator is close to your requirement but not exactly what you need, have a look at the Aggregate and the Generate Attributes operators because they perform similar tasks.

## Differentiation

- **Aggregate** This operator performs the aggregation functions known from SQL. It provides a lot of functionalities in the same format as provided by the SQL aggregation functions. SQL aggregation functions and GROUP BY and HAVING clauses can be imitated using this operator. See page 260 for details.
- **Generate Attributes** It is a very powerful operator for generating new attributes from existing attributes. It even supports regular expressions and conditional statements for specifying the new attributes See page 222 for details.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set output *(exa)*** The ExampleSet with the additional attribute generated after applying the specified aggregation function is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute name *(string)*** The name of the resulting attribute is specified through this parameter.

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute *(string)*** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes *(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression *(string)*** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression *(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**aggregation function** ***(selection)*** This parameter specifies the function for aggregating the values of the selected attribute. Numerous options are available e.g. average, variance, standard deviation, count, minimum, maximum, sum, mode, median, product and concatenation.

**concatenation separator** ***(string)*** This parameter specifies the separator between the concatenated values. Only visible if the "concatenation" aggregation function is selected.

**keep all** ***(boolean)*** This parameter indicates if all old attributes should be kept. If this parameter is set to false then all the selected attributes (i.e. attributes that are used for aggregation) are removed.

**ignore missings *(boolean)*** This parameter indicates if missing values should be ignored and if the aggregation function should be only applied on existing values. If this parameter is not set to true the aggregated value will be a missing value in the presence of missing values in the selected attribute.

**ignore missing attributes *(boolean)*** Normally an error is shown when the attribute filter doesn't match any attributes of the ExampleSet. If this parameter is set to true, that situation will be ignored.

## Related Documents

- **Aggregate** (page 260)
- **Generate Attributes** (page 222)

## Tutorial Processes

### Generating an attribute having average of real attributes of Sonar data set

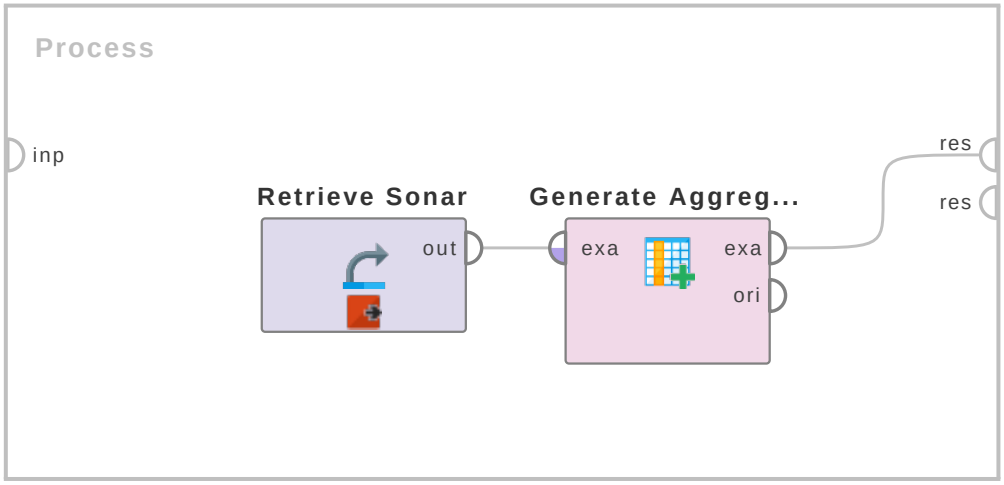


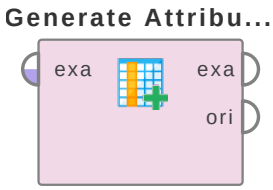

Figure 2.34: Tutorial process 'Generating an attribute having average of real attributes of Sonar data set'.

The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has one nominal and sixty real attributes. The Generate Aggregation operator is applied on this ExampleSet to generate a new attribute from the real attributes of the ExampleSet.

The attribute name parameter is set to 'Average' thus the new attribute will be named 'Average'. The attribute filter type parameter is set to 'value type' and the value type parameter is set to 'real', thus the new attribute will be created from real attributes of the ExampleSet. The aggregation function parameter is set to 'average', thus the new attribute will be average of the selected attributes.

The resultant ExampleSet can be seen in the Results Workspace. You can see that there is a new attribute named 'Average' in the ExampleSet that has the average value of the attribute_1, attribute_2, ..., attribute_60 attributes.

# Generate Attributes




This operator constructs new user defined attributes using mathematical expressions.

## Description

The Generate Attributes operator constructs new attributes from the attributes of the input ExampleSet and arbitrary constants using mathematical expressions. The attribute names of the input ExampleSet might be used as variables in the mathematical expressions for new attributes. During the application of this operator these expressions are evaluated on each example, these variables are then filled with the example's attribute values. Thus this operator not only creates new columns for new attributes, but also fills those columns with corresponding values of those attributes. If a variable is undefined in an expression, the entire expression becomes undefined and '?' is stored at its location.

Please note that there are some restrictions for the attribute names in order to let this operator work properly:

- Attribute names containing dashes '-' or other special characters, or having the same name as a constant (e.g. 'e' or 'pi') must be placed in square brackets e.g. '[weird-name]' or '[pi]'.
- Attribute names containing square brackets or backslashes must be placed in square brackets and the square brackets and backslashes inside the name must be escaped, e.g. '[a\\tt\[1\]]' for an attribute 'a\tt[1]'.

If you want to apply this operator but the attributes of your ExampleSet do not fulfill above mentioned conditions you can rename attributes with the Rename operator before application of the Generate Attributes operator. When replacing several attributes following a certain schema, the Rename by Replacing operator might prove useful.

A large number of operations and functions is supported, which allows you to write rich expressions. For a list of operations and functions and their descriptions open the Edit Expression dialog. Complicated expressions can be created by using multiple operations and functions. Parenthesis can be used to nest operations.

This operator also supports various constants (for example 'INFINITY', 'PI' and 'e'). Again you can find a complete list in the Edit Expression dialog. You can also use strings in operations but the string values should be enclosed in double quotes (").

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Rename operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set *(exa)*** The ExampleSet with new attributes is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the results workspace.

## Parameters

**function descriptions** The list of functions for generating new attributes is provided here.

**keep all *(boolean)*** If set to true, all the original attributes are kept, otherwise they are removed from the output ExampleSet.

## Tutorial Processes

### Generating attributes through different function descriptions

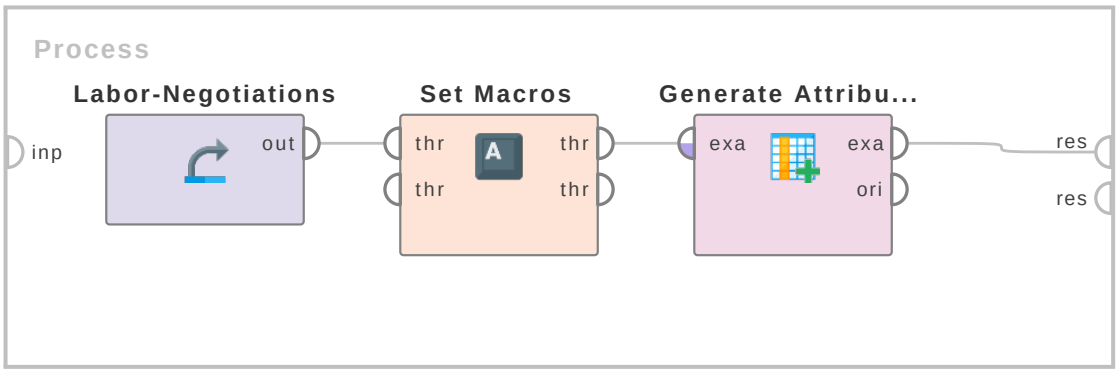



Figure 2.35: Tutorial process 'Generating attributes through different function descriptions'.

The 'Labor-Negotiations' data set is loaded using the Retrieve operator.

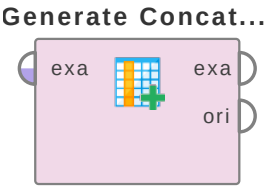
Now have a look at the Generate Attributes operator's parameters. The keep all parameter is checked, thus all attributes of the 'Labor-Negotiations' data set are also kept along with attributes generated by the Generate Attributes operator.

Click on the Edit List button of the function descriptions parameter to have a look at descriptions of functions defined for generating new attributes. 18 new attributes are generated, there might be better ways of generating these attributes but here they are written to explain the usage of the different type of functions available in the Generate Attributes operator. Please read the function description of each attribute and then see the values of the corresponding attribute in the Results Workspace to understand it completely. Here is a description of attributes created by this operator:

The 'average wage-inc' attribute takes sum of the wage-inc-1st, wage-inc-2nd and wage-inc-3rd attribute values and divides the sum by 3. This gives an average of wage-increments. There are better ways of doing this, but this example was just shown to clarify the use of some basic functions. The 'neglected worker bool' attribute is a boolean attribute i.e. it has only two possible values '0' and '1'. This attribute was created here to show usage of logical operations like 'AND' and 'OR' in the Generate Attributes operator. This attribute assumes value '1' if three conditions are satisfied. First, the working-hours attribute has value 35 or more. Second, the education-allowance attribute is not equal to 'yes'. Third, the vacation attribute has value 'average' OR 'below-average'. If any of these conditions is not satisfied, the new attribute gets

value ‘0’. The ‘logarithmic attribute’ attribute shows the usage of logarithm base 10 and natural logarithm functions. The ‘trigno attribute’ attribute shows the usage of various trigonometric functions like sine and cosine. The ‘rounded average wage-inc’ attribute uses the avg function to take average of wage-increments and then uses the round function to round the resultant values. The ‘vacations’ attribute uses the replaceAll function to replace all occurrences of value ‘generous’ with ‘above-average’ in the ‘vacation’ attribute. The ‘deadline’ attribute shows usage of the If-then-Else and Date functions. This attribute assumes value of current date plus 25 days if class attribute has value ‘good’. Otherwise it stores the date of the current date plus 10 days. The ‘shift complete’ attribute shows the usage of the If-then-Else, random, floor and missing functions. This attribute has values of the shift-differential attribute but it does not have missing values. Missing values are replaced with a random number between 0 and 25. The ‘remaining_holidays’ attribute stores the difference of the statutory-holidays attribute value from 15. The ‘remaining_holidays_percentage’ attribute uses the ‘remaining_holidays’ attribute to find the percentage of remaining holidays. This attribute was created to show that attributes created in this Generate Attribute operator can be used to generate new attributes in the same Generate Attributes operator. The ‘constants’ attribute was created to show the usage of constants like ‘e’ and ‘PI’. The ‘cut’ attribute shows the usage of cut function. If you want to specify a string, you should place it in double quotes (””) as in the last term of this attribute’s expression. If you want to specify name of an attribute you should not place it in the quotes. First term of expression cuts first two characters of the ‘class’ attribute values. This is because name of attribute is not placed in quotes. Last term of the expression selects first two characters of the string ‘class’. As first two characters of string ‘class’ are ‘cl’, thus cl is appended at the end of this attribute’s values. The middle term is used to concatenate a blank space between first and last term’s results. The ‘index’ attribute shows usage of the index function. If the ‘class’ attribute has value ‘no’, 1 is stored because ‘o’ is at first index. If the ‘class’ attribute has value ‘yes’, -1 is stored because ‘o’ is not present in this value. The ‘date constants’ attribute shows the usage of the date constants. It shows the date of the ‘deadline’ attribute in full format, but only time is selected for display. The ‘macro’ attribute shows how to use macros in functions.The ‘macro eval’ attribute shows how to use macros that contain a number. The macro function %{} always returns a string, so if you want to obtain the number you have to use the eval function or the parse function.The ‘expression eval’ attribute shows usage of the eval function. If there is a string containing an expression, for example coming from a macro %{expression} you can evaluate this expression by using the eval function. The ‘macro with attribute’ attribute shows the usage of the #{} function. If there is a macro containing the name of an attribute, you can use this attribute in your expression by using #{attribute_macro} where attribute_macro is the macro containing the attribute name. Using eval(%{attribute_macro}) would lead to the same result, but the #{} function fails when the macro does not contain an attribute name, while eval(%{attribute_macro}) evaluates whatever is contained in the macro.

# Generate Concatenation




This operator merges two attributes into a single new attribute by concatenating their values. The new attribute is of nominal type. The original attributes remain unchanged.

## Description

The Generate Concatenation operator merges two attributes of the input ExampleSet into a single new nominal attribute by concatenating the values of the two attributes. If the resultant attribute is actually of numerical type, it can be converted from nominal to numerical type by using the Nominal to Numeric operator. The original attributes remain unchanged, just a new attribute is added to the ExampleSet. The two attributes to be concatenated are specified by the *first attribute* and *second attribute* parameters.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

## Output Ports

**example set output *(exa)*** The ExampleSet with the new attribute that has concatenated values of the specified attributes is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**first attribute *(string)*** This parameter specifies the first attribute to be concatenated.

**second attribute *(string)*** This parameter specifies the second attribute to be concatenated.

**separator *(string)*** This parameter specifies the string which is used as separation of values of the first and second attribute i.e. the string that is concatenated between the two values.

**trim values *(boolean)*** This parameter indicates if the values of the first and second attribute should be trimmed i.e. leading and trailing whitespaces should be removed before the concatenation is performed.

## Tutorial Processes

### Generating a concatenated attribute in the Labor-Negotiations data set

The 'Labor-Negotiations' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. The 'vacation' and 'statutory-holidays' attributes will be concatenated to form a new attribute. The Generate Concatenation operator is

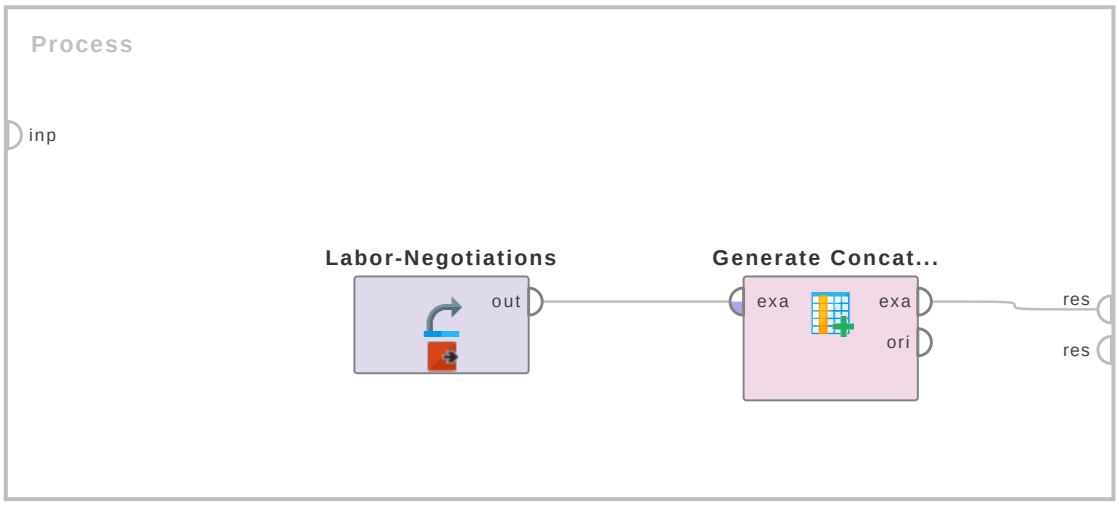
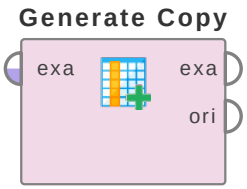



Figure 2.36: Tutorial process 'Generating a concatenated attribute in the Labor-Negotiations data set'.

applied on the Labor-Negotiations data set. The first attribute and second attribute parameters are set to 'vacation' and 'statutory-holidays' respectively. The separator parameter is set to '-_'. Thus the values of the 'vacation' and 'statutory-holidays' attributes will be merged with a '_' between them. You can verify this by seeing the resultant ExampleSet in the Results Workspace. The 'vacation' and 'statutory-holidays' attributes remain unchanged. A new attribute named 'vacation_statutory-holidays' is created. The type of the new attribute is nominal.

# Generate Copy




This operator generates the copy of an attribute. The original attribute remains unchanged.

## Description

The Generate Copy operator adds a copy of the selected attribute to the input ExampleSet. Please note that the original attribute remains unchanged, just a new attribute is added to the ExampleSet. The attribute whose copy is required is specified by the *attribute name* parameter. The name of the new attribute is specified through the *new name* parameter. Please note that the names of attributes of an ExampleSet should be unique. Please note that only the view on the data column is copied, not the data itself.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

## Output Ports

**example set output *(exa)*** The ExampleSet with the new attribute that is a copy of the specified attribute is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute name *(string)*** The attribute whose copy is required is specified by the *attribute name* parameter.

**new name *(string)*** The name of the new attribute is specified through the *new name* parameter. Please note that the names of attributes of an ExampleSet should be unique.

## Tutorial Processes

### Generating a copy of the Temperature attribute of the Golf data set

The 'Golf' data set is loaded using the Retrieve operator. The Generate Copy operator is applied on it. The attribute name parameter is set to 'Temperature'. The new name parameter is set to 'New Temperature'. Run the process. You will see that an attribute named 'New Temperature' has been added to the 'Golf' data set. The new attribute has the same values as the 'Temperature' attribute. The 'Temperature' attribute remains unchanged.

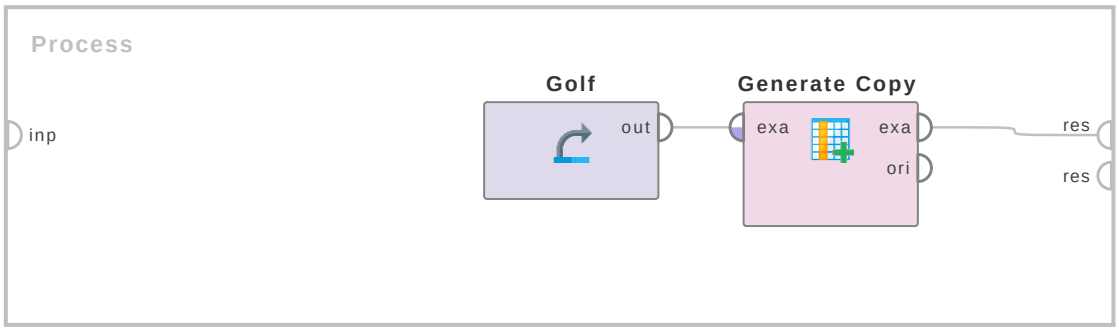
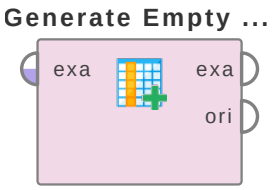



Figure 2.37: Tutorial process 'Generating a copy of the Temperature attribute of the Golf data set'.

## Generate Empty Attribute




This operator adds a new attribute of specified name and type to the input ExampleSet.

### Description

The Generate Empty Attribute operator creates an empty attribute of specified name and type which are specified by the *name* and the *value type* parameter respectively. One of the following types can be selected: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date, time. Please note that all values are missing right after creation of the attribute. The operators like the Set Data operator can be used to fill values of this attribute. Please note that the name of the attribute can be changed later by the Rename operator and many type conversion operators are also available for changing the type of the attribute. Please note that this operator creates an empty attribute independent of the input ExampleSet, if you want to generate an attribute from the existing attributes of the input ExampleSet you can use the Generate Attributes operator.

### Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

### Output Ports

**example set output *(exa)*** An empty attribute of the specified name and type is added to the input ExampleSet and the resultant ExampleSet is delivered through this output port.

**original** ***(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**name** ***(string)*** This parameter specifies the name of the new attribute. Please note that the names of attributes should be unique. Please make sure that the input ExampleSet does not have an attribute with the same name.

**value type** ***(selection)*** The type of the new attribute is specified by this parameter. One of the following types can be selected: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date, time.

## Tutorial Processes

### Adding an empty attribute to the 'Golf' data set

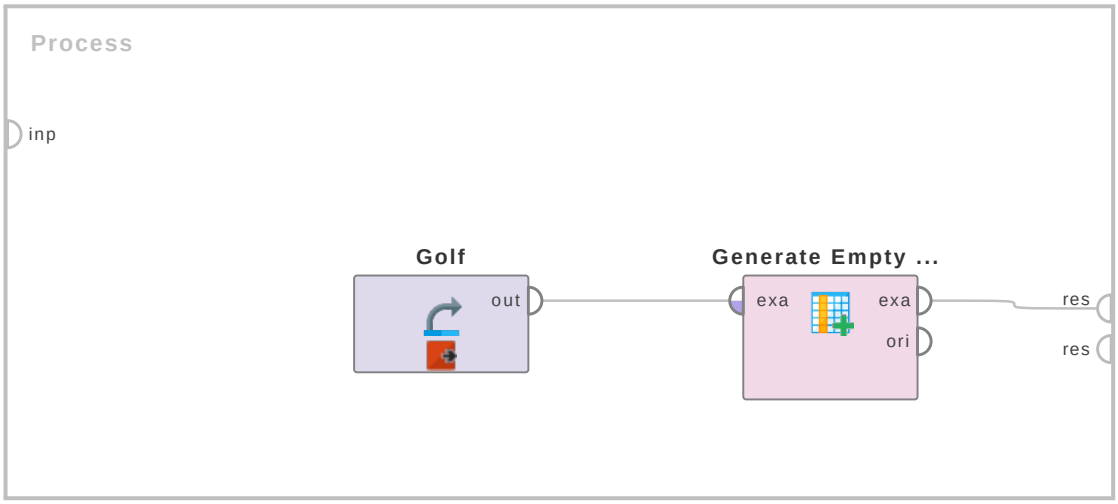
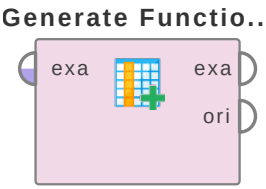



Figure 2.38: Tutorial process 'Adding an empty attribute to the 'Golf' data set'.

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the input ExampleSet. As you can see that the 'Golf' data set has 5 attributes: Play, Outlook, Temperature, Humidity and Wind. The Generate Empty Attribute operator is applied on the 'Golf' data set. The name parameter is set to 'name' and the value type parameter is set to 'nominal'. When the process execution is complete, you can see the ExampleSet in the Results Workspace. This ExampleSet has one attribute more than the 'Golf' data set. The name and type of the attribute are the same as specified in the parameters of the Generate Empty Attribute operator. Please note that all values of this new attribute are missing. These values can be filled by using operators like the Set Data operator. Please note that the created empty attribute is independent of the input ExampleSet, if you want to generate an attribute from the existing attributes of the input ExampleSet you can use the Generate Attributes operator.

# Generate Function Set




This is an attribute generation operator which generates new attributes by applying a set of selected functions on all attributes.

## Description

This operator applies a set of selected functions on all attributes of the input ExampleSet for generating new attributes. Numerous functions are available including summation, difference, multiplication, division, reciprocal, square root, power, sine, cosine, tangent, arc tangent, absolute, minimum, maximum, ceiling, floor and round. It is important to note that the functions with two arguments will be applied on all possible pairs. For example suppose an ExampleSet with three numerical attributes A, B and C. If the summation function is applied on this ExampleSet then three new attributes will be generated with values A+B, A+C and B+C. Similarly non-commutative functions will be applied on all possible permutations. This is a useful attribute generation operator but if it does not meet your requirements please try the Generate Attributes operator which is a very powerful attribute generation operator.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set output *(exa)*** New attributes are created by application of the selected functions and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**keep all *(boolean)*** This parameter indicates if the original attributes should be kept.

**use plus *(boolean)*** This parameter indicates if the summation function should be applied for generation of new attributes.

**use diff *(boolean)*** This parameter indicates if the difference function should be applied for generation of new attributes.

**use mult *(boolean)*** This parameter indicates if the multiplication function should be applied for generation of new attributes.

**use div *(boolean)*** This parameter indicates if the division function should be applied for generation of new attributes.

**use reciprocals** ***(boolean)*** This parameter indicates if the reciprocal function should be applied for generation of new attributes.

**use square roots** ***(boolean)*** This parameter indicates if the square roots function should be applied for generation of new attributes.

**use power functions** ***(boolean)*** This parameter indicates if the power function should be applied for generation of new attributes.

**use sin** ***(boolean)*** This parameter indicates if the sine function should be applied for generation of new attributes.

**use cos** ***(boolean)*** This parameter indicates if the cosine function should be applied for generation of new attributes.

**use tan** ***(boolean)*** This parameter indicates if the tangent function should be applied for generation of new attributes.

**use atan** ***(boolean)*** This parameter indicates if the arc tangent function should be applied for generation of new attributes.

**use exp** ***(boolean)*** This parameter indicates if the exponential function should be applied for generation of new attributes.

**use log** ***(boolean)*** This parameter indicates if the logarithmic function should be applied for generation of new attributes.

**use absolute values** ***(boolean)*** This parameter indicates if the absolute values function should be applied for generation of new attributes.

**use min** ***(boolean)*** This parameter indicates if the minimum values function should be applied for generation of new attributes.

**use max** ***(boolean)*** This parameter indicates if the maximum values function should be applied for generation of new attributes.

**use ceil** ***(boolean)*** This parameter indicates if the ceiling function should be applied for generation of new attributes.

**use floor** ***(boolean)*** This parameter indicates if the floor function should be applied for generation of new attributes.

**use rounded** ***(boolean)*** This parameter indicates if the round function should be applied for generation of new attributes.

## Tutorial Processes

### Using the power function for attribute generation

The 'Iris' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 4 real attributes. The Generate Function Set operator is applied on this ExampleSet for generation of new attributes, only the Power function is used. It is not a commutative function e.g. 2 raised to power 3 is not the same as 3 raised to power 2. The non-commutative functions are applied for all possible permutations. As there are 4 original attributes, there are 16 (i.e. 4 x 4) possible permutations. Thus 16 new attributes are created as a result of this operator. The resultant ExampleSet can be seen in the Results Workspace. As the keep all parameter was set to true, the original attributes of the ExampleSet are not discarded.

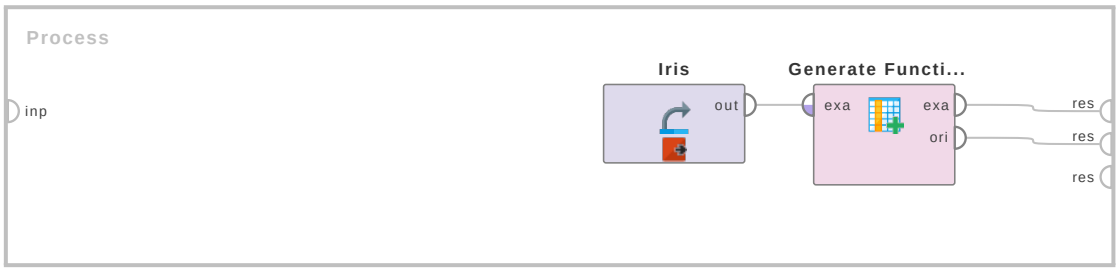
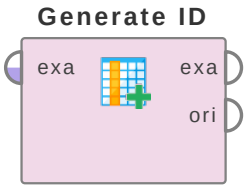



Figure 2.39: Tutorial process 'Using the power function for attribute generation'.

# Generate ID




This operator adds a new attribute with *id* role in the input ExampleSet. Each example in the input ExampleSet is tagged with an incremented *id*. If an attribute with *id* role already exists, it is overridden by the new *id* attribute.

## Description

This operator adds a new attribute with *id* role in the input ExampleSet. It assigns a unique *id* to each example. This operator is usually used to uniquely identify each example. Each example in the input ExampleSet is tagged with an incremented *id*. The number from where the *ids* start can be controlled by the *offset* parameter. Numerical and integer *ids* can be assigned. If an attribute with *id* role already exists in the input ExampleSet, it is overridden by the new *id* attribute.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**example set output *(exa)*** The ExampleSet with an *id* attribute is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**create nominal ids *(boolean)*** This parameter indicates if nominal *ids* should be created instead of integer *ids*. By default this parameter is not checked, thus integer *ids* are created by default. Nominal *ids* are of the format id_1, id_2, id_3 and so on.

**offset *(integer)*** This is an expert parameter. It is used if you want to start *id* from a number other than 1. This parameter is used to set the offset value. It is 0 by default, thus *ids* start from 1 by default.

## Tutorial Processes

### Overriding the id attribute of the 'Iris' data set

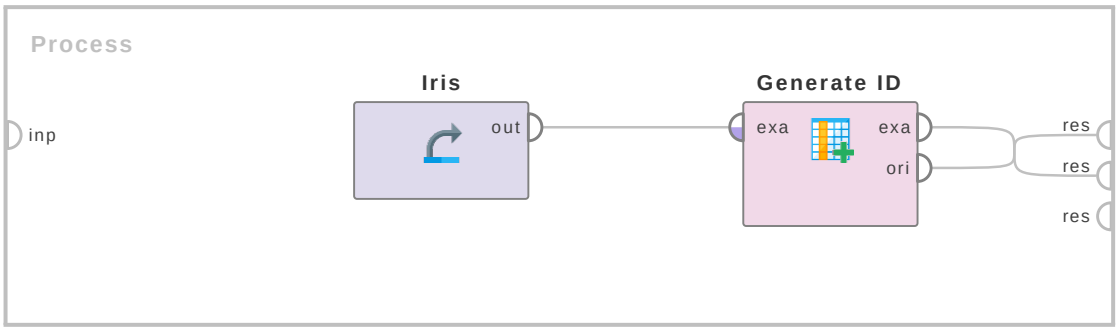
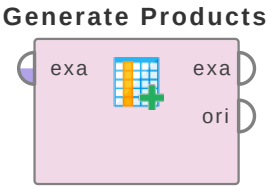



Figure 2.40: Tutorial process 'Overriding the id attribute of the 'Iris' data set'.

The 'Iris' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it. All parameters are used with default values. The 'Iris' data set already has an id attribute. The old id attribute is overridden when the Generate ID operator is applied on it. Run the process and you can see the ExampleSet with the new id attribute. The type of this new attribute is integer. Set the create nominal ids parameter to true and run the process again, you will see that the ids are in nominal form now (i.e. id_1, id_2 and so on). The offset parameter is set to 0 that is why the ids start from 1. Now set the offset parameter to 10 and run the process again. Now you can see that ids start from 11.

# Generate Products




This operator generates new attributes by taking the products of the specified attributes.

## Description

The Generate Products operator generates new attributes by taking the products of the specified attributes. The attributes are specified through the *first attribute name* and *second attribute name* parameters. For example, if the *first attribute name* parameter has attributes A and B, and the *second attribute name* has attributes C and D. Then four attributes A*C, A*D, B*C and B*D will be generated by this operator. These attributes will have products of the corresponding attribute values.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is output of the Generate Data operator in the attached Example Process.

## Output Ports

**example set output *(exa)*** New attributes are generated by taking the products of the specified attributes and the resultant ExampleSet is returned through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**first attribute name *(string)*** This parameter specifies the name(s) of the first attribute(s) to be multiplied. Attribute names can be specified through regular expressions.

**second attribute name *(string)*** This parameter specifies the name(s) of the second attribute(s) to be multiplied. Attribute names can be specified through regular expressions.

## Tutorial Processes

### Generating products of attributes

The Generate Data operator provides a sample ExampleSet. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has four real attributes i.e. att1, att2, att3 and att4. The Generate Products operator is applied on this ExampleSet. Have a look at the parameters of this operator. The att1 and att2 attributes are selected through the first attribute name parameter. The att3 and att4 attributes are selected through the second attribute name parameter. The resultant ExampleSet can be seen in the Results Workspace. You can see that this ExampleSet has new attributes that have been generated by multiplying the first attribute name attributes with the second attribute name attributes.

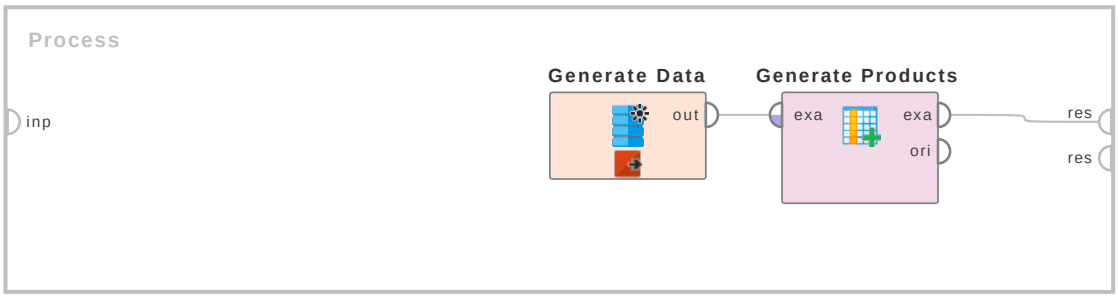
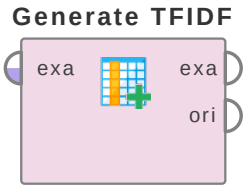



Figure 2.41: Tutorial process 'Generating products of attributes'.

# Generate TFIDF




This operator performs a TF-IDF filtering of the given ExampleSet. TF-IDF is a numerical statistic which reflects how important a word is to a document.

## Description

The Generate TFIDF operator generates TF-IDF values from the given ExampleSet The ExampleSet must contain either the binary occurrences (which will be normalized during calculation of the term frequency TF) or it should already contain the calculated term frequency values (in this case no normalization will be done). This behavior can be selected using the *calculate term frequencies* parameter.

The TF-IDF (term frequency–inverse document frequency) is a numerical statistic which reflects how important a word is to a document in a collection or corpus. It is often used as a weighting factor in information retrieval and text mining. The tf-idf value increases proportionally to the number of times a word appears in the document, but is offset by the frequency of the word in the corpus, which helps to control for the fact that some words are generally more common than others.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is output of the Read CSV operator in the attached Example Process.

## Output Ports

**example set output *(exa)*** The TF-IDF is calculated and the resultant ExampleSet is returned through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**calculate term frequencies *(boolean)*** This parameter indicates if term frequency values should be generated. This parameter must be set to true if the input data is given as simple occurrence counts.

## Tutorial Processes

### Introduction to the Generate TFIDF operator

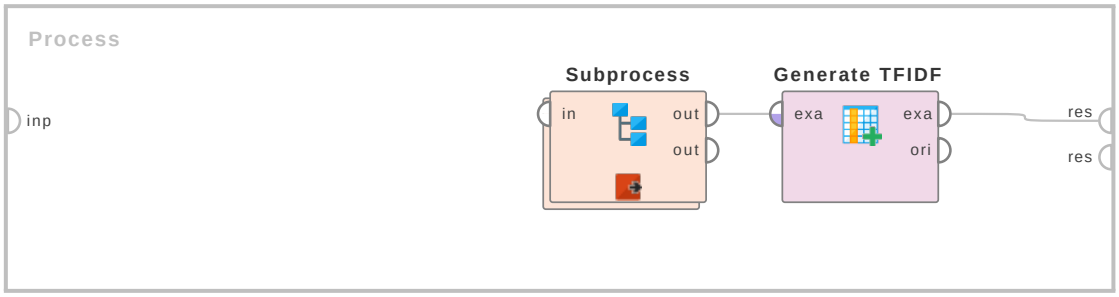
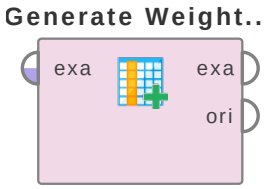



Figure 2.42: Tutorial process 'Introduction to the Generate TFIDF operator'.

This Example Process starts with a Subprocesses operator which generates a sample ExampleSet. A breakpoint is inserted here so that you can have a look at the ExampleSet. This is a very simple ExampleSet. It has a text attribute which has different words. There are three integer attributes named Doc1, Doc2 and Doc3 that have the count of the corresponding words in these documents. The Generate TFIDF operator is applied on this ExampleSet to calculate the TFIDF. The resultant ExampleSet can be seen in the Results Workspace.

# Generate Weight (Stratification)




This operator distributes the specified weight over all the examples, such that weights sum up equally per label.

## Description

The Generate Weight (Stratification) operator divides the weight specified through the *total weight* parameter among all the examples. While dividing the weight, this operator makes sure that the sum of example weights of all label values is same. This often improves the representativeness of the label values. Please study the attached Example Process for better understanding.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**example set output *(exa)*** The examples are assigned weights and the resultant ExampleSet is returned through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**total weight *(real)*** This parameter specifies the total weight that should be distributed over all the examples.

## Tutorial Processes

### Assigning weights such that weights sum up equally per label

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the label of this ExampleSet has two possible values i.e. 'yes' and 'no'. The Generate Weight (Stratification) operator is applied on this ExampleSet for weight assignment. The total weight parameter is set to 10. This operator assigns weight to examples such that:

The sum of all weights is equal to the total weight.The sum of weights is equal per label.

Thus in this process, the sum of all weights should be 10 and the weight sum of examples with label 'no' should be equal to the weight sum of examples with label 'yes'. You can verify this by viewing the resultant ExampleSet in the Results Workspace.

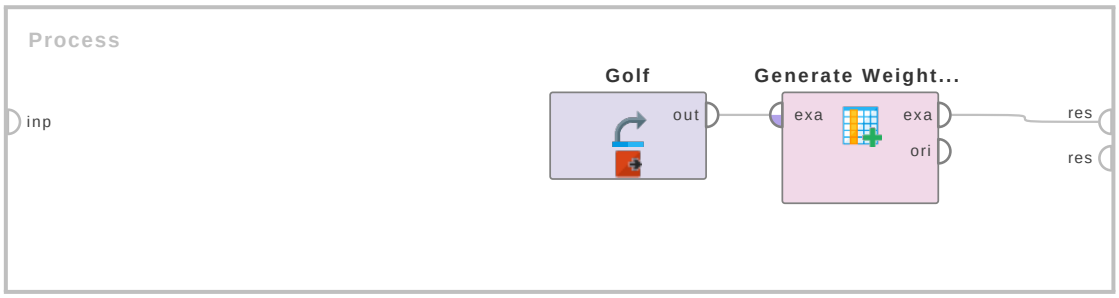
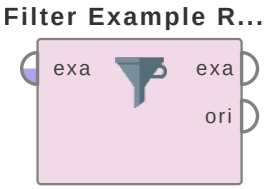



Figure 2.43: Tutorial process 'Assigning weights such that weights sum up equally per label'.

# 2.2 Examples

## 2.2.1 Filter

### Filter Example Range




This operator selects which examples (i.e. rows) of an ExampleSet should be kept and which examples should be removed. Examples within the specified index range are kept, remaining examples are removed.

#### Description

This operator takes an ExampleSet as input and returns a new ExampleSet including only those examples that are within the specified index range. Lower and upper bound of index range are specified using *first example* and *last example* parameters. This operator may reduce the number of examples in an ExampleSet but it has no effect on the number of attributes. The Select Attributes operator is used to select required attributes.

If you want to filter examples by options other than index range, you may use the Filter Examples operator. It takes an ExampleSet as input and returns a new ExampleSet including only those examples that satisfy the specified condition. Several predefined conditions are provided; users can select any of them. Users can also define their own conditions to filter examples. The Filter Examples operator is frequently used to filter examples that have (or do not have) missing values. It is also frequently used to filter examples with correct or wrong predictions (usually after testing a learnt model).

#### Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**example set output *(exa)*** A new ExampleSet including only the examples that are within the specified index range is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**first example *(integer)*** This parameter is used to set the lower bound of the index range. The *last example* parameter is used to set the upper bound of the index range. Examples within this index range are delivered to the output port. Examples outside this index range are discarded.

**last example *(integer)*** This parameter is used to set the upper bound of the index range. The *first example* parameter is used to set the lower bound of the index range. Examples within this index range are delivered to the output port. Examples outside this index range are discarded.

**invert filter *(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected examples are removed and previously removed examples are selected. In other words it inverts the index range. For example if the *first example* parameter is set to 1 and the *last example*parameter is set to 10. Then the output port will deliver an ExampleSet with all examples other than the first ten examples.

## Tutorial Processes

### Filtering examples using the invert filter parameter

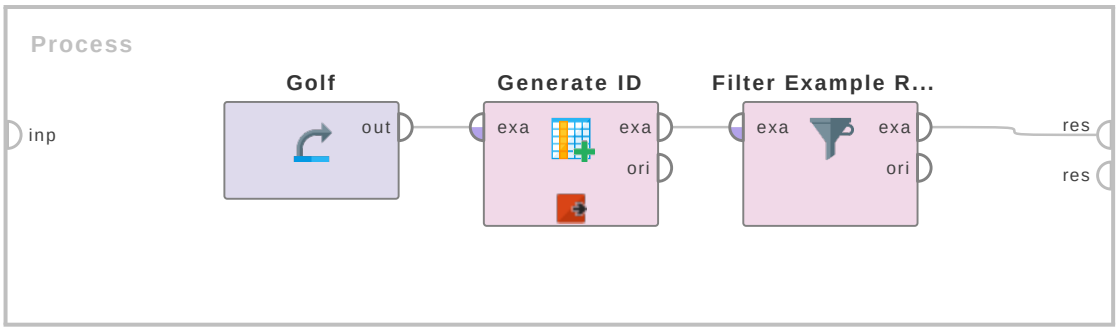
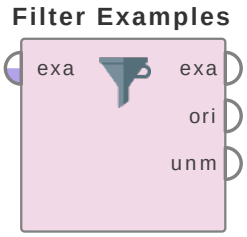



Figure 2.44: Tutorial process 'Filtering examples using the invert filter parameter'.

The 'Golf' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it with offset set to 0. Thus all examples are assigned unique ids from 1 to 14. This is done so that examples can be distinguished easily. A breakpoint is inserted here so that you can have a look at the data set before application of the Filter Example Range operator. In the Filter Example Range operator the first example parameter is set to 5 and the last example parameter is set to 10. The invert filter parameter is also set to true. Thus all examples other than examples in

index range 5 to 10 are delivered through the output port. You can clearly identify rows through their ids. Rows with IDs from 1 to 4 and from 11 to 14 make it to the output port.

# Filter Examples




This Operator selects which Examples of an ExampleSet are kept and which Examples are removed.

## Description

The Operator returns those Examples that match the given condition. The conditions are defined by the user. Several pre-defined conditions also exist as advanced options.

## Differentiation

- **Select Attributes**

  Filter Examples may reduce the number of Examples in an ExampleSet but it has no effect on the number of Attributes. The Select Attributes Operator is used to select Attributes.

  See page 199 for details.

- **Filter Example Range**

  The Filter Example Range Operator can be used to select Examples that lie in the specified index range (i.e. number of lines).

  See page 238 for details.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet on which the defined filter will be applied.

## Output Ports

**example set output *(exa)*** This port outputs an ExampleSet with only the Examples, that satisfied the specified condition.

**original *(ori)*** The ExampleSet, that was given as input is passed through without changes.

**unmatched example set *(unm)*** An ExampleSet including only those Examples, that did not meet the specified condition.

## Parameters

**filters** This is the default parameter for defining filter condtions via 'Add Filters...' dialog window. It is also available when the 'custom_filters' *condition class* is selected. This option allows the definition of a custom filter condition. A condition consists of an Attribute, a comparison function and a value to match. More conditions can be added by the "Add Entry" button. Several filters can be joined either by "Match all" or "Match any".

**condition class** This parameter only appears when the 'show advanced parameters' is activated. Otherwise the default selection 'custom_filters' is shown. Various predefined conditions are available for filtering Examples. Examples matching the selected condition are passed to the output port, others are removed. The available conditions are:

- **all** If this option is selected, no Examples are removed.
- **correct_predictions** If this option is selected, only those Examples are returned, where the prediction is correct. This option requires that the ExampleSet has two Attributes with the special roles Label and Prediction. Then only those Examples are returned, where the values of the label Attribute and prediction Attribute are the same.
- **wrong predictions** This option is the same as the correct_predictions option, but with the reversed result. Those Examples are matched, where the prediction is not the same as the label.
- **no_missing_attributes** If this option is selected, only those Examples are matched that have no missing values. Missing values or null values are shown as '?' in RapidMiner.
- **missing_attributes** If this option is selected, only those Examples are matched, that have missing values. Missing values are shown as '?' in RapidMiner.
- **no_missing_labels** If this option is selected, only those Examples are matched, that don't have a missing value in the special Attribute with the label role.
- **missing_labels** If this option is selected, only those Examples are matched, that have a missing value in the special Attribute with the label role.
- **attribute_value_filter** If this option is selected, a condition can be entered in the field of the *parameter string*. The option is like the default filter. The details are explained below in the *parameter string* description. The benefit of declaring a filter statement as a string is an increased flexibility using macros.
- **expression** With this option, expressions can be defined that offer more functions to write matching condition. How expressions can be used to filter Examples is explained below in the *parameter expression* description.
- **custom_filters** This option is the same as the default filters parameter.

**parameter string** This parameter is available when the parameter 'attribute_value_filter' is selected as condition class. The condition format is an Attribute name, followed by a comparison function and a value to match. For numerical Attributes the comparison functions are >, <, <=, >= and = while the matching value has to be a number. Nominal Attributes can be compared by = and != with an arbitrary string, which can also include a regular expression. Multiple conditions can be linked by a logical OR (||) or a logical AND (&&). Missing values can be written as '?' for numerical attributes and as '\?' for nominal attributes.

**parameter expression** This parameter is available when the parameter 'expression' is selected as *condition class*. Expressions can be entered as String or with the expression builder dialog. The expression needs to evaluate to a boolean value and should include one or more Attributes. This option is useful to build more complex matching conditions. For example including mathematical calculations or text manipulation.

**invert filter** If this parameter is set to true the selected condition is inverted. All matching Examples are removed from the output and Examples that don't match the condition are in the output.

## Tutorial Processes

### Filter Examples using custom filters

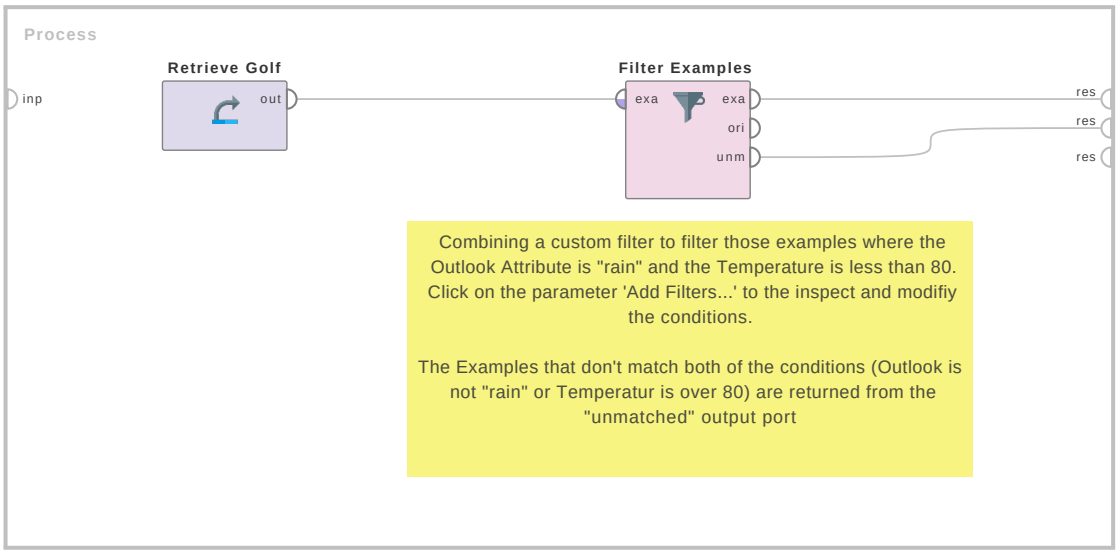



Figure 2.45: Tutorial process ‘Filter Examples using custom filters’.

This tutorial Process shows how to filter the Golf ExampleSet by combining two filter conditions.

### Filter Examples using attribute value filter

This tutorial Process uses the advanced parameter attribute_filter to define a condition string. It uses the regular expression .*n.* to filter all Examples where the value of the Outlook Attribute contains the letter ‘n’. The second statement filters the Examples where the Temperature Attribute is greater than 70. Both conditions are combined with an OR statement ( || )

### Filter Examples using expression

This tutorial Process loads the Titanic data and uses an expression string to select all passengers whose name contains “Miss.” and who are younger than 30, as well as all male passengers.

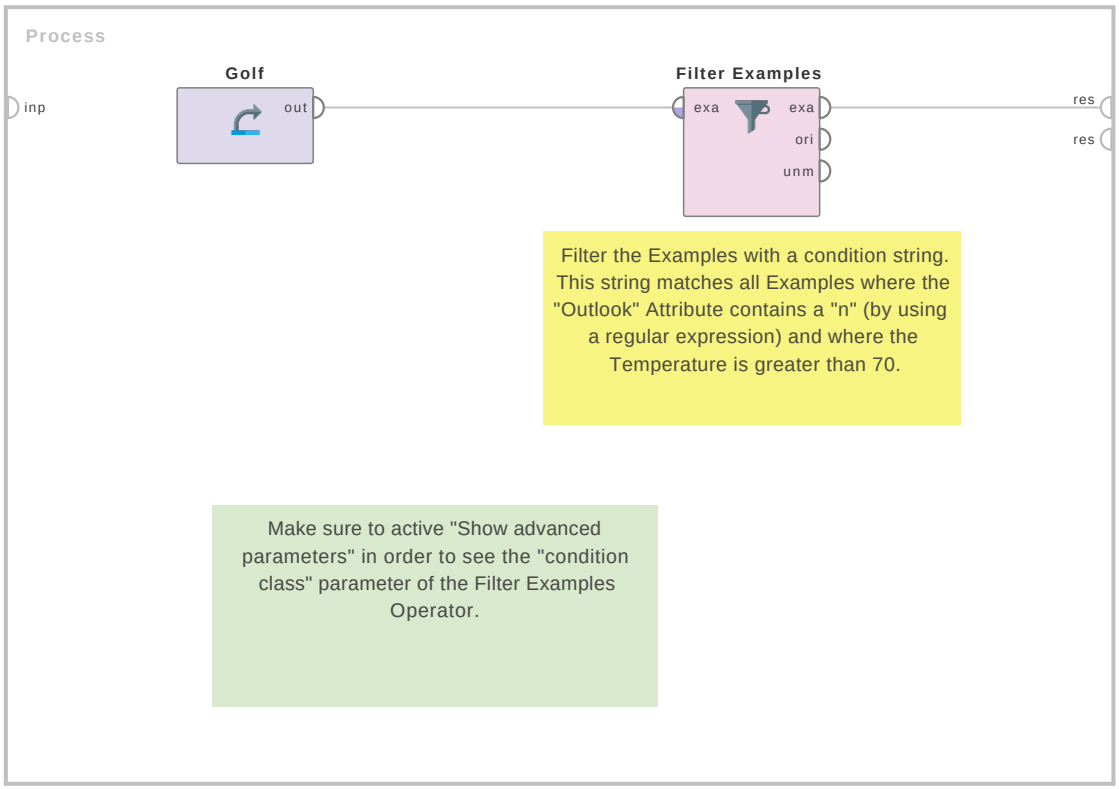
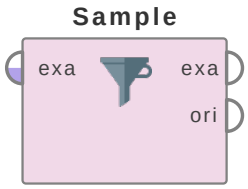



Figure 2.46: Tutorial process 'Filter Examples using attribute value filter'.

## 2.2.2 Sampling

### Sample




This operator creates a sample from an ExampleSet by selecting examples randomly. The size of a sample can be specified on absolute, relative and probability basis.

#### Description

This operator is similar to the Filter Examples operator in principle that it takes an ExampleSet as input and delivers a subset of the ExampleSet as output. The difference is this that the Filter Examples operator filters examples on the basis of specified conditions. But the Sample operator focuses on the number of examples and class distribution in the resultant sample. Moreover, the samples are generated randomly. The number of examples in the sample can be specified on absolute, relative or probability basis depending on the setting of the *sample* parameter. The class distribution of the sample can be controlled by the *balance data* parameter.

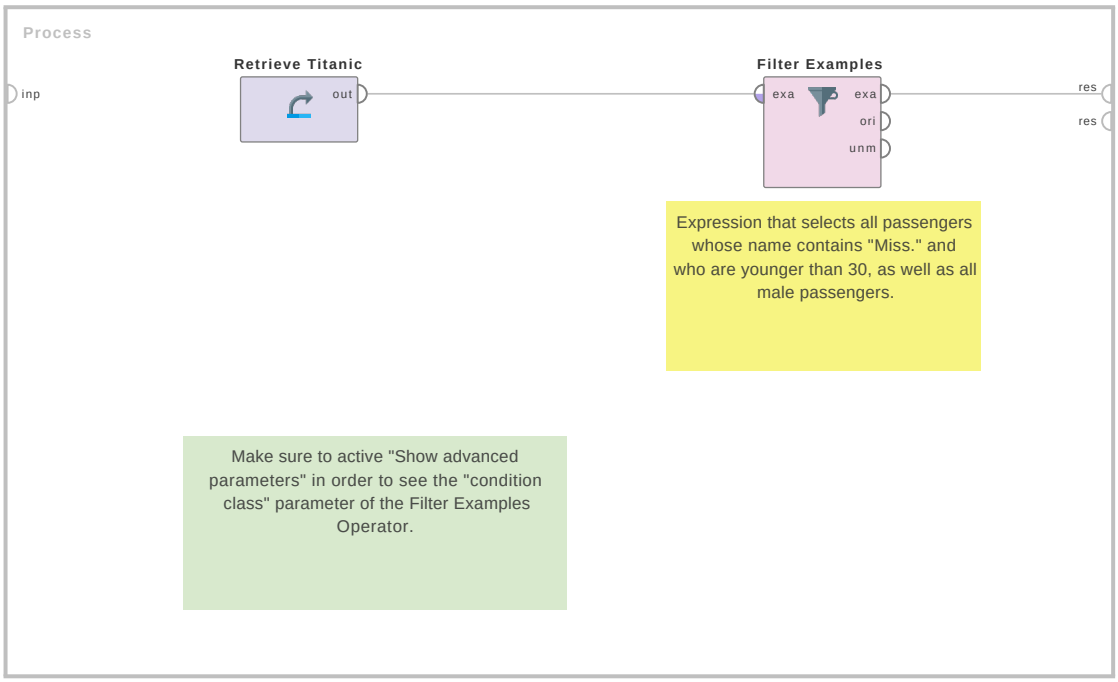



Figure 2.47: Tutorial process 'Filter Examples using expression'.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**example set output *(exa)*** A randomized sample of the input ExampleSet is output of this port.

**original *(ori)*** ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**sample *(selection)*** This parameter determines how the amount of data is specified.

- **absolute** If the *sample* parameter is set to 'absolute' the sample is created of an exactly specified number of examples. The required number of examples is specified in the *sample size* parameter.
- **relative** If the *sample* parameter is set to 'relative' the sample is created as a fraction of the total number of examples in the input ExampleSet. The required ratio of examples is specified in the *sample ratio* parameter.

- **probability** If the *sample* parameter is set to 'probability' the sample is created of probability basis. The required probability is specified in the *sample probability* parameter.

**balance data *(boolean)*** You can set this parameter to true if you need to sample differently for examples of a certain class. If this parameter is set to true, *sample size*, *sample ratio* and *sample probability* parameters are replaced by *sample size per class*, *sample ratio per class* and *sample probability per class* parameters respectively. These parameters allow you to specify different sample sizes for different values of the label attribute.

**sample size *(integer)*** This parameter specifies the exact number of examples which should be sampled. This parameter is only available when the *sample* parameter is set to 'absolute' and the *balance data* parameter is not set to true.

**sample ratio *(real)*** This parameter specifies the fraction of examples which should be sampled. This parameter is only available when the *sample* parameter is set to 'relative' and the *balance data* parameter is not set to true.

**sample probability *(real)*** This parameter specifies the sample probability for each example. This parameter is only available when the *sample* parameter is set to 'probability' and the *balance data* parameter is not set to true.

**sample size per class** This parameter specifies the absolute sample size per class. This parameter is only available when the *sample* parameter is set to 'absolute' and the *balance data* parameter is set to true.

**sample ratio per class** This parameter specifies the fraction of examples per class. This parameter is only available when the *sample* parameter is set to 'relative' and the *balance data* parameter is set to true.

**sample probability per class** This parameter specifies the probability of examples per class. This parameter is only available when the *sample* parameter is set to 'probability' and the *balance data* parameter is set to true.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomizing examples of the sample. Using the same value of *local random seed* will produce the same sample. Changing the value of this parameter changes the way the examples are randomized, thus the sample will have a different set of examples.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Sampling the Ripley-Set data set

The 'Ripley-Set' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it so that the examples can be identified uniquely. A breakpoint is inserted at this stage so that you can see the ExampleSet before the Sample operator is applied. You can see that there are 250 examples with two possible classes: 0 and 1. 125 examples have class 0 and 125 examples have class 1. Now, the Sample operator is applied on the ExampleSet. The sample parameter is set to 'relative'. The balance data parameter is set to true. The sample ratio per class parameter specifies two ratios. Class 0 is assigned ratio 0.2. Thus, of all the examples where label attribute is 0 only 20 percent will be selected. There were 125 examples with class 0, so 25 (i.e. 20% of

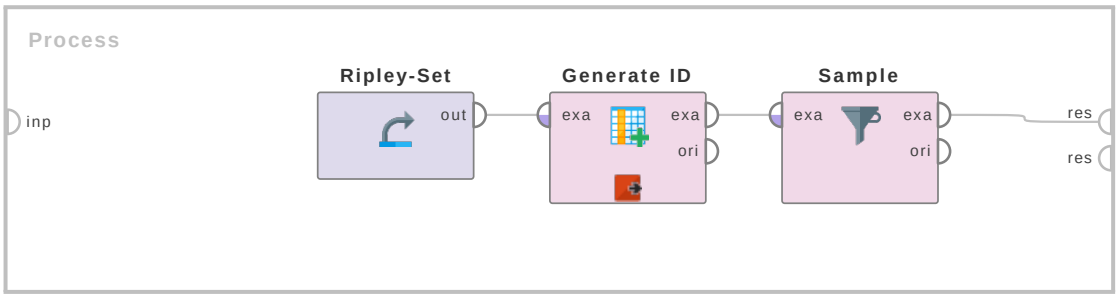
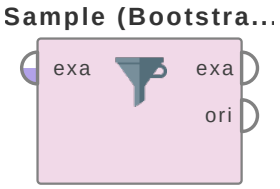



Figure 2.48: Tutorial process 'Sampling the Ripley-Set data set'.

125) examples will be selected. Class 1 is assigned ratio 1. Thus, of all the examples where label attribute is 1, 100 percent will be selected. There were 125 examples with class 1, so all 125 (i.e. 100% of 125) examples will be selected. Run the process and you can verify the results. Also note that the examples are taken randomly. The randomization can be changed by changing the local random seed parameter.

# Sample (Bootstrapping)




This operator creates a bootstrapped sample from an ExampleSet. Bootstrapped sampling uses sampling with replacement, thus the sample may not have all unique examples. The size of the sample can be specified on absolute and relative basis.

## Description

This operator is different from other sampling operators because it uses sampling with replacement. In sampling with replacement, at every step all examples have equal probability of being selected. Once an example has been selected for the sample, it remains candidate for selection and it can be selected again in any other coming steps. Thus a sample with replacement can have the same example multiple number of times. More importantly, a sample with replacement can be used to generate a sample that is greater in size than the original ExampleSet. The number of examples in the sample can be specified on absolute or relative basis depending on the setting of the *sample* parameter.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is output of the Generate ID operator in the attached Example Process.

## Output Ports

**example set output *(exa)*** A bootstrapped sample of the input ExampleSet is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**sample *(selection)*** This parameter determines how the amount of data is specified.

- **absolute** If the *sample* parameter is set to 'absolute' the sample is created of the exactly specified number of examples. The required number of examples is specified in the *sample size* parameter.
- **relative** If the *sample* parameter is set to 'relative' the sample is created as a fraction of the total number of examples in the input ExampleSet. The required ratio of examples is specified in the *sample ratio* parameter.

**sample size *(integer)*** This parameter specifies the exact number of examples which should be sampled. This parameter is only available when the *sample* parameter is set to 'absolute'.

**sample ratio *(real)*** This parameter specifies the fraction of examples which should be sampled. This parameter is only available when the *sample* parameter is set to 'relative'.

**use weights *(boolean)*** If set to true, example weights will be considered during the bootstrapping if such weights are present.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomizing examples of the sample. Using the same value of the *local random seed* will produce the same sample. Changing the value of this parameter changes the way the examples are randomized, thus the sample will have a different set of examples.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Bootstrapped Sampling of the Golf data set

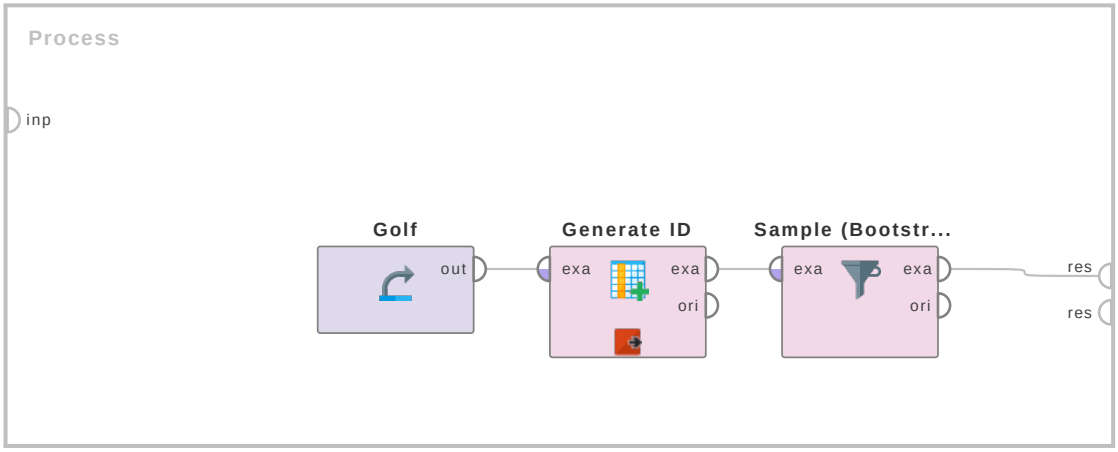
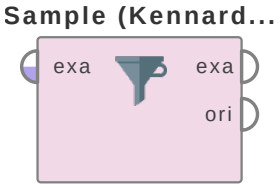



Figure 2.49: Tutorial process 'Bootstrapped Sampling of the Golf data set'.

The 'Golf' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it to create an id attribute with ids starting from 1. This is done so that the examples can be identified uniquely, otherwise the id attribute was not necessary here. A breakpoint is inserted here so that you can view the ExampleSet before the application of the Sample (Bootstrapping) operator. As you can see, the ExampleSet has 14 examples. The Sample (Bootstrapping) operator is applied on the ExampleSet. The sample parameter is set to 'absolute' and the sample size parameter is set to 140. Thus a sample 10 times in size of the original ExampleSet is generated. Instead of repeating each example of the input ExampleSet 10 times, examples are selected randomly. You can verify this by seeing the results of this process in the Results Workspace.

# Sample (Kennard-Stone)




This operator creates a sample from the given ExampleSet by using the Kennard-Stone algorithm. The size of the sample can be specified on absolute and relative basis.

## Description

The Sample (Kennard-Stone) operator performs a Kennard-Stone Sampling. This sampling algorithm works as follows:

- Find the two most separated points in the ExampleSet.
- For each candidate point, find the smallest distance to any already selected object.
- Select the point which has the largest of these smallest distances.

This algorithm always gives the same result because the two starting points are always the same. This implementation reduces the number of iterations by holding a list with candidates of the largest smallest distances. Please note that the number of examples in the sample may not be exactly the same as specified because of the way this algorithm works.

The sampling operators are similar to the Filter Examples operator in principle that they take an ExampleSet as input and delivers a subset of the ExampleSet as output. The difference is this that the Filter Examples operator filters examples on the basis of specified conditions. But the Sample operators focus on the number of examples and class distribution in the resultant sample. Moreover, the samples are generated randomly. The number of examples in the sample can be specified on absolute and relative basis depending on the setting of the *sample* parameter.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**example set output *(exa)*** The Kennard-Stone algorithm is applied and the resultant sample of the input ExampleSet is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**sample *(selection)*** This parameter determines how the amount of data is specified.

- **absolute** If the *sample* parameter is set to 'absolute' then the sample is created of an exactly specified number of examples. The required number of examples is specified in the *sample size* parameter.

- **relative** If the *sample* parameter is set to 'relative' then the sample is created as a fraction of the total number of examples in the input ExampleSet. The required ratio of examples is specified in the *sample ratio* parameter.

**sample size *(integer)*** This parameter specifies the exact number of examples which should be sampled. This parameter is only available when the *sample* parameter is set to 'absolute'.

**sample ratio *(real)*** This parameter specifies the fraction of examples which should be sampled. This parameter is only available when the *sample* parameter is set to 'relative'.

## Tutorial Processes

### Kennard-Stone sampling of the Iris data set

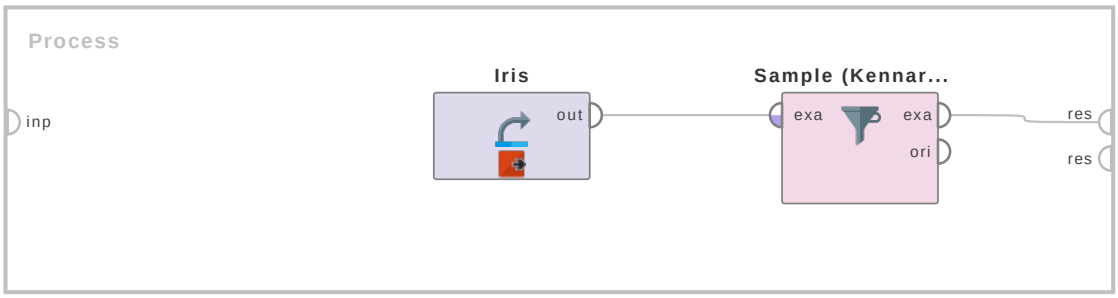
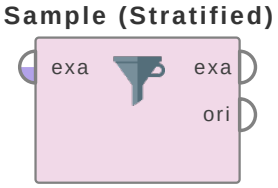



Figure 2.50: Tutorial process 'Kennard-Stone sampling of the Iris data set'.

The 'Iris' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the ExampleSet. You can see that the ExampleSet has 150 examples. The Sample (Kennard-Stone) operator is applied on the ExampleSet. The sample parameter is set to 'absolute' and the sample size parameter is set to 15. Thus the resultant sample will have only 15 examples. The resultant ExampleSet with 15 examples can be seen in the Results Workspace.

# Sample (Stratified)




This operator creates a stratified sample from an ExampleSet. Stratified sampling builds random subsets and ensures that the class distribution in the subsets is the same as in the whole ExampleSet. This operator cannot be applied on data sets without a label or with a numerical label. The size of the sample can be specified on absolute and relative basis.

## Description

The stratified sampling builds random subsets and ensures that the class distribution in the subsets is the same as in the whole ExampleSet. For example in the case of a binominal classification, Stratified sampling builds random subsets such that each subset contains roughly the same proportions of the two values of class *labels*.

When there are different classes in an ExampleSet, it is sometimes advantageous to sample each class independently. Stratification is the process of dividing examples of the ExampleSet into homogeneous subgroups (i.e. classes) before sampling. The subgroups should be mutually exclusive i.e. every examples in the ExampleSet must be assigned to only one subgroup (or class). The subgroups should also be collectively exhaustive i.e. no example can be excluded. Then random sampling is applied within each subgroup. This often improves the representativeness of the sample by reducing the sampling error.

A real-world example of using stratified sampling would be for a political survey. If the respondents needed to reflect the diversity of the population, the researcher would specifically seek to include participants of various minority groups such as race or religion, based on their proportionality to the total population as mentioned above. A stratified survey could thus claim to be more representative of the population than a survey of simple random sampling or systematic sampling.

In contrast to the simple sampling operator (the Sample operator), this operator performs a stratified sampling of the data sets with nominal label attributes, i.e. the class distributions remains (almost) the same after sampling. Hence, this operator cannot be applied on data sets without a label or with a numerical label. In these cases a simple sampling without stratification should be performed through the Sample operator.

This operator is similar to the Filter Examples operator in principle that it takes an ExampleSet as input and delivers a subset of the ExampleSet as output. The difference is this that the Filter Examples operator filters examples on the basis of specified conditions. But the Sample operator focuses on the number of examples and class distribution in the resultant sample. Moreover, the samples are generated randomly. The number of examples in the sample can be specified on absolute and relative basis depending on the setting of the *sample* parameter.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is output of the Filter Examples operator in the attached Example Process.

## Output Ports

**example set output *(exa)*** A randomized sample of the input ExampleSet is output of this port. The class distributions of the sample is (almost) the same as the class distribution of the complete ExampleSet.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**sample *(selection)*** This parameter determines how the amount of data is specified.

- **absolute** If the *sample* parameter is set to 'absolute' then the sample is created of an exactly specified number of examples. The required number of examples is specified in the *sample size* parameter.
- **relative** If the *sample* parameter is set to 'relative' then the sample is created as a fraction of the total number of examples in the input ExampleSet. The required ratio of examples is specified in the *sample ratio* parameter.

**sample size *(integer)*** This parameter specifies the exact number of examples which should be sampled. This parameter is only available when the *sample* parameter is set to 'absolute'.

**sample ratio *(real)*** This parameter specifies the fraction of examples which should be sampled. This parameter is only available when the *sample* parameter is set to 'relative'.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomizing examples of the sample. Using the same value of *local random seed* will produce the same sample. Changing the value of this parameter changes the way the examples are randomized, thus sample will have a different set of examples.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Stratified Sampling of the Golf data set

The 'Golf' data set is loaded using the Retrieve operator. The Filter Example Range operator is applied on it to select the first 10 examples. This is done to simplify the Example Process otherwise the filtering was not necessary here. A breakpoint is inserted here so that you can view the ExampleSet before the application of the Sample (Stratified) operator. As you can see, the ExampleSet has 10 examples. 6 examples (i.e. 60%) belong to class 'yes' and 4 examples (i.e. 40%) belong to class 'no'. The Sample (Stratified) operator is applied on the ExampleSet. The sample parameter is set to 'absolute' and the sample size parameter is set to 5. Thus the resultant sample will have only 5 examples. The sample will have the same class distribution as the class distribution of the input ExampleSet i.e. 60% examples with class 'yes' and 40% examples with class 'no'. You can verify this by viewing the results of this process. 3 out of 5 examples (i.e. 60%) have class 'yes' and 2 out of 5 examples (i.e. 40%) have class 'no'.

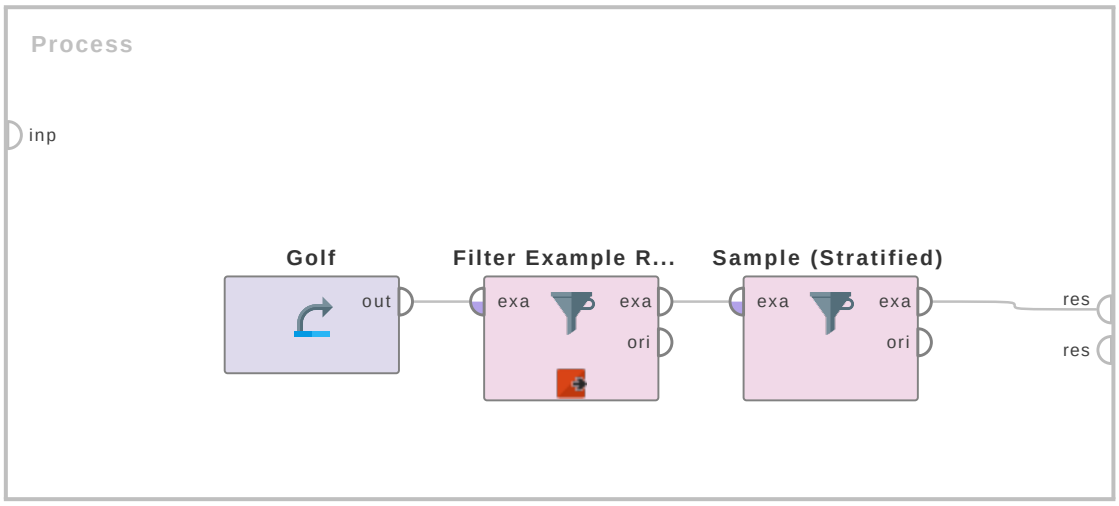
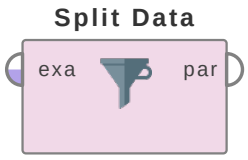



Figure 2.51: Tutorial process 'Stratified Sampling of the Golf data set'.

# Split Data




This operator produces the desired number of subsets of the given ExampleSet. The ExampleSet is partitioned into subsets according to the specified relative sizes.

## Description

The Split Data operator takes an ExampleSet as its input and delivers the subsets of that ExampleSet through its output ports. The number of subsets (or partitions) and the relative size of each partition are specified through the *partitions* parameter. The sum of the ratio of all partitions should be 1. The *sampling type* parameter decides how the examples should be shuffled in the resultant partitions. For more information about this operator please study the parameters section of this description. This operator is different from other sampling and filtering operators in the sense that it is capable of delivering multiple partitions of the given ExampleSet.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

## Output Ports

**partition *(par)*** This operator can have multiple number of partition ports. The number of useful partition ports depends on the number of partitions (or subsets) this operator is configured to produce. The *partitions* parameter is used for specifying the desired number of partitions.

## Parameters

**partitions** ***(enumeration)*** This is the most important parameter of this operator. It specifies the number of partitions and the relative ratio of each partition. The user just requires to specify the ratio of all partitions. The number of required partitions is not explicitly specified by the user because it is calculated automatically by the number of ratios specified in this parameter. The ratios should be between 0 and 1. The sum of all ratios should be 1. For better understanding of this parameter please study the attached Example Process.

**sampling type** ***(selection)*** The Split Data operator can use several types of sampling for building the subsets. Following options are available:

- **Linear sampling** Linear sampling simply divides the ExampleSet into partitions without changing the order of the examples i.e. subsets with consecutive examples are created.
- **Shuffled sampling** Shuffled sampling builds random subsets of the ExampleSet. Examples are chosen randomly for making subsets.
- **Stratified sampling** Stratified sampling builds random subsets and ensures that the class distribution in the subsets is the same as in the whole ExampleSet. For example in the case of a binominal classification, Stratified sampling builds random subsets such that each subset contains roughly the same proportions of the two values of the class *labels*.
- **Automatic** Uses stratified sampling if the label is nominal, shuffled sampling otherwise.

**use local random seed** ***(boolean)*** Indicates if a *local random seed* should be used for randomizing examples of a subset. Using the same value of *local random seed* will produce the same subsets. Changing the value of this parameter changes the way examples are randomized, thus subsets will have a different set of examples. This parameter is only available if Shuffled or Stratified sampling is selected. It is not available for Linear sampling because it requires no randomization, examples are selected in sequence.

**local random seed** ***(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Creating partitions of the Golf data set using the Split Data operator

The 'Golf' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it so the examples can be identified uniquely. A breakpoint is inserted here so the ExampleSet can be seen before the application of the Split Data operator. It can be seen that the ExampleSet has 14 examples which can be uniquely identified by the id attribute. The examples have ids from 1 to 14. The Split Data operator is applied next. The sampling type parameter is set to 'linear sampling'. The partitions parameter is configured to produce two partitions with ratios 0.8 and 0.2 respectively. The partitions can be seen in the Results Workspace. The number of examples in each partition is calculated by this formula:

(Total number of examples) / (sum of ratios) * ratio of this partition

If the answer is a decimal number it is rounded off. The number of examples in each partition turns out to be: (14) / (0.8 + 0.2) * (0.8) = 11.2 which is rounded off to 11(14) / (0.8 + 0.2) * (0.2) = 2.8 which is rounded off to 3

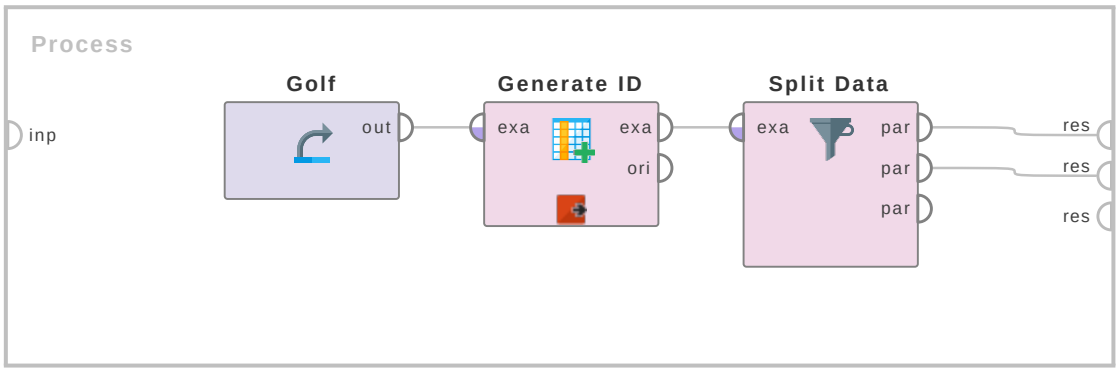
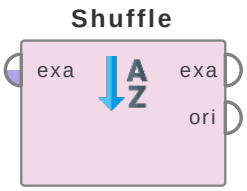



Figure 2.52: Tutorial process 'Creating partitions of the Golf data set using the Split Data operator'.

It is a good practice to adjust ratios such that the sum of ratios is 1. But this operator also works if the sum of ratios is lower than or greater than 1. For example if two partitions are created with ratios 1.0 and 0.4. The resultant partitions would be calculated as follows: (14) / (1.0 + 0.4) * (1.0) = 10(14) / (1.0 + 0.4) * (0.4) = 4

## 2.2.3 Sort

### Shuffle




This operator creates a new, shuffled ExampleSet from the given ExampleSet by making a new copy of the given ExampleSet in the main memory.

#### Description

The Shuffle operator creates a new, shuffled ExampleSet by making a new copy of the given ExampleSet in the main memory. Please note that the system may run out of memory, if the ExampleSet is too large. The *local random seed* parameter can be used for randomizing the shuffling process.

#### Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

#### Output Ports

**example set output *(exa)*** The shuffled ExampleSet is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

#### Parameters

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of the *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

#### Tutorial Processes

##### Shuffling the Iris data set

The 'Iris' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has an id attribute. The ExampleSet is sorted in ascending order of this attribute. The Shuffle operator is applied on this ExampleSet to randomize the order of its examples. The resultant shuffled ExampleSet can be seen in the Results Workspace.

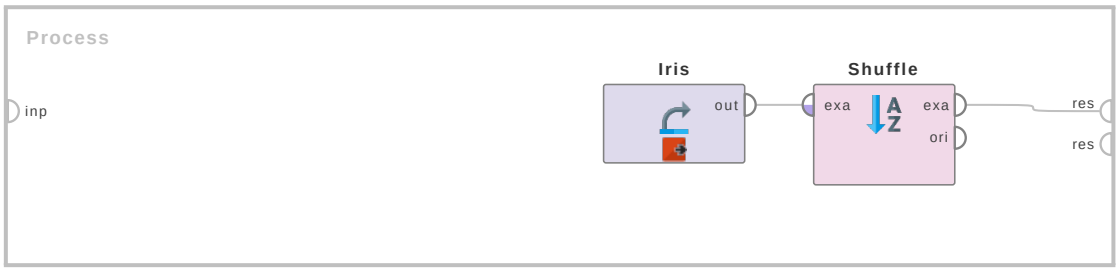
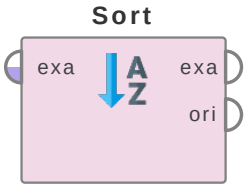



Figure 2.53: Tutorial process 'Shuffling the Iris data set'.

## Sort




This operator sorts the input ExampleSet in ascending or descending order according to a single attribute.

### Description

This operator sorts the ExampleSet provided at the input port. The complete data set is sorted according to a single attribute. This attribute is specified using the *attribute name* parameter. Sorting is done in increasing or decreasing direction depending on the setting of the *sorting direction* parameter.

### Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

### Output Ports

**example set output *(exa)*** The sorted ExampleSet is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

### Parameters

**attribute name *(string)*** This parameter is used to specify the attribute which should be used for sorting the ExampleSet.

**sorting direction** This parameter indicates the direction of the sorting. The ExampleSet can be sorted in increasing(ascending) or decreasing(descending) order.

## Tutorial Processes

### Sorting the Golf data set according to Temperature

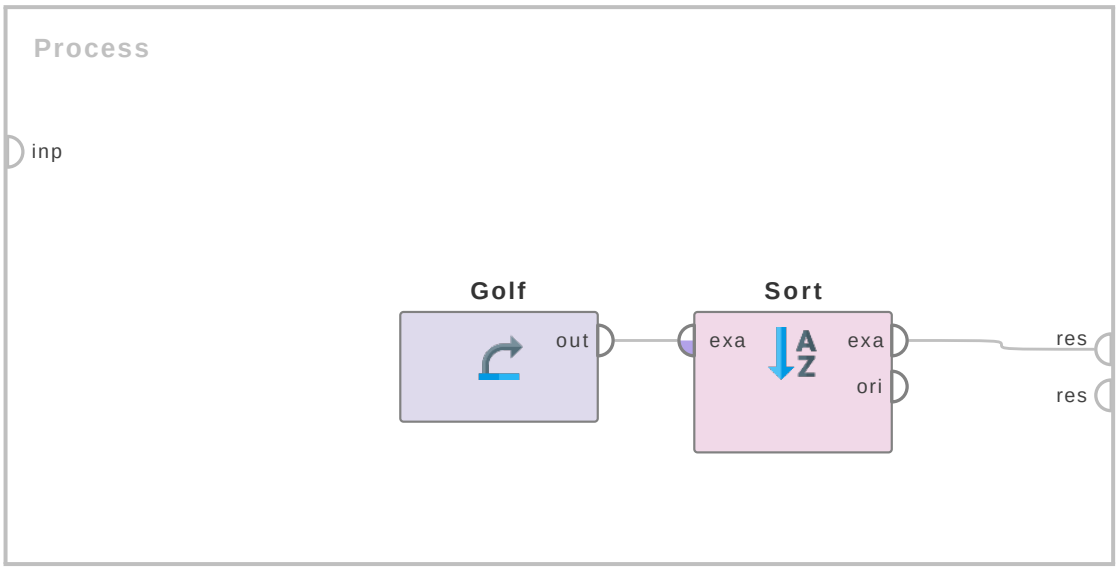



Figure 2.54: Tutorial process 'Sorting the Golf data set according to Temperature'.

The 'Golf' data set is loaded using the Retrieve operator. The Sort operator is applied on it. The attribute name parameter is set to 'Temperature'. The sort direction parameter is set to 'increasing'. Thus the 'Golf' data set is sorted in ascending order of the 'Temperature' attribute. The example with the smallest value of the 'Temperature' attribute becomes the first example and the example with the largest value of the 'Temperature' attribute becomes the last example of the ExampleSet.

### Sorting on multiple attributes

This Example Process shows how two Sort operators can be used to sort an ExampleSet on two attributes. The 'Golf' data set is loaded using the Retrieve operator. The Sort operator is applied on it. The attribute name parameter is set to 'Temperature'. The sort direction parameter is set to 'increasing'. Then another Sort operator is applied on it. The attribute name parameter is set to 'Humidity' this time. The sort direction parameter is set to 'increasing'. Thus the 'Golf' data set is sorted in ascending order of the 'Humidity' attribute. The example with smallest value of the 'Humidity' attribute becomes the first example and the example with the largest value of the 'Humidity' attribute becomes the last example of the ExampleSet. If some examples have the same value of the 'Humidity' attribute, they are sorted using the 'Temperature' attribute. Where examples have same value of the 'Humidity' attribute then the examples with smaller value of the 'Temperature' attribute precede the examples with higher value of the 'Temperature' attribute. This can be seen in the Results Workspace.

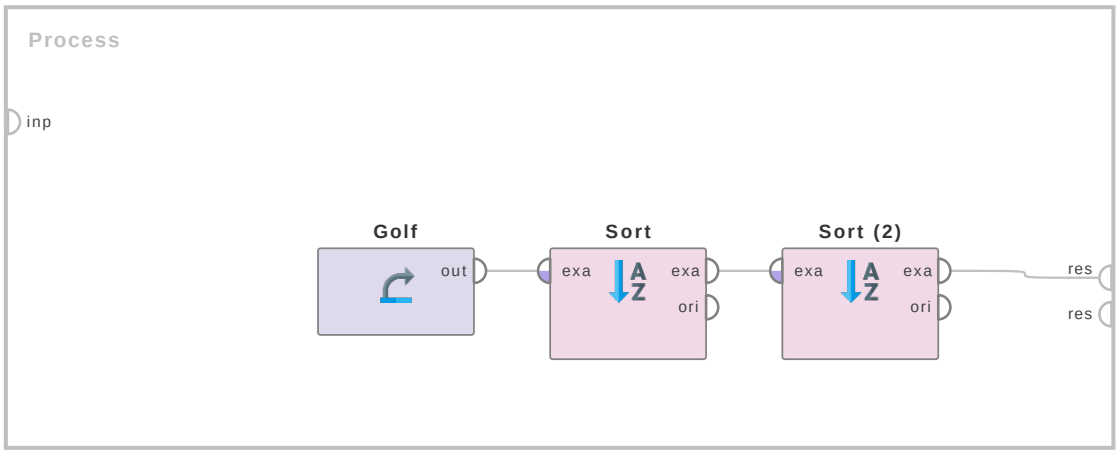
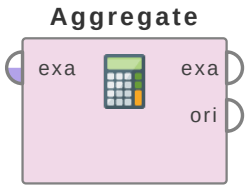



Figure 2.55: Tutorial process 'Sorting on multiple attributes'.

# 2.3 Table

## 2.3.1 Grouping

### Aggregate




This operator performs the aggregation functions known from SQL. This operator provides a lot of functionalities in the same format as provided by the SQL aggregation functions. SQL aggregation functions and GROUP BY and HAVING clauses can be imitated using this operator.

#### Description

The Aggregate operator creates a new ExampleSet from the input ExampleSet showing the results of the selected aggregation functions. Many aggregation functions are supported including SUM, COUNT, MIN, MAX, AVERAGE and many other similar functions known from SQL. The functionality of the GROUP BY clause of SQL can be imitated by using the *group by attributes* parameter. You need to have a basic understanding of the GROUP BY clause of SQL for understanding the use of this parameter because it works exactly the same way. If you want to imitate the known HAVING clause from SQL, you can do that by applying the Filter Examples operator after the Aggregation operator. This operator imitates aggregation functions of SQL. It focuses on obtaining summary information, such as averages and counts etc. It can group examples in an ExampleSet into smaller sets and apply aggregation functions on those sets. Please study the attached Example Process for better understanding of this operator.

#### Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Filter Examples operator in the attached Example Process. Output of other operators can also be used as

input.

## Output Ports

**example set *(exa)*** The ExampleSet generated after applying the specified aggregation functions is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**use default aggregation *(boolean)*** This parameter allows you to define the default aggregation for selected attributes. A number of parameters become available if this parameter is set to true. These parameters allow you to select the attributes and corresponding default aggregation function.

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose all examples satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of *parameter* attribute if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list. Attributes can be shifted to the right list, which is the list of selected attributes.

**regular expression** ***(string)*** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression** ***(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter *(except regular expression*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will not be selected even if they match the previously mentioned type i.e. *value type* parameter's value.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be not be selected even if they match the previously mentioned block type i.e. *block type* parameter's value.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '$>$', '$=$' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection *(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**default aggregation function** This parameter is only available when the *use default aggregation* parameter is set to true. It is used for specifying the default aggregation function for the selected attributes.

**aggregation attributes** This parameter is one of the most important parameters of the operator. It allows you to select attributes and the aggregation function to apply on them. Many aggregation functions are available including count, average, minimum, maximum variance and many more.

**group by attributes** This operator can group examples of the input ExampleSet into smaller groups using this parameter. The aggregation functions are applied on these groups. This parameter allows the Aggregate operator to replicate the functionality of the known GROUP BY clause of SQL. From version 6.0.3 on the operator will cause an error if a given attribute can't be found in the example set.

**count all combinations *(boolean)*** This parameter indicates if all possible combinations of the values of the group by attributes are counted, even if they don't occur. All possible combinations may result in a huge number so handle this parameter carefully.

**only distinct *(boolean)*** This parameter indicates if only examples with distinct values for the aggregation attribute should be used for the calculation of the aggregation function.

**ignore missings *(boolean)*** This parameter indicates if missing values should be ignored and aggregation functions should be applied only on existing values. If this parameter is not set to true then the aggregated value will be a missing value in the presence of missing values in the selected attribute.

## Tutorial Processes

### Imitating an SQL aggregation query using the Aggregate operator

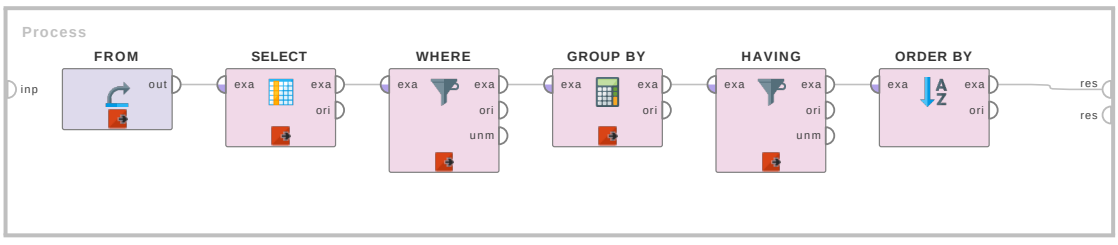



Figure 2.56: Tutorial process 'Imitating an SQL aggregation query using the Aggregate operator'.

This Example Process discusses an arbitrary scenario. Then describes how this scenario could be handled using SQL aggregation functions. Then the SQL's solution is imitated in RapidMiner. The Aggregate operator plays a key role in this process.

Let us assume a scenario where we want to apply certain aggregation functions on the Golf data set. We don't want to include examples where the Outlook attribute has the value 'overcast'. We group the remaining examples of the 'Golf' data set by values of the Play and Wind attributes. We wish to find the average Temperature and average Humidity for these groups. Once these averages have been calculated, we want to see only those examples where the average Temperature is above 71. Lastly, we want to see the results in ascending order of the average Temperature.

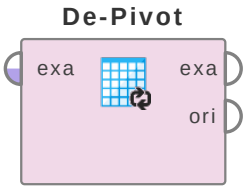
This problem can be solved by the following SQL query:
SELECT Play, Wind, AVG (Temperature), AVG (Humidity)
FROM Golf
WHERE Outlook NOT LIKE 'overcast'
GROUP BY Play, Wind
HAVING AVG (Temperature)>71
ORDER BY AVG (Temperature)

The SELECT clause selects the attributes to be displayed. The FROM clause specifies the data set. The WHERE clause pre-excludes the examples where the Outlook attribute has value 'overcast'. The GROUP BY clause groups the data set according to the specified attributes. The HAVING clause filters the results after the aggregation functions have been applied. Finally the ORDER BY clause sorts the results in ascending order of the Temperature averages.

Here is how this scenario can be tackled using RapidMiner. First of all the Retrieve operator is used for loading the 'Golf' data set. This is similar to the FROM clause. Then the Select Attributes operator is applied on it to select the required attributes. This works a little different from the SQL query. If we select only the Play and Wind attributes as in the query, then the coming operators cannot be applied. Thus we select all attributes for now. You will see later that the attribute set will be reduced automatically, thus the Select Attributes operator is not really required here. Then the Filter Examples operator is applied to pre-exclude examples where the Outlook attribute has the value 'overcast'. This is similar to the WHERE clause of SQL. Then the Aggregate operator is applied on the remaining examples. The Aggregate operator performs a number of tasks here. Firstly, it specifies the aggregation functions using the aggregation attributes parameter. We need averages of the Temperature and Humidity attribute; this is specified using the aggregation attributes parameter. Secondly, we do not want the averages of the entire data set. We want the averages by groups, grouped by the Play and Wind attribute values. These groups are specified using the group by attributes parameter of the Aggregate operator. Thirdly, required attributes are automatically filtered by this operator. Only those attributes appear in the resultant data set that have been specified in the Aggregate operator. Next, we are interested only in those examples where the average Temperature is greater than 71. This condition can be applied using the Filter Examples operator. This step is similar to the HAVING clause. Lastly we want the results to be sorted. The Sort operator is used to do the required sorting. This step is very similar to the ORDER BY clause. Breakpoints are inserted after every operator in the Example Process so that you can understand the part played by each operator.

## 2.3.2 Rotation

### De-Pivot




This operator transforms the ExampleSet by converting the examples of the selected attributes (usually attributes that measure the same characteristic) into examples of a single attribute.

#### Description

This operator is usually used when your ExampleSet has multiple attributes that measure the same characteristic (may be at different time intervals) and you want to merge these observations into a single attribute without loss of information. If the original ExampleSet has *n* examples and *k* attributes that measure the same characteristic, after application of this operator the ExampleSet will have *k* x *n* examples. The *k* attributes will be combined into one attribute. This attribute will have *n* examples of each of the *k* attributes. This can be easily understood by studying the attached Example Process.

In other words, this operator converts an ExampleSet by dividing examples which consist of multiple observations (at different times) into multiple examples, where each example covers one point in time. An index attribute is added in the ExampleSet, which denotes the actual point in time the example belongs to after the transformation.

The *keep missings* parameter specifies whether an example should be kept, even if it has missing values for all series at a certain point in time. The *create nominal index* parameter is only applicable if only one time series per example exists. Instead of using a numeric index, then the names of the attributes representing the single time points are used as index attribute values.

#### Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input.

#### Output Ports

**example set output *(exa)*** The selected attributes are converted into examples of a new attribute and the resultant ExampleSet is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

#### Parameters

**attribute name *(list)*** This parameter maps a number of source attributes onto result attributes. The attribute name parameter is used for specifying the group of attributes that you want to combine and the name of the new attribute. The attributes of a group are selected through a regular expression. There can be numerous groups with each group having multiple attributes.

**index attribute *(string)*** This parameter specifies the name of the newly created index attribute. The index attribute is used for differentiating between examples of different attributes of a group after the transformation.

**create nominal index *(boolean)*** The *create nominal index* parameter is only applicable if only one time series per example exists. Instead of using a numeric index, then the names of the attributes representing the single time points are used as index attribute values.

**keep missings *(boolean)*** The *keep missings* parameter specifies whether an example should be kept, even if it has missing values for all series at a certain point in time.

## Tutorial Processes

### Merging multiple attributes that measure the same characteristic into a single attribute

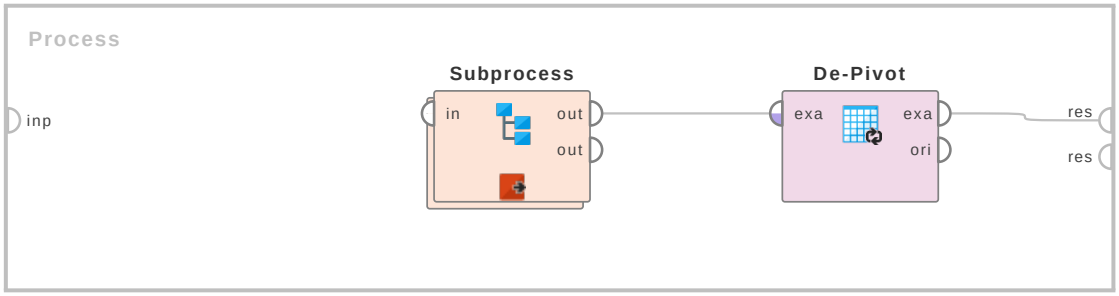



Figure 2.57: Tutorial process 'Merging multiple attributes that measure the same characteristic into a single attribute'.

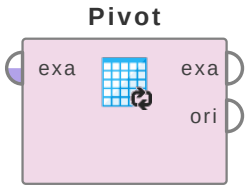
This process starts with the Subprocess operator which delivers an ExampleSet. The subprocess is used for creating a sample ExampleSet therefore it is not important to understand what is going on inside the subprocess. A breakpoint is inserted after the subprocess so that you can have a look at the ExampleSet. You can see that the ExampleSet has 14 examples and it has two attributes i.e. 'Morning' and 'Evening'. These attributes measure the temperature of an area in morning and evening respectively. We want to convert these attributes into a single attribute but we still want to be able to differentiate between morning and evening temperatures.

The De-Pivot operator is applied on this ExampleSet to perform this task. The attribute name parameter is used for specifying the group of attributes that you want to combine and the name of the new attribute. The attributes of a group are selected through a regular expression. There can be numerous groups with each group having multiple attributes. In our case, there is only one group which has all the attributes of the ExampleSet (i.e. both 'Morning' and 'Evening' attributes). The new attribute is named 'Temperatures' and the regular expression: ' .* ' is used for selecting all the attributes of the ExampleSet. The index attribute is used for differentiating between examples of different attributes of a group after transformation. The name of the index attribute is set to 'Time'. The create nominal index parameter is also set to true so that the resultant ExampleSet is more self-explanatory.

Execute the process and have a look at the resultant ExampleSet. You can see that there are 28 examples in this ExampleSet. The original ExampleSet had 14 examples, and 2 attributes were grouped, therefore the resultant ExampleSet has 28 (i.e. 14 x 2) examples. There are 14 examples from the Morning attribute and 14 examples of the Evening attribute in the 'Temperatures'

attribute. The ‘Time’ attribute explains whether an example measures morning or evening temperature.

# Pivot




This operator rotates an ExampleSet by grouping multiple examples of same groups to single examples.

## Description

The Pivot operator rotates the given ExampleSet by grouping multiple examples of same groups to single examples. The *group attribute* parameter specifies the grouping attribute (i.e. the attribute which identifies examples belonging to the groups). The resultant ExampleSet has *n* examples where *n* is the number of unique values of the group attribute. The *index attribute* parameter specifies the attribute whose values are used to identify the examples inside the groups. The values of this attribute are used to name the group attributes which are created during the pivoting. Typically the values of such an attribute capture subgroups or dates. The resultant ExampleSet has *m* regular attributes in addition to the group attribute where *m* is the number of unique values of the index attribute. If the given ExampleSet contains example weights (i.e. an attribute with weight role), these weights may be aggregated in each group to maintain the weightings among groups. This description can be easily understood by studying the attached Example Process.

## Differentiation

- **Transpose** The Transpose operator simply rotates the given ExampleSet (i.e. interchanges rows and columns) but the Pivot operator provides additional options like grouping and handling weights. See page 271 for details.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process.

## Output Ports

**example set output *(exa)*** The ExampleSet produced after pivoting is the output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without any modifications to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**group attribute *(string)*** This parameter specifies the grouping attribute (i.e. the attribute which identifies examples belonging to the groups). The resultant ExampleSet has *n* examples where *n* is the number of unique values of the group attribute.

**index attribute** ***(string)*** This parameter specifies the attribute whose values are used to identify the examples inside the groups. The values of this attribute are used to name the group attributes which are created during the pivoting. Typically the values of such an attribute capture subgroups or dates. The resultant ExampleSet has *m* regular attributes in addition to the group attribute where *m* is the number of unique values of the index attribute.

**consider weights** ***(boolean)*** This parameter specifies whether attribute weights (if any) should be kept and aggregated or ignored.

**weight aggregation** ***(selection)*** This parameter is only available when the *consider weights* parameter is set to true. It specifies how example weights should be aggregated in the groups. It has the following options: average, variance, standard_deviation, count, minimum, maximum, sum, mode, median, product.

**skip constant attributes** ***(boolean)*** This parameter specifies if the attributes should be skipped if their value never changes within a group.

**data management** ***(selection)*** This is an expert parameter. There are different options, users can choose any of them

## Related Documents

- **Transpose** (page 271)

## Tutorial Processes

### Introduction to the Pivot operator

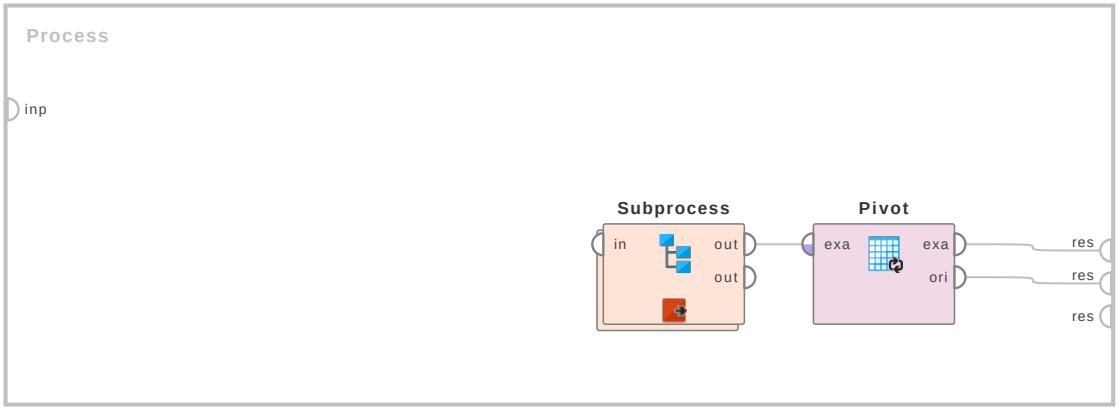



Figure 2.58: Tutorial process 'Introduction to the Pivot operator'.

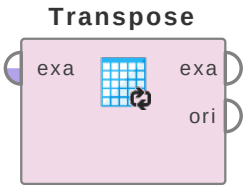
This Example Process starts with the Subprocess operator. There is a sequence of operators in this Subprocess operator that produces an ExampleSet that is easy to understand. A breakpoint is inserted after the Subprocess operator to show this ExampleSet. The Pivot operator is applied on this ExampleSet. The group attribute and index attribute parameters are set to 'group_attribute' and 'index_attribute' respectively. The consider weights parameter is set to true and the weight aggregation parameter is set to 'sum'. The group_attribute has 5 possible values therefore the pivoted ExampleSet has 5 examples i.e. one for each possible value of the

group_attribute. The index_attribute has 5 possible values therefore the pivoted ExampleSet has 5 regular attributes (in addition to the group_attribute). Here is an explanation of values of the first example of the pivoted ExampleSet. The remaining examples also follow the same idea.

The value of the group_attribute of the first example of the pivoted ExampleSet is 'group0', therefore all values of this example will be derived from all examples of the input ExampleSet where the group_attribute had the value 'group0'. The ids of examples with 'group0' in the input ExampleSet are 12, 16, 19 and 20. In the coming explanation these examples will be called group0 examples for simplicity.

The value of the weight_attribute attribute of the pivoted ExampleSet is 11. It is the sum of weights of group0 examples i.e. 4 + 4 + 0 + 3 = 11. The weights were added because the weight aggregation parameter is set to 'sum'. The value of the value_attribute_index0 attribute of the pivoted ExampleSet is 4. Only two examples (id 12 and 16) of the group0 examples had 'index0' in index_attribute. The value of the latter of these examples (id 16) is selected i.e. 4 is selected. The value of the value_attribute_index1 attribute of the pivoted ExampleSet is 1. Only one example (id 19) of the group0 examples had 'index1' in index_attribute. Therefore its value (i.e. 1) is selected. The value of the value_attribute_index2 attribute of the pivoted ExampleSet is undefined because no example of the group0 examples had 'index2' in index_attribute. Therefore its value is missing in the pivoted ExampleSet. The value of the value_attribute_index3 attribute of the pivoted ExampleSet is 3. Only one example (id 20) of the group0 examples had 'index3' in index_attribute. Therefore its value (i.e. 3) is selected. The value of the value_attribute_index4 attribute of the pivoted ExampleSet is undefined because no example of the group0 examples had 'index4' in index_attribute. Therefore its value is missing in the pivoted ExampleSet.

# Transpose




This operator transposes the input ExampleSet i.e. the current rows become columns of the output ExampleSet and current columns become rows of the output ExampleSet. This operator works very similar to the well known transpose operation for matrices.

## Description

This operator transposes the input ExampleSet i.e. the current rows become the columns of the output ExampleSet and the current columns become the rows of the output ExampleSet. In other words every example or row becomes a column with attribute values and each attribute column becomes an example row. This operator works very similar to the well known transpose operation for matrices. The transpose of a transpose of a matrix is same as the original matrix, but the same rule cannot be applied here because the types of the original ExampleSet and the transpose of the transpose of an ExampleSet may be different.

If an *id* attribute is part of the input ExampleSet, the *ids* will become the names of the new attributes. The names of the old attributes will be transformed into the *id* values of a new *id* attribute. All other new attributes will have *regular* role after the transformation. You can use the Set Role operator after the transpose operator to assign roles to new attributes.

If all old attributes have the same value type, all new attributes will have the same value type. If at least one nominal attribute is part of the input ExampleSet, the type of all new attributes will be nominal. If the old attribute values were all mixed numbers, the type of all new attributes will be real. This operator produces a copy of the data in the main memory. Therefore, it should not be used on very large data sets.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**example set output *(exa)*** The transpose of the input ExampleSet is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Tutorial Processes

### Different scenarios of Transpose

There are four different cases in this Example Process:

Case 1: The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet before application of the Transpose operator. You can see that the 'Golf' data set has no id attribute. The types of attributes are different including attributes of nominal type. Press the Run button to continue. Now the Transpose operator is applied on the 'Golf' data set. A breakpoint is inserted here so that you can see the ExampleSet

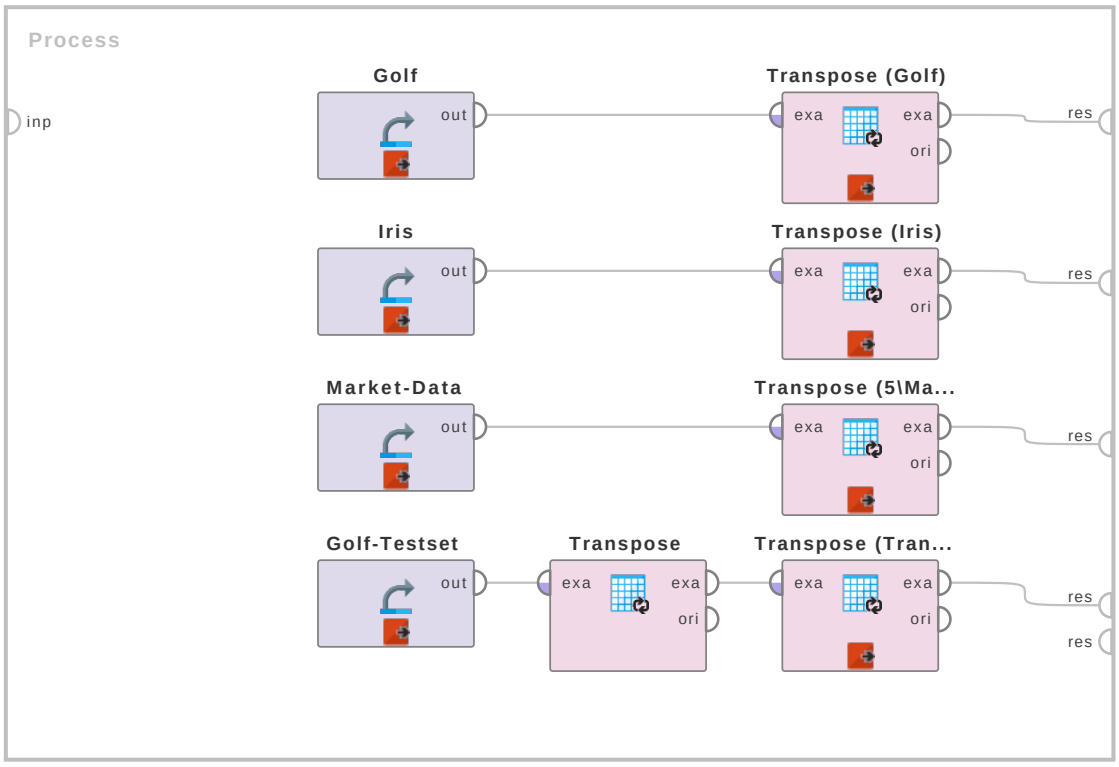



Figure 2.59: Tutorial process 'Different scenarios of Transpose'.

after the application of the Transpose operator. Here you can see that a new attribute with id role has been created. The values of the new id attribute are the names of the old attributes. New attributes are named in a general format like 'att_1', 'att_2' etc because the input ExampleSet had no id attribute. The type of all new attributes is nominal because there were attributes with different types including at least one nominal attribute in the input ExampleSet.

Case 2: The 'Iris' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet before application of the Transpose operator. You can see that the 'Iris' data set has an id attribute. The types of attributes are different including attributes of nominal type. Press the Run button to continue. Now the Transpose operator is applied on the 'Iris' data set. A breakpoint is inserted here so that you can see the ExampleSet after the application of the Transpose operator. Here you can see that a new attribute with id role has been created. The values of the new id attribute are the names of the old attributes. The ids of the old ExampleSet become names of the new attributes. The type of all new attributes is nominal because there were attributes with different types including at least one nominal attribute in the input ExampleSet.

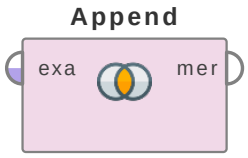
Case 3:The 'Market-Data' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet before application of the Transpose operator. You can see that the 'Market-Data' data set has no special attributes. The type of all attributes is integer. Press the Run button to continue. Now the Transpose operator is applied on the 'Market-Data' data set. A breakpoint is inserted here so that you can see the ExampleSet after the application of the Transpose operator. Here you can see that a new attribute with id role has been created. Values of the new id attribute are the names of the old attributes. New

attributes are named in a general format like ‘att_1’, ‘att_2’ etc because the input ExampleSet had no id attribute. The Type of all new attributes is real because there were attributes with mixed numbers type in the input ExampleSet.

Case 4:The ‘Golf-Testset’ data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet before application of the Transpose operator. The Transpose operator is applied on the ‘Golf-Testset’ data set. Then the Transpose operator is applied on the output of the first Transpose operator. Note that the types of the attributes of the original ExampleSet and the Transpose of the Transpose of the original data set are different.

## 2.3.3 Joins

# Append




This operator builds a merged ExampleSet from two or more compatible ExampleSets by adding all examples into a combined set.

## Description

This operator builds a merged ExampleSet from two or more compatible ExampleSets by adding all examples into a combined set. All input ExampleSets must have the same attribute signature. This means that all ExampleSets must have the same number of attributes. Names and roles of attributes should be the same in all input ExampleSets. Please note that the merged ExampleSet is built in memory and this operator might therefore not be applicable for merging huge data set tables from database. In that case other preprocessing tools should be used that aggregate, join, and merge tables into one table which is then used by RapidMiner.

## Input Ports

**example set *(exa)*** The Append operator can have multiple inputs. When one *input* port is connected, another *input* port becomes available which is ready to accept another input (if any).This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. Output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with data.

## Output Ports

**merged set *(mer)*** The merged ExampleSet is delivered through this port.

## Parameters

**data management *(selection)*** This is an expert parameter. A long list is provided; users can select any option from this list.

## Tutorial Processes

### Merging Golf and Golf-Testset data sets

In this process the 'Golf' data set and 'Golf-Testset' data set are loaded using the Retrieve operators. Breakpoints are inserted after the Retrieve operators so that you can have a look at the input ExampleSets. When you run the process, first you see the 'Golf' data set. As you can see, it has 14 examples. When you continue the process, you will see the 'Golf-Testset' data set. It also has 14 examples. The Append operator is applied to merge these two ExampleSets into a single ExampleSet. The merged ExampleSet has all examples from all input ExampleSets, thus it has 28 examples. You can see that both input ExampleSets had the same number of attributes,

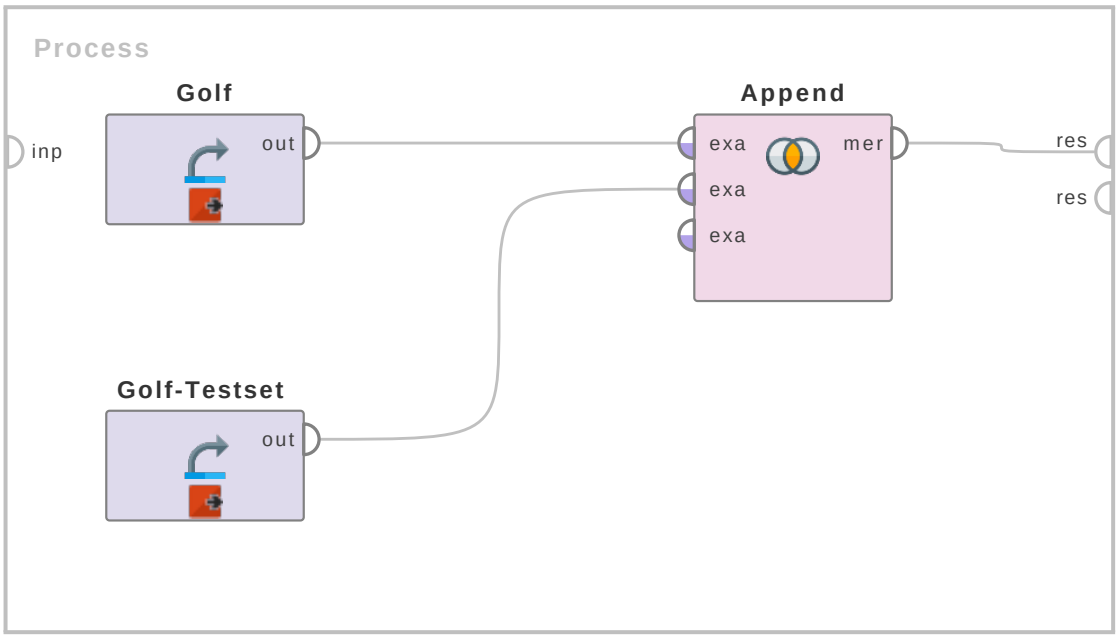
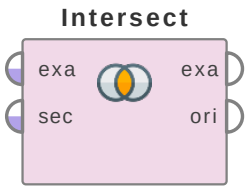



Figure 2.60: Tutorial process ‘Merging Golf and Golf-Testset data sets’.

same names and roles of attributes. This is why the Append operator could produce a merged ExampleSet.

# Intersect




This operator returns those examples of the first ExampleSet (given at the *example set input* port) whose IDs are contained within the other ExampleSet (given at the *second* port). It is necessary that both ExampleSets should have the ID attribute. The ID attribute of both ExampleSets should be of the same type.

## Description

This operator performs a set intersection on two ExampleSets on the basis of the ID attribute i.e. the resulting ExampleSet contains all the examples of the first ExampleSet (given at the *example set input* port) whose IDs appear in the second ExampleSet (given at the *second* port). It is important to note that the ExampleSets do not need to have the same number of columns or the same data types. The operation only depends on the ID attributes of the ExampleSets. It should be made sure that the ID attributes of both ExampleSets are of the same type i.e. either both are nominal or both are numerical.

## Differentiation

- **Set Minus** The Set Minus and Intersect operators can be considered as opposite of each other. The Set Minus operator performs a set minus on two ExampleSets on the basis of the ID attribute i.e. the resulting ExampleSet contains all the examples of the first ExampleSet whose IDs do NOT appear in the second ExampleSet. See page 281 for details.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Generate ID operator in the attached Example Process because this operator only works if the ExampleSets have the ID attribute.

**second *(sec)*** This input port expects an ExampleSet. It is the output of the Generate ID operator in the attached Example Process because this operator only works if the ExampleSets have the ID attribute.

## Output Ports

**example set output *(exa)*** The ExampleSet with remaining examples (i.e. examples remaining after the set intersection) of the first ExampleSet is output of this port.

**original *(ori)*** The ExampleSet that was given as input (at *example set input* port) is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Related Documents

- **Set Minus** (page 281)

## Tutorial Processes

### Intersection of two ExampleSets

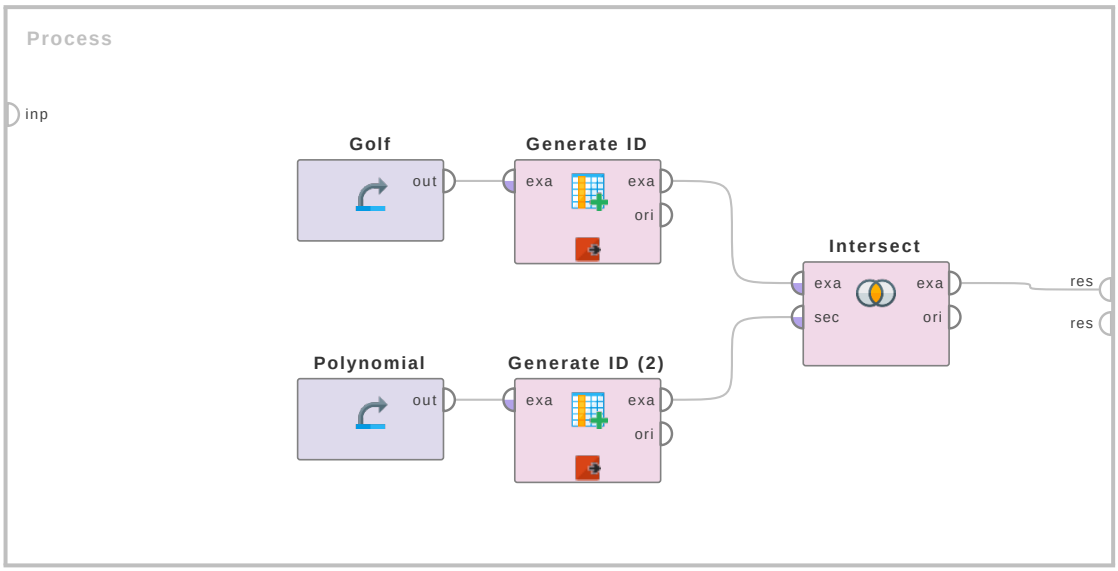



Figure 2.61: Tutorial process ‘Intersection of two ExampleSets’.

The ‘Golf’ data set is loaded using the Retrieve operator. The Generate ID operator is applied on it with the offset parameter set to 0. Thus the ids of the ‘Golf’ data set are from 1 to 14. A breakpoint is inserted here so you can have a look at the ‘Golf’ data set. The ‘Polynomial’ data set is loaded using the Retrieve operator. The Generate ID operator is applied on it with the offset parameter set to 10. Thus the ids of the ‘Polynomial’ data set are from 11 to 210. A breakpoint is inserted here so you can have a look at the ‘Polynomial’ data set.

The Intersect operator is applied next. The ‘Golf’ data set is provided at the example set input port and the ‘Polynomial’ data set is provided at the second port. The order of ExampleSets is very important. The Intersect operator compares the ids of the ‘Golf’ data set with the ids of the ‘Polynomial’ data set and then returns only those examples of the ‘Golf’ data set whose id is present in the ‘Polynomial’ data set. The ‘Golf’ data set ids are from 1 to 14 and the ‘Polynomial’ data set ids are from 11 to 210. Thus ‘Golf’ data set examples with ids 11 to 14 are returned by the Intersect operator. It is important to note that the meta data of both ExampleSets is very different but it does not matter because the Intersect operator only depends on the ID attribute.

If the ExampleSets are switched at the input ports of the Intersect operator the results will be very different. In this case the Intersect operator returns only those examples of the ‘Polynomial’ data set whose id is present in the ‘Golf’ data set. The ‘Golf’ data set ids are from 1 to 14 and the ‘Polynomial’ data set ids are from 11 to 210. Thus the ‘Polynomial’ data set examples with ids 11 to 14 are returned by the Intersect operator.

# Join


This Operator joins two ExampleSets using one or more Attributes of the input ExampleSets as *key attributes*.

## Description

This Operator joins two ExampleSets using one or more Attributes of the input ExampleSets as *key attributes*.

Identical values of the *key attributes* indicate matching Examples. An Attribute with id role is selected as key by default but an arbitrary set of one or more Attributes can be chosen as key. Four types of joins are possible: *inner*, *left*, *right* and *outer* join. All these types of joins are explained in the parameters section.

## Differentiation

- **Append**

  The Append Operator merges the Examples of the input ExampleSets into the resulting ExampleSet. Therefore all input ExampleSet need to have the same structure (number of Attributes, Attribute names and value types).

  See page 274 for details.

- **Cartesian Product**

  The Cartesian Product Operator builds a cartesian product of the input ExampleSets, i.e. every Example from the left ExampleSet is joined with each Example of the right ExampleSet.

  See page **??** for details.

- **Union**

  The Union Operator combines both input ExampleSets in such a way that all Attributes and Examples are part of the resulting union ExampleSet.

  See page 285 for details.

- **Superset**

  The Superset Operator expects two ExampleSets as input and adds the Attributes of the first ExampleSet to the second ExampleSet and vice versa. Both resulting ExampleSets are delivered as output of the Superset Operator.

  See page 283 for details.

## Input Ports

**left *(lef)*** The left input port expects an ExampleSet. This ExampleSet will be used as the left ExampleSet for the join.

**right *(rig)*** The right input port expects an ExampleSet. This ExampleSet will be used as the right ExampleSet for the join.

## Output Ports

**join *(joi)*** The output port delivers the joint ExampleSet.

## Parameters

**remove double attributes** This parameter indicates if double Attributes should be removed or renamed. Double Attributes are those Attributes that are present in both ExampleSets. If this parameter is checked, from Attributes which are present in both ExampleSets only the one from the left ExampleSet will be taken and the one from the right ExampleSet will be discarded. If this parameter is unchecked, the Attributes from the right ExampleSet are renamed. The *key attributes* will always be taken from the left ExampleSet. Please note that this check for double Attributes will only be applied for regular Attributes. Special Attributes of the right ExampleSet which do not exist in the left ExampleSet will simply be added. If they already exist they are simply skipped.

**join type** This parameter specifies which join should be performed. You can easily understand these joins by studying the tutorial Process. Four types of joins are supported:

- **inner** The resulting ExampleSet will contain only those Examples where the *key attributes* of both input ExampleSets match, i.e. have the same value.
- **left** This is also called left outer join. The resulting ExampleSet will contain all Examples from the left ExampleSet. If no matching Examples were found in the right ExampleSet, then its Attributes will consist of missing values. Missing values or null values are shown as '?' in RapidMiner. The left join will always contain the results of the inner join; however it can contain some Examples that have no matching Examples in the right ExampleSet.
- **right** This is also called right outer join. The resulting ExampleSet will contain all records from the right ExampleSet. If no matching Examples were found in the left ExampleSet, then its Attributes will consist of missing values. Missing values or null values are shown as '?' in RapidMiner. The right join will always contain the results of the inner join; however it can contain some Examples that have no matching Examples in the left ExampleSet.
- **outer** This is also called full outer join. This type of join combines the results of the left and the right join. All Examples from both ExampleSets will be part of the resulting ExampleSet, whether the matching *key attribute* value exists in the other ExampleSet or not. If no matching *key attribute* value was found the corresponding resulting Attributes will consist of missing values. Missing values or null values are shown as '?' in RapidMiner. The outer join will always contain the results of the inner join; however it can contain some Examples that have no matching Examples in the other ExampleSet.

**use id attribute as key** This parameter indicates if the Attribute with the id role should be used as the *key attribute*. This option is checked by default. If unchecked, then you have to specify the *key attributes* for both left and right ExampleSets. Identical values of the *key attributes* indicate matching Examples.

**key attributes** This parameter is available when when the parameter *use id attribute as key* is unchecked. This parameter specifies Attribute(s) which are used as the *key attributes*. Identical values of the *key attributes* indicate matching Examples. For each *key attribute* from the left ExampleSet a corresponding *key attribute* from the right ExampleSet has to be chosen. Choosing appropriate *key attributes* is critical for obtaining the desired results.

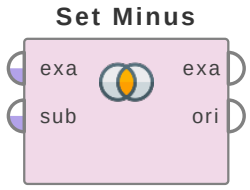
**keep both join attributes** If checked, both Attributes of a join pair will be kept. Usually this is unneccessary since both Attributes are identical. It may be useful to keep such a column if there are missing values on one side.

## Tutorial Processes

### Explore the different join types

After creating two similar ExampleSets which are connected to each port of the Join Operator you can play around with the available join types. The description inside this process explains the created ExampleSets as well as the results of each join type.

# Set Minus




This operator returns those examples of the ExampleSet (given at the *example set input* port) whose IDs are not contained within the other ExampleSet (given at the *subtrahend* port). It is necessary that both ExampleSets should have the ID attribute. The ID attribute of both ExampleSets should be of the same type.

## Description

This operator performs a set minus on two ExampleSets on the basis of the ID attribute i.e. the resulting ExampleSet contains all the examples of the minuend ExampleSet (given at the *example set input* port) whose IDs do not appear in the subtrahend ExampleSet (given at the *subtrahend* port). It is important to note that the ExampleSets do not need to have the same number of columns or the same data types. The operation only depends on the ID attributes of the ExampleSets. It should be made sure that the ID attributes of both ExampleSets are of the same type i.e. either both are nominal or both are numerical.

## Differentiation

- **Intersect** The Set Minus and Intersect operators can be considered as opposite of each other. The Intersect operator performs a set intersect on two ExampleSets on the basis of the ID attribute i.e. the resulting ExampleSet contains all the examples of the first ExampleSet whose IDs appear in the second ExampleSet. See page 276 for details.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Generate ID operator in the attached Example Process because this operator only works if the ExampleSets have the ID attribute.

**subtrahend *(sub)*** This input port expects an ExampleSet. It is the output of the Generate ID operator in the attached Example Process because this operator only works if the ExampleSets have the ID attribute.

## Output Ports

**example set output *(exa)*** The ExampleSet with remaining examples (i.e. examples remaining after the set minus) of the minuend ExampleSet is output of this port.

**original *(ori)*** The ExampleSet that was given as input (at *example set input* port) is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Related Documents

- **Intersect** (page 276)

## Tutorial Processes

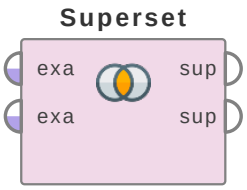
### Introduction to the Set Minus operator

The 'Golf' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it with the offset parameter set to 0. Thus the ids of the 'Golf' data set are from 1 to 14. A breakpoint is inserted here so you can have a look at the 'Golf' data set. The 'Polynomial' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it with the offset parameter set to 10. Thus the ids of the 'Polynomial' data set are from 11 to 210. A breakpoint is inserted here so you can have a look at the 'Polynomial' data set.

The Set Minus operator is applied next. The 'Golf' data set is provided at the example set input port and the 'Polynomial' data set is provided at the subtrahend port. The order of ExampleSets is very important. The Set Minus operator compares the ids of the 'Golf' data set with the ids of the 'Polynomial' data set and then returns only those examples of the 'Golf' data set whose id is not present in the 'Polynomial' data set. The 'Golf' data set ids are from 1 to 14 and the 'Polynomial' data set ids are from 11 to 210. Thus 'Golf' data set examples with ids 1 to 10 are returned by the Set Minus operator. It is important to note that the meta data of both ExampleSets is very different but it does not matter because the Set Minus operator only depends on the ID attribute.

If the ExampleSets are switched at the input ports of the Set Minus operator the results will be very different. In this case the Set Minus operator returns only those examples of the 'Polynomial' data set whose id is not present in the 'Golf' data set. The 'Golf' data set ids are from 1 to 14 and the 'Polynomial' data set ids are from 11 to 210. Thus the 'Polynomial' data set examples with ids 15 to 210 are returned by the Set Minus operator.

# Superset




This operator takes two ExampleSets as input and adds new features of the first ExampleSet to the second ExampleSet and vice versa to generate two supersets. The resultant supersets have the same set of attributes but the examples may be different.

## Description

The Superset operator generates supersets of the given ExampleSets by adding new features of one ExampleSet to the other ExampleSet. The values of the new features are set to missing values in the supersets. This operator delivers two supersets as output:

1. The first has all attributes and examples of the first ExampleSet + all attributes of the second ExampleSet (with missing values)

2. The second has all attributes and examples of the second ExampleSet + all attributes of the first ExampleSet (with missing values)

Thus both supersets have the same set of regular attributes but the examples may be different. It is important to note that the supersets can have only one special attribute of a kind. By default this operator adds only new 'regular' attributes to the other ExampleSet for generating supersets. For example, if both input ExampleSets have a label attribute then the first superset will have all attributes of the first ExampleSet (including label) + all regular attributes of the second ExampleSet. The second superset will behave correspondingly. The *include special attributes* parameter can be used for changing this behavior. But it should be used carefully because even if this parameter is set to true, the resultant supersets can have only one special attribute of a kind. Please study the attached Example Process for better understanding.

## Input Ports

**example set 1** ***(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.

**example set 2** ***(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.

## Output Ports

**superset 1** ***(sup)*** The first superset of the input ExampleSets is delivered through this port.

**superset 2** ***(sup)*** The second superset of the input ExampleSets is delivered through this port.

## Parameters

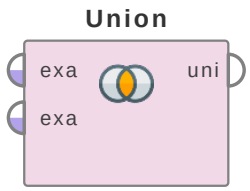
**include special attributes** ***(boolean)*** This parameter indicates if the special attributes should be included for generation of the supersets. This operator should be used carefully especially if both ExampleSets have the same special attributes because the resultant supersets can have only one special attribute of a kind.

## Tutorial Processes

### Generating supersets of the Golf and Iris data sets

In this process the 'Golf' and 'Iris' data sets are loaded using the Retrieve operators. Breakpoints are inserted after the Retrieve operators so that you can have a look at the input ExampleSets. When you run the process, first you see the 'Golf' data set. It has four regular and one special attribute with 14 examples each. When you continue the process, you will see the 'Iris' data set. It has four regular and two special attributes with 150 examples each. Note that the meta data of both ExampleSets is very different. The Superset operator is applied for generating supersets of these two ExampleSets. The resultant supersets can be seen in the Results Workspace. You can see that one superset has all attributes and examples of the 'Iris' data set + 4 regular attributes of the 'Golf' data set (with missing values). The other superset has all attributes and examples of the 'Golf' data set + 4 regular attributes of the 'Iris' data set (with missing values).

# Union




This operator builds the union of the input ExampleSets. The input ExampleSets are combined in such a way that attributes and examples of both input ExampleSets are part of the resultant union ExampleSet.

## Description

The Union operator builds the superset of features of both input ExampleSets such that all regular attributes of both ExampleSets are part of the superset. The attributes that are common in both ExampleSets are not repeated in the superset twice, a single attribute is created that holds data of both ExampleSets. If the special attributes of both input ExampleSets are compatible with each other then only one special attribute is created in the superset which has examples of both the input ExampleSets. If special attributes of ExampleSets are not compatible, the special attributes of the first ExampleSet are kept. If both ExampleSets have any attributes with the same name, they should be compatible with each other; otherwise you will get an error message. This can be understood by studying the attached Example Process.

## Input Ports

**example set 1** ***(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.

**example set 2** ***(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.

## Output Ports

**union** ***(uni)*** The union of the input ExampleSets is delivered through this port.

## Tutorial Processes

### Union of the Golf and Golf-Testset data sets

In this process the 'Golf' data set and 'Golf-Testset' data set are loaded using the Retrieve operators. Breakpoints are inserted after the Retrieve operators so that you can have a look at the input ExampleSets. When you run the process, first you see the 'Golf' data set. As you can see, it has 14 examples. When you continue the process, you will see the 'Golf-Testset' data set. It also has 14 examples. Note that the meta data of both ExampleSets is almost the same. The Union operator is applied to combine these two ExampleSets into a single ExampleSet. The combined ExampleSet has all attributes and examples from the input ExampleSets, thus it has 28 examples. You can see that both input ExampleSets had the same number of attributes, same names and roles of attributes. This is why the Union ExampleSet also has the same number of attributes

with the same names and roles. Here the Union operator behaves like the Append operator i.e. it simply combines examples of two ExampleSets with compatible meta data.

### Union of the Golf and Iris data sets

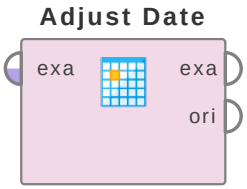
In this process the 'Golf' data set and the 'Iris' data set are loaded using the Retrieve operators. Breakpoints are inserted after the Retrieve operators so that you can have a look at the input ExampleSets. When you run the process, first you see the 'Golf' data set. As you can see, it has 14 examples. When you continue the process, you will see the 'Iris' data set. It has 4 regular and 2 special attributes with 150 examples. Note that the meta data of both ExampleSets is very different. The Union operator is applied to combine these two ExampleSets into a single ExampleSet. The combined ExampleSet has all attributes and examples from the input ExampleSets, thus it has 164 (14+150) examples. Note that the 'Golf' data set has an attribute with label role: the 'Play' attribute. The 'Iris' data set also has an attribute with label role: the 'label' attribute. As these two label attributes are not compatible, only the label attribute of the first ExampleSet is kept. The examples of 'Iris' data set have null values in this attribute of the resultant Union ExampleSet.

### Union of the Golf(with id attribute) and Iris data sets

In this process the 'Golf' data set and 'Iris' data set are loaded using the Retrieve operators. The Generate ID operator is applied on the Golf data set to generate nominal ids starting from id_1. Breakpoints are inserted before the Union operator so that you can have a look at the input ExampleSets. When you run the process, first you see the 'Golf' data set. As you can see, it has 14 examples. It has two special attributes. When you continue the process, you will see the 'Iris' data set. It has 4 regular and 2 special attributes with 150 examples. Note that the meta data of both ExampleSets is very different. The Union operator is applied to combine these two ExampleSets into a single ExampleSet. The combined ExampleSet has all attributes and examples from the input ExampleSets, thus it has 164 (14+150) examples. Note that the 'Golf' data set has an attribute with label role: the 'Play' attribute. The 'Iris' data set also has an attribute with label role: the 'label' attribute. As these two label attributes are not compatible, only the label attribute of the first ExampleSet is kept. The examples of the 'Iris' data set have null values in this attribute of the union ExampleSet. Also note that both input ExampleSets have id attributes. The names of these attributes are the same and they both have nominal values, thus these two attributes are compatible with each other. Thus a single id attribute is created in the resultant Union ExampleSet. Also note that the values of ids are not unique in the resultant ExampleSet.

# 2.4 Values

## Adjust Date




This operator adjusts the date in the specified attribute by adding or subtracting the specified amount of time.

### Description

The Adjust Date operator adjusts the values of the specified date attribute by adding or subtracting constant values. Year, month, day, hour, minute, second and millisecond adjustments are allowed. Multiple adjustments can be made to a single attribute. For example, you can add a month and subtract 2 hours from an attribute. If the *keep old attribute* parameter is set to true, the old attribute will be kept along with the adjusted attribute. Otherwise, the adjusted attribute will replace the previous attribute.

### Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input. The ExampleSet should have at least one date/time attribute because if there is no such attribute, the use of this operator does not make sense.

### Output Ports

**example set output *(exa)*** The values of the selected date attribute are adjusted and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

### Parameters

**attribute name *(string)*** This parameter specifies the name of the date attribute which should be adjusted.

**adjustments *(list)*** This parameter defines the list of all date adjustments. Multiple adjustments can be made to a single attribute. For example, you can add a month and subtract 2 hours from the selected attribute.

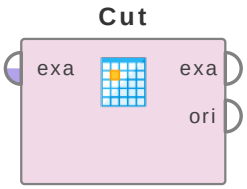
**keep old attribute *(boolean)*** This parameter indicates if the original date attribute should be kept. If this parameter is set to true, the old attribute will be kept along with the adjusted attribute. Otherwise, the adjusted attribute will replace the previous attribute.

## Tutorial Processes

### Making multiple adjustments in a date attribute

This Example Process starts with the Subprocess operator. The operator chain inside the Subprocess operator generates an ExampleSet for this process. The explanation of this inner chain of operators is not relevant here. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that this ExampleSet has a date attribute named 'deadline_date'. The Adjust Date operator is applied on this ExampleSet to adjust this date attribute. Two adjustments are made to this attribute. 1) 5 days are added 2) 2 months are subtracted. Run the process and compare the resultant ExampleSet with the original ExampleSet. You can clearly see that the date values have been adjusted. For example, the date value 20-August has been changed to 25-June after addition of 5 days and subtraction of two months.

# Cut




This operator cuts the nominal values of the specified regular attributes. The resultant attributes have values that are substrings of the original attribute values.

## Description

The Cut operator creates new attributes from nominal attributes where the new attributes contain only substrings of the original values. The range of characters to be cut is specified by the *first character index* and *last character index* parameters. The *first character index* parameter specifies the index of the first character and the *last character index* parameter specifies the index of the last character to be included. All characters of the attribute values that are at index equal to or greater than the *first character index* and less than or equal to the *last character index* are included in the resulting substring. Please note that the counting starts with 1 and that the first and the last character will be included in the resulting substring. For example, if the value is "RapidMiner" and the first index is set to 6 and the last index is set to 9 the result will be "Mine". If the last index is larger than the length of the word, the resulting substrings will end with the last character.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

## Output Ports

**example set output *(exa)*** The ExampleSet with new attributes that have values that are substrings of the original attributes is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another

parameter becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute *(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *parameter* attribute if the meta data is known.

**attributes *(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and shifted to the right list, which is the list of selected attributes.

**regular expression *(string)*** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression *(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter *(except regular expression*) becomes visible in the Parameters panel.

**except regular expression *(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the *regular expression* parameter).

**value type *(selection)*** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception *(boolean)*** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will not be selected even if they match the previously mentioned type i.e. *value type* parameter's value.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be not be selected even if they match the previously mentioned block type i.e. *block type* parameter's value.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**first character index** ***(integer)*** This parameter specifies the index of the first character of the substring which should be kept. Please note that the counting starts with 1.

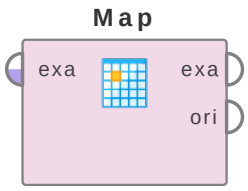
**last character index** ***(integer)*** This parameter specifies the index of the last character of the substring which should be kept. Please note that the counting starts with 1.

## Tutorial Processes

### Applying the Cut operator on label of the Iris data set

The 'Iris' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the data set before application of the Cut operator. You can see that the label attribute has three possible values: 'Iris-setosa', 'Iris-versicolor' and 'Iris-virginica'. If we want to remove the 'Iris-' substring from the start of all the label values we can use the Cut operator. The Cut operator is applied on the Iris data set. The first character index parameter is set to 6 because we want to remove first 5 characters ('Iris-'). The last character index parameter can be set to any value greater than the length of longest possible value. Thus the last character index parameter can be safely set to 20 because if the last index is larger than the length of the word, the resulting substrings will end with the last character. Run the process and you can see that the substring 'Iris-' has been removed from the start of all possible values of the label attribute.

# Map




This operator maps specified values of selected attributes to new values. This operator can be applied on both numerical and nominal attributes.

## Description

This operator can be used to replace nominal values (e.g. replace the value 'green' by the value 'green_color') as well as numerical values (e.g. replace all values '3' by '-1'). But, one use of this operator can do mappings for attributes of only one type. A single mapping can be specified using the parameters *replace what* and *replace by* as in Replace operator. Multiple mappings can be specified through the *value mappings* parameter. Additionally, the operator allows defining a default mapping. This operator allows you to select attributes to make mappings in. This operator allows you to specify a regular expression. Attribute values of selected attributes that match this regular expression are mapped by the specified value mapping. Please go through the parameters and the Example Process to develop a better understanding of this operator.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with data.

## Output Ports

**example set *(exa)*** The ExampleSet with value mappings is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes on which you want to apply mappings. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This

option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)

- **regular_expression** This option allows you to specify a regular expression for the attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. The user should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are not selected.
- **numeric_value_filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute *(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *parameter* attribute if the meta data is known.

**attributes *(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes.

**regular expression *(string)*** Attributes whose name match this expression will be selected. Regular expression is very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression *(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (except regular expression) becomes visible in the Parameters panel.

**except regular expression *(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in *regular expression* parameter).

**value type *(selection)*** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameters panel.

**except value type** ***(selection)*** Attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type*parameter's value.

**block type** ***(selection)*** Block type of attributes to be selected can be chosen from drop down list.

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameters panel.

**except block type** ***(selection)*** Attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** ***(string)*** Numeric condition for testing examples of numeric attributes is mention here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** Special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is removed prior to selection of this parameter. After selection of this parameter 'att1' will be removed and 'att2' will be selected.

**value mappings** Multiple mappings can be specified through this parameter. If only a single mapping is required. It can be done using the parameters *replace what* and *replace by* as in the Replace operator. Old values and new values can be easily specified through this parameter. Multiple mappings can be defined for the same old value but only the new value corresponding to the first mapping is taken as replacement. Regular expressions can also be used here if the *consider regular expressions* parameter is set to true.

**replace what** ***(string)*** This parameter specifies what is to be replaced. This can be specified using regular expressions. This parameter is useful only if single mapping is to be done. For multiple mappings use the *value mappings* parameter

**replace by** ***(string)*** Regions matching regular expression of the *replace what* parameter are replaced by the value of the *replace by* parameter.This parameter is useful only if single mapping is to be done. For multiple mappings use the *value mappings* parameter.

**consider regular expressions *(boolean)*** This parameter enables matching based on regular expressions; old values(old values are original values, old values and 'replace what' represent the same thing) may be specified as regular expressions. If the parameter *consider regular expressions* is enabled, old values are replaced by the new values if the old values match the given regular expressions. The value corresponding to the first matching regular expression in the mappings list is taken as a replacement.

**add default mapping *(boolean)*** If set to true, all values that occur in the selected attributes of the ExampleSet but are not listed in the value mappings list are mapped to the value of the *default value* parameter.

**default value *(string)*** This parameter is only available if the *add default mapping* parameter is checked. If *add default mapping* is set to true and the *default value* is properly set, all values that occur in the selected attributes of the ExampleSet but are not listed in the value mappings list are replaced by the*default value*. This may be helpful in cases where only some values should be mapped explicitly and many unimportant values should be mapped to a default value (e.g. 'other').

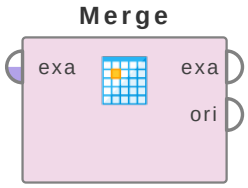
## Tutorial Processes

### Mapping multiple values

Focus of this Example Process is the use of the value mappings parameter and the default value parameter. Use of the replace what and replace by parameter can be seen in the Example Process of the Replace operator. Almost all other parameters of the Map operator are also part of the Select Attributes operator, their use can be better understood by studying the Attributes operator and it's Example Process.

The 'Golf' data set is loaded using the Retrieve operator. The Map operator is applied on it. 'Wind' and 'Outlook' attributes are selected for mapping. Thus, the effect of the Map operator will be limited to just these two attributes. Four value mappings are specified in the value mappings parameter. 'true', 'false', 'overcast' and 'sunny' are replaced by 'yes', 'no', 'bad' and 'good' respectively. The add default mappings parameter is set to true and 'other' is specified in the default value parameter. 'Wind' attribute has only two possible values i.e. 'true' and 'false'. Both of them were mapped in the mappings list. 'Outlook' attribute has three possible values i.e. 'sunny', 'overcast' and 'rain'. 'sunny' and 'overcast' were mapped in the mappings list but 'rain' was not mapped. As add default mappings parameter is set to true, 'rain' will be mapped to the default value i.e. 'other'.

# Merge




This operator merges two nominal values of the specified regular attribute.

## Description

The Merge operator is used for merging two nominal values of the specified attribute of the input ExampleSet. Please note that this operator can merge only the values of regular attributes. The required regular attribute is specified using the *attribute name* parameter. The *first value* parameter is used for specifying the first value to be merged. The *second value* parameter is used for specifying the second value to be merged. The two values are merged in '*first_second*' format where *first* is the value of the *first value* parameter and *second* is the value of the *second value* parameter. It is not compulsory for the *first value* and *second value* parameters to have values from the range of possible values of the selected attribute. However, at least one of the *first value* and *second value* parameters should have a value from the range of possible values of the selected attribute. Otherwise this operator will have no affect on the input ExampleSet.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**example set output *(exa)*** The ExampleSet with the merged attribute values is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute name *(string)*** The required nominal attribute whose values are to be merged is selected through this parameter. This operator can be applied only on regular attributes.

**first value *(string)*** This parameter is used for specifying the first value to be merged. It is not compulsory for the *first value* parameter to have a value from the range of possible values of the selected attribute.

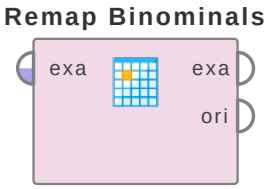
**second value *(string)*** This parameter is used for specifying the second value to be merged. It is not compulsory for the *second value* parameter to have a value from the range of possible values of the selected attribute.

## Tutorial Processes

### Introduction to the Merge operator

The Golf data set is loaded using the Retrieve operator. The Merge operator is applied on it. The attribute name parameter is set to 'Outlook'. The first value parameter is set to 'sunny' and the second value parameter is set to 'hot'. All the occurrences of value 'sunny' are replaced by 'sunny_hot' in the Outlook attribute of the resultant ExampleSet. Now set the value of the second value parameter to 'rain' and run the process again. As 'rain' is also a possible value of the Outlook attribute, all occurrences of 'sunny' and 'rain' in the Outlook attribute are replaced by 'sunny_rain' in the resultant ExampleSet. This Example Process is just to explain basic working of the Merge operator.

# Remap Binominals




This operator modifies the internal value mapping of binominal attributes according to the specified negative and positive values.

## Description

The Remap Binominals operator modifies the internal mapping of binominal attributes according to the specified positive and negative values. The positive and negative values are specified by the *positive value* and *negative value* parameters respectively. If the internal mapping differs from the specified values then the internal mapping is switched. If the internal mapping contains other values than the specified ones the mapping is not changed and the attribute is simply skipped. Please note that this operator changes the internal mapping so the changes are not explicitly visible in the ExampleSet. This operator can be applied only on binominal attributes. Please note that if there is a nominal attribute in the ExampleSet with only two possible values, this operator will still not be applicable on it. This operator requires the attribute to be explicitly defined as binominal in the meta data.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. Please note that there should be at least one binominal attribute in the input ExampleSet.

## Output Ports

**example set output *(exa)*** The resultant ExampleSet is output of this port. Externally this data set is the same as the input ExampleSet, only the internal mappings may be changed.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute *(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of *parameter* attribute if the meta data is known.

**attributes *(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list. Attributes can be shifted to the right list, which is the list of selected attributes.

**regular expression *(string)*** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression *(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter *(except regular expression*) becomes visible in the Parameters panel.

**except regular expression *(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the *regular expression* parameter).

**value type *(selection)*** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception *(boolean)*** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type *(selection)*** The attributes matching this type will not be selected even if they match the previously mentioned type i.e. *value type* parameter's value.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be not be selected even if they match the previously mentioned block type i.e. *block type* parameter's value.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**negative value** ***(string)*** This parameter specifies the internal mapping for the negative or false value of the selected binominal attributes.

**positive value** ***(string)*** This parameter specifies the internal mapping for the positive or true value of the selected binominal attributes.

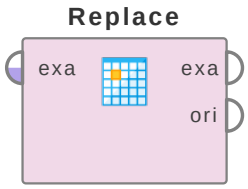
## Tutorial Processes

### Changing mapping of the Wind attribute of the Golf data set

The 'Golf' data set is loaded using the Retrieve operator. In this Example Process we shall change the internal mapping of the 'Wind' attribute of the 'Golf' data set. A breakpoint is inserted after the Retrieve operator so that you can view the 'Golf' data set. As you can see the 'Wind' attribute of the 'Golf' data set is nominal but it has only two possible values. The Remap Binominals operator cannot be applied on such an attribute; it requires that the attribute should be explicitly declared as binominal in the meta data. To accomplish this, the Nominal to Binominal operator is applied on the 'Golf' data set to convert the 'Wind' attribute to binominal type. A breakpoint is inserted here so that you can view the ExampleSet. Now that the 'Wind' attribute has been converted to binominal type, the Remap Binominals operator can be applied on it. The 'Wind' attribute is selected in the Remap Binominals operator. The negative value and positive value parameter are set to 'true' and 'false' respectively. Run the process and the

internal mapping is changed. This change is an internal one so it will not be visible explicitly in the Results Workspace. Now change the value of the positive value and negative value parameters to ‘a’ and ‘b’ respectively and run the complete process. Have a look at the log. You will see the following message: “WARNING: Remap Binominals: specified values do not match values of attribute Wind, attribute is skipped.” This log shows that as the values ‘a’ and ‘b’ are not values of the ‘Wind’ attribute so no change in mapping is done.

# Replace




This operator replaces parts of the values of selected nominal attributes matching a specified regular expression by a specified replacement.

## Description

This operator allows you to select attributes to make replacements in and to specify a regular expression. Attribute values of selected attributes that match this regular expression are replaced by the specified replacement. The replacement can be empty and can contain capturing groups. Please keep in mind that although regular expressions are much more powerful than simple strings, you might simply enter characters to search for.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with data.

## Output Ports

**example set *(exa)*** An ExampleSet with replacements is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes in which you want to make replacements. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in the Parameters panel.(Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.(Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. User should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are not selected.
- **numeric_value_filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *parameter* attribute if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes.

**regular expression** ***(string)*** Attributes whose name match this expression will be selected. Regular expression is very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. It gives a good idea of regular expressions and also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (except regular expression) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from drop down list.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameters panel.

**except value type** ***(selection)*** Attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type*parameter's value.

**block type** ***(selection)*** The Block type of attributes to be selected can be chosen from drop down list.

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameters panel.

**except block type** ***(selection)*** Attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** ***(string)*** Numeric condition for testing examples of numeric attributes is mention here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** Special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is removed prior to selection of this parameter. After selection of this parameter 'att1' will be removed and 'att2' will be selected.

**replace what** ***(string)*** This parameter specifies what is to be replaced. This can be specified using regular expressions. The *edit regular expression* menu can assist you in specifying the right regular expression.

**replace by** ***(string)*** The regions matching regular expression of the *replace what* parameter are replaced by the value of the *replace by* parameter.

## Tutorial Processes

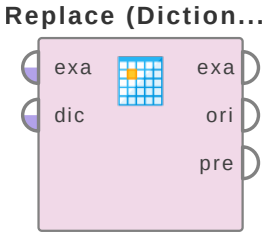
### Use of replace what and replace by parameters

The focus of this process is to show the use of the replace what and replace by parameters. All other parameters are for the selection of attributes on which the replacement is to be made. For understanding these parameters please study the Example Process of the Select Attributes operator.

The 'Golf' data set is loaded using the Retrieve operator. The attribute filter type parameter is set to 'all' and the include special attributes parameter is also checked. Thus, replacements are made on all attributes including special attributes. The replace what parameter is provided with the regular expression '.*e.*' which means any attribute value that has character 'e' in it. The

replace by parameter is given the value ‘E’. Run the process. You will see that ‘E’ is placed in place of ‘yes’, ‘overcast’, ‘true’ and ‘false’. This is because all the values have an ‘e’ in it. You can see the power of this operator. Now set the regular expression of replace what operator to ‘e’. Run the process again. This time you will see that the entire values are not replaced by ‘E’, instead only the character ‘e’ is replaced by ‘E’. Thus new values of ‘yes’, ‘overcast’, ‘true’ and ‘false’ are ‘yEs’, ‘ovErcast’, ‘truE’ and ‘falsE’ respectively. You can see the power of this operator and regular expressions. Thus it should be made sure that the correct regular expression is provided. If you leave the replace by parameter empty or write ‘?’ in it, the null value is used as replacement

# Replace (Dictionary)




This operator replaces substrings (in the values) of the selected nominal attributes of the first ExampleSet by using the dictionary specified by the second ExampleSet.

## Description

This operator takes two ExampleSets as input. It replaces substrings (in the values) of the selected nominal attributes of the first ExampleSet by using the value-mappings defined in the second ExampleSet. This operator uses the second ExampleSet as a dictionary. The second ExampleSet must have two nominal attributes for value-mappings i.e. the 'from' attribute (i.e. specified through the *from attribute* parameter) and the 'to' attribute (i.e. specified through the *to attribute* parameter). For every example in the second ExampleSet a dictionary entry is created that matches the 'from attribute' value to the 'to attribute' value. Finally, this dictionary is used for replacing substrings in the first ExampleSet. If the values of the 'from' attribute of the second ExampleSet are found (as a whole or as a substring) in the selected nominal attributes of the first ExampleSet, then the corresponding value of the 'to' attribute is used as a replacement for the substring in the first ExampleSet. Please study the attached Example Process for better understanding.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. The ExampleSet should have at least one nominal attribute because if there is no such attribute, the use of this operator does not make sense. The substrings of this ExampleSet will be replaced by using the second ExampleSet.

**dictionary *(dic)*** This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input. This ExampleSet should have a 'from attribute' and 'to attribute' as specified in the description of this operator. These attributes will be used for substring replacements in the first ExampleSet.

## Output Ports

**example set output *(exa)*** The substrings of the selected nominal attributes of the first ExampleSet are replaced and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The first ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model *(pre)*** This port delivers the preprocessing model, which has the information regarding the parameters of this operator in the current process.

## Parameters

**create view *(boolean)*** It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When it is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute *(string)*** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes *(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression *(string)*** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu

also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**from attribute** ***(string)*** This parameter specifies the name of the attribute of the second ExampleSet that specifies the substrings that should be replaced.

**to attribute** ***(string)*** This parameter specifies the name of the attribute of the second ExampleSet that specifies the replacements of the substrings.

**use regular expressions** ***(boolean)*** This parameter specifies if the replacements should be treated as regular expressions.

**convert to lowercase** ***(boolean)*** This parameter specifies if the strings should be converted to lower case.

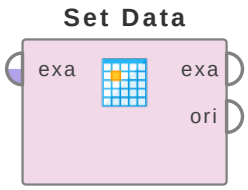
**first match only** ***(boolean)*** This parameter specifies if only the first match in the dictionary should be considered. If set to false, subsequent matches will be applied iteratively.

## Tutorial Processes

### Replacing substrings by using a dictionary

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at this ExampleSet. This ExampleSet will be used as the first ExampleSet for the Replace (Dictionary) operator. Therefore substring replacements will be made in this ExampleSet. The second ExampleSet is provided by the Subprocess operator. The operator chain inside the Subprocess operator generates a dictionary ExampleSet for this process. The explanation of this inner chain of operators is not relevant here. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that this ExampleSet has two nominal attributes 'att1' and 'att2'. The Replace (Dictionary) operator takes these two ExampleSets as input and makes substring replacements in the first ExampleSet by using the second ExampleSet. Have a look at the parameters of the Replace (Dictionary) operator. The attribute filter type parameter is set to 'all', thus substring replacements will be done in all attributes of the first ExampleSet. The from attribute and to attribute parameters are set to 'att1' and 'att2' respectively. Thus if the values of the 'att1' attribute (i.e. 'true' and 'false') are found in any attribute of the first ExampleSet, they will be replaced by the corresponding 'att2' attribute values (i.e. 'YES' and 'NO' respectively). All other parameters are used with default values. Run the process and compare the resultant ExampleSet with the original ExampleSet. You can clearly see in the Wind attribute that the substrings 'true' and 'false' have been replaced by 'YES' and 'NO' respectively. Please note that this operator is a substring replacement tool, although it was used for value replacement in this process. If the 'att1' attribute had the value 'tr' instead of 'true'; all occurrences of this substring would have been replaced by 'YES'. In that case 'true' value in the Wind attribute would have been changed to 'YESue'.

# Set Data




This operator sets the value of one or more attributes of the specified example.

## Description

The Set Data operator sets the value of one or more attributes of the specified example of the input ExampleSet. The example is specified by the *example index* parameter. The *attribute name* parameter specifies the attribute whose value is to be set. The *value* parameter specifies the new value. Values of other attributes of the same example can be set by the *additional values* parameter. Please note that the values should be consistent with the type of the attribute e.g. specifying a string value is not allowed for an integer type attribute.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

## Output Ports

**example set output *(exa)*** The ExampleSet with new values of the selected example's attributes is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**example index *(integer)*** This parameter specifies the index of the example whose value should be set. Please note that counting starts from 1.

**attribute name *(string)*** This parameter specifies the name of the attribute whose value should be set.

**count backwards *(boolean)*** If set to true, the counting order is reversed. The last example is addressed by index 1, the second last example is addressed by index 2 and so on.

**value *(string)*** This parameter specifies the new value of the selected attribute (selected by the *attribute name* parameter) of the specified example (specified by the *example index* parameter).

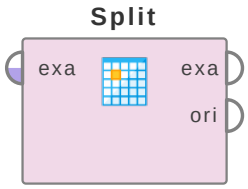
**additional values** The values of other attributes of the same example can be set by this parameter.

## Tutorial Processes

### Introduction to the Set Data operator

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the data set before application of the Set Data operator. You can see that the value of the Temperature and Wind attributes is '85' and 'false' respectively in the first example. The Set Data operator is applied on the 'Golf' data set. The example index parameter is set to 1, the attribute name parameter is set to 'Temperature' and the value parameter is set to 50. Thus the value of the Temperature attribute will be set to 50 in the first example. Similarly, the value of the Wind attribute in the first example is set to 'fast' using the additional values parameter. You can verify this by running the process and seeing the results in the Results Workspace. Please note that a string value cannot be set for the Temperature attribute because it is of integer type. An integer value can be specified for the Wind attribute (nominal type) but it will be stored as a nominal value.

# Split




This operator creates new attributes from the selected nominal attributes by splitting the nominal values into parts according to the specified split criterion.

## Description

The Split operator creates new attributes from the selected nominal attributes by splitting the nominal values into parts according to the split criterion which is specified through the *split pattern* parameter in form of a regular expression. This operator provides two different modes for splitting; the desired mode can be selected by the *split mode* parameter. The two splitting modes are explained with an imaginary ExampleSet with a nominal attribute named 'att' assumimg that the *split pattern* parameter is set to ',' (comma). Suppose the ExampleSet has three examples:

1. value1
2. value2, value3
3. value3

### Ordered Splits

In case of ordered split the resulting attributes get the name of the original attribute together with a number indicating the order. In our example scenario there will be two attributes named 'att_1' and 'att_2' respectively. After splitting the three examples will have the following values for 'att_1' and 'att_2' (described in form of tuples):

1. (value1,?)
2. (value2,value3)
3. (value3,?)

This mode is useful if the original values indicated some order like, for example, a preference.

### Unordered Splits

In case of unordered split the resulting attributes get the name of the original attribute together with the value for each of the occurring values. In our example scenario there will be three attributes named 'att_value1', 'att_value2' and 'att_value3' respectively. All these new attributes are boolean. After splitting the three examples will have the following values for 'att_value1', 'att_value2' and 'att_value3' (described in form of tuples):

1. (true, false, false)
2. (false, true, true)
3. (false, false, true)

This mode is useful if the order is not important but the goal is a basket like data set containing all occurring values.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input. The ExampleSet should have at least one nominal attribute because if there is no such attribute, the use of this operator does not make sense.

## Output Ports

**example set output *(exa)*** The selected nominal attributes are split into new attributes and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When it is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression** ***(string)*** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**split pattern** ***(string)*** This parameter specifies the pattern which is used for dividing the nominal values into different parts. It is specified in form of a regular expression. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions.

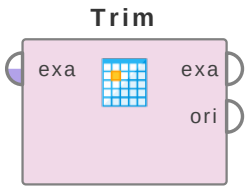
**split mode** ***(selection)*** This parameter specifies the split mode for splitting. The two options of this parameter are explained in the description of this operator.

## Tutorial Processes

### Ordered and unordered splits

This Example Process starts with a Subprocess operator. The operator chain inside the Subprocess operator generates an ExampleSet for this process. The explanation of this inner chain of operators is not relevant here. A breakpoint is inserted here so that you can have a look at the ExampleSet before the application of the Split operator. You can see that this ExampleSet is the same ExampleSet that is described in the description of this operator. The Split operator is applied on it with default values of all parameters. The split mode parameter is set to 'ordered split' by default. Run the process and compare the results with the explanation of ordered split in the description section of this document. Now change the split mode parameter to 'unordered split' and run the process again. You can understand the results by studying the description of unordered split in the description of this operator.

# Trim




This operator removes leading and trailing spaces from the values of the selected nominal attributes.

## Description

The Trim operator creates new attributes from the selected nominal attributes by removing leading and trailing spaces from the nominal values. The required attributes can be selected through parameters. Please note that this operator only removes leading and trailing spaces from attribute values; spaces between a value are not removed. For example, values ‘ value 1’, ‘value 2 ‘ and ‘ value 3 ‘ will be trimmed to ‘value 1’, ‘value 2’ and ‘value 3’ respectively.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input. The ExampleSet should have at least one nominal attribute because if there is no such attribute, the use of this operator does not make sense.

## Output Ports

**example set output *(exa)*** The values of the selected nominal attributes are trimmed and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel. (Since RapidMiner 6.0.4 the Operator will fail if a selected Attribute is not in the ExampleSet)

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When it is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression** ***(string)*** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '> 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '> 6 && < 11' or '<= 5 || < 0'. But && and || cannot be used together in one numeric condition. Conditions like '(> 0 && < 2) || (>10 && < 12)' are not allowed because they use both && and ||. Use a blank space after '>', '=' and '<' e.g. '<5' will not work, so use '< 5' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

## Tutorial Processes

### Removing leading and trailing spaces from attribute values

This Example Process starts with the Subprocess operator. The operator chain inside the Subprocess operator generates an ExampleSet for this process. The explanation of this inner chain of operators is not relevant here. A breakpoint is inserted here so that you can have a look at the ExampleSet before the application of the Trim operator. You can see that this ExampleSet has two nominal attributes 'att1' and 'att2'. You can see that some values of these attributes have leading and trailing spaces. The Trim operator is applied on this ExampleSet to remove these spaces. All parameters are used with default values. Run the process and compare the resultant ExampleSet with the original ExampleSet. You can clearly see that the leading and trailing spaces have been removed.

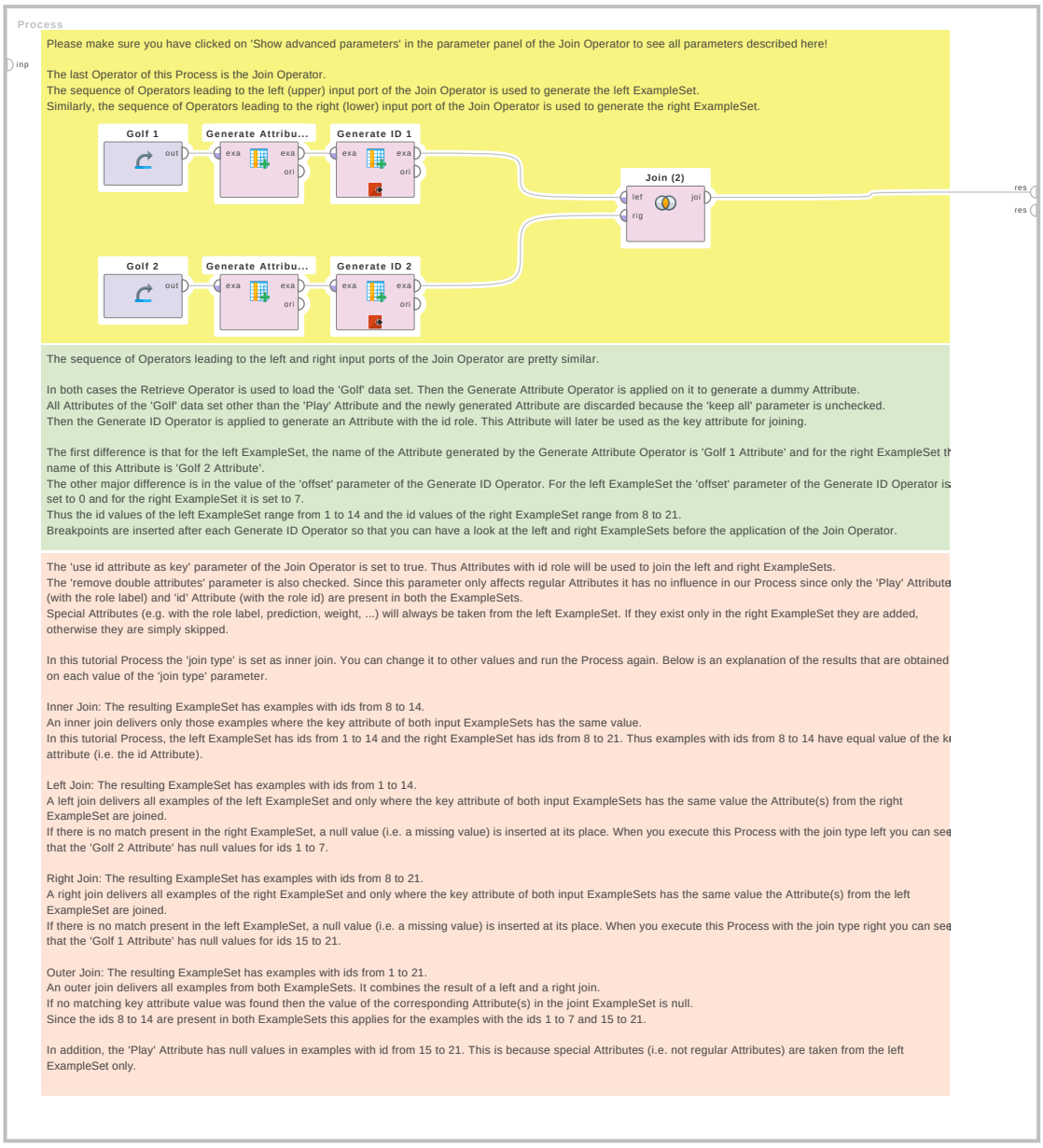



Figure 2.62: Tutorial process ‘Explore the different join types’.

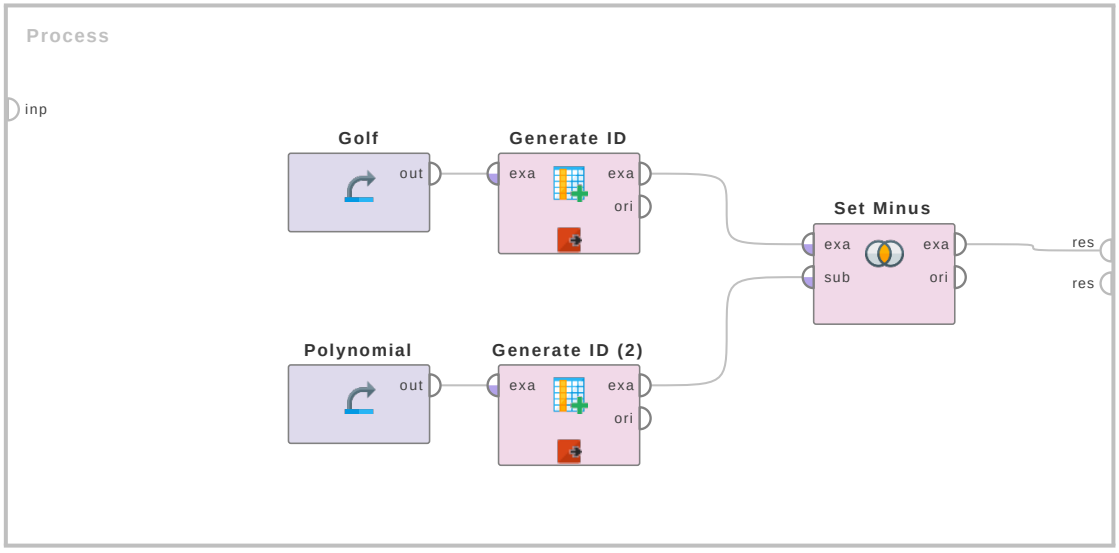



Figure 2.63: Tutorial process ‘Introduction to the Set Minus operator’.

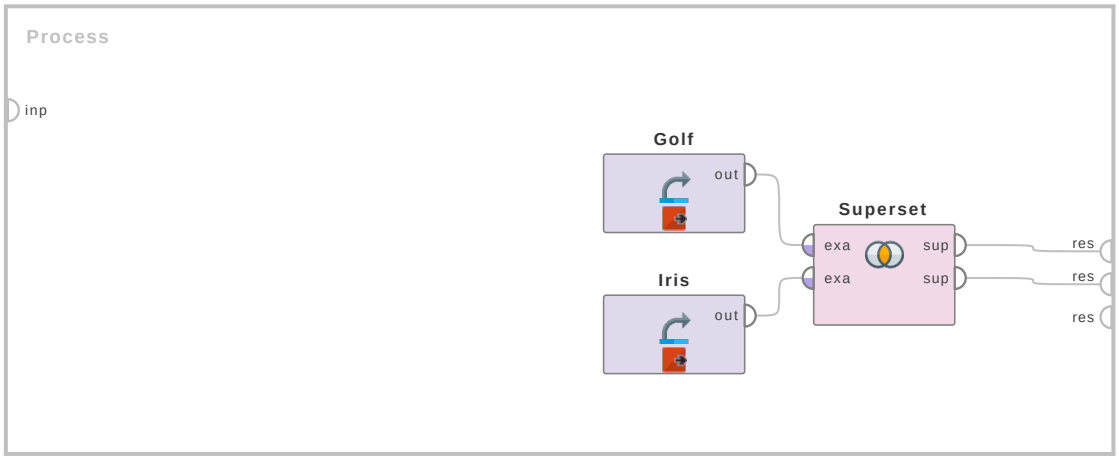



Figure 2.64: Tutorial process ‘Generating supersets of the Golf and Iris data sets’.

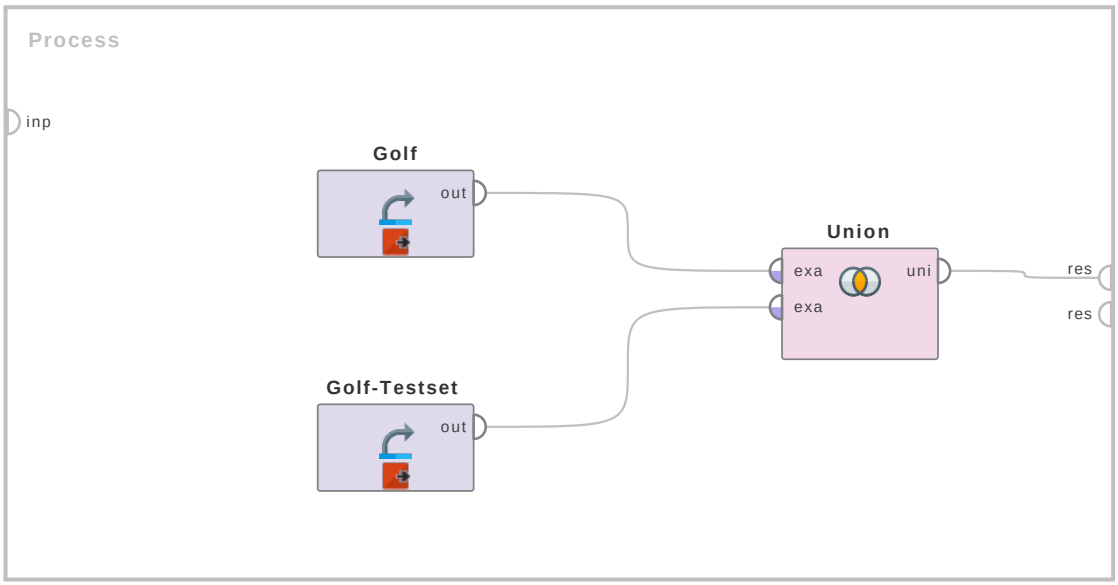



Figure 2.65: Tutorial process ‘Union of the Golf and Golf-Testset data sets’.

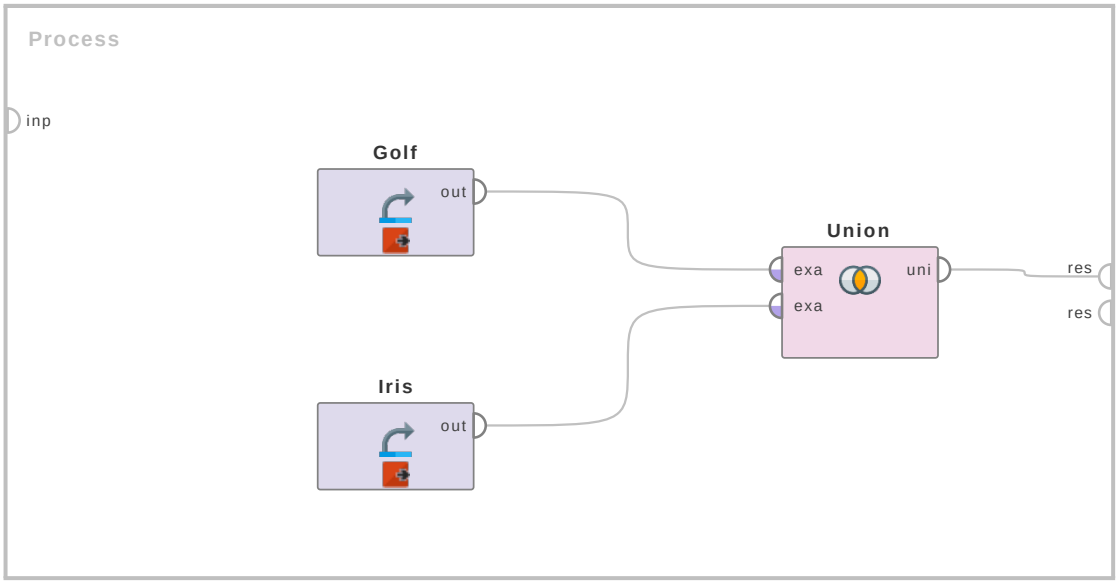



Figure 2.66: Tutorial process ‘Union of the Golf and Iris data sets’.

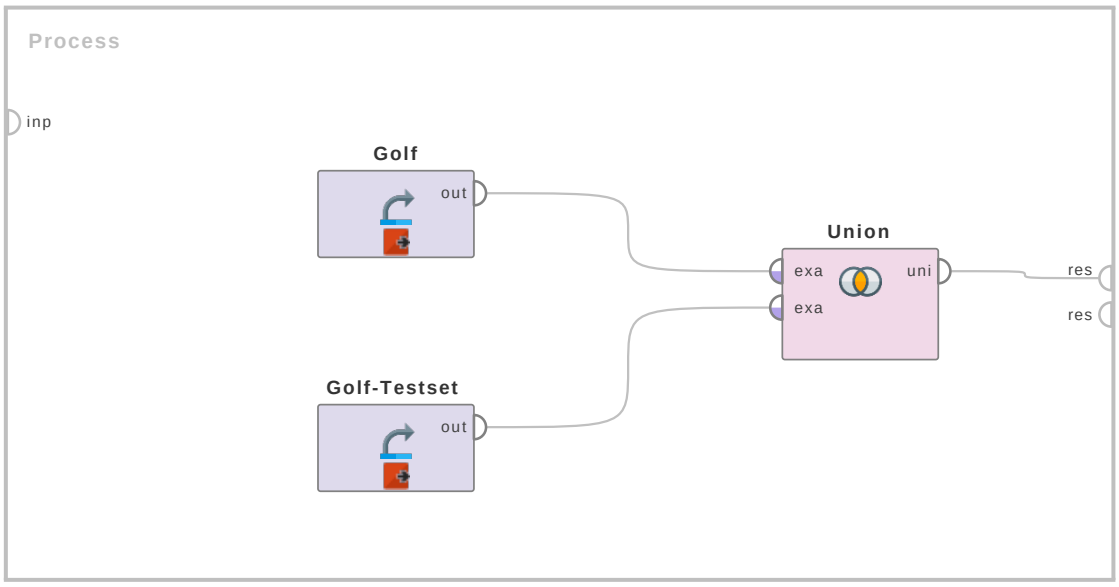



Figure 2.67: Tutorial process ‘Union of the Golf(with id attribute) and Iris data sets’.

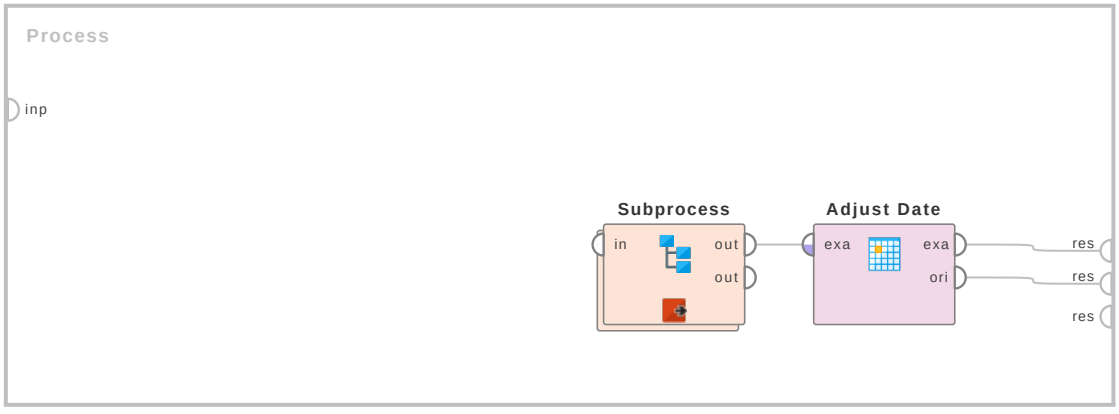



Figure 2.68: Tutorial process ‘Making multiple adjustments in a date attribute’.

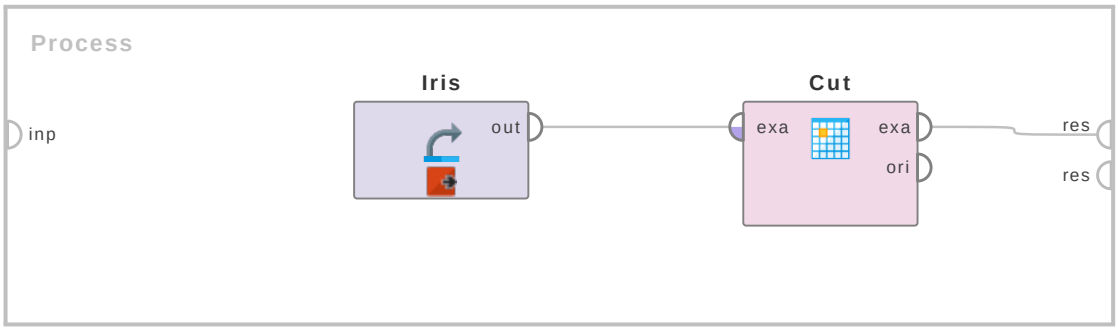



Figure 2.69: Tutorial process ‘Applying the Cut operator on label of the Iris data set’.

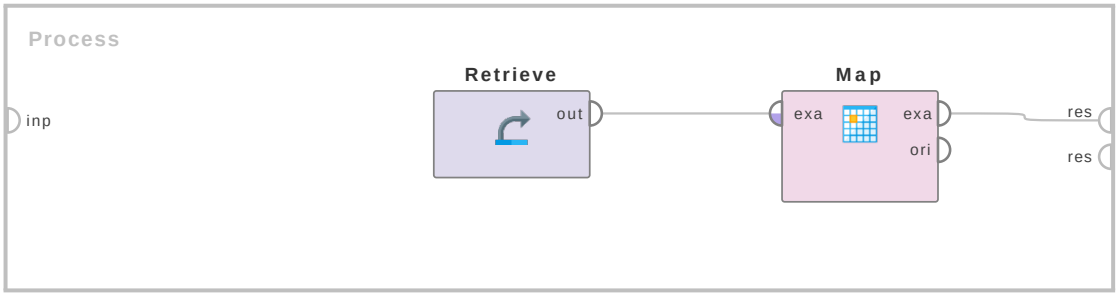



Figure 2.70: Tutorial process ‘Mapping multiple values’.

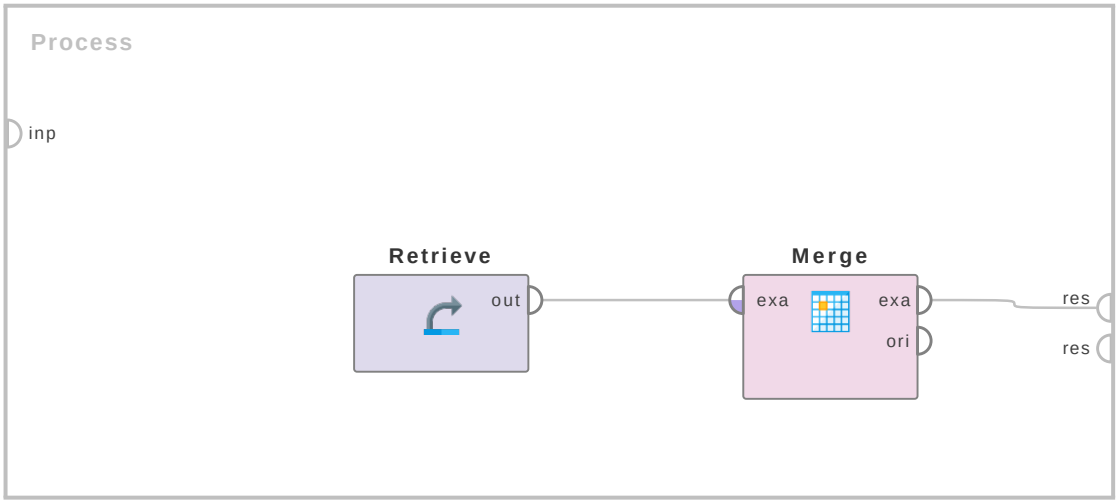



Figure 2.71: Tutorial process ‘Introduction to the Merge operator’.

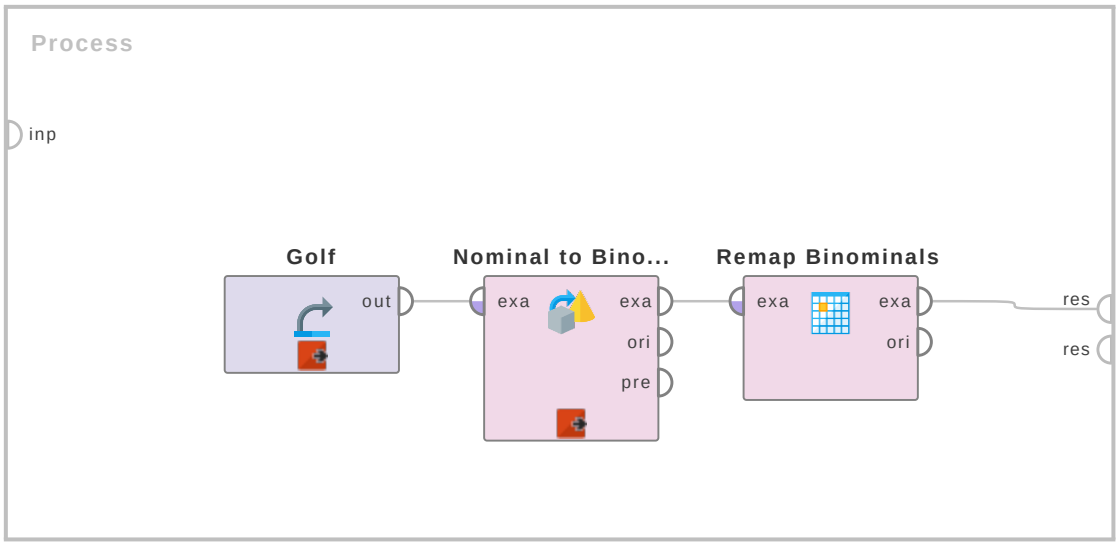



Figure 2.72: Tutorial process ‘Changing mapping of the Wind attribute of the Golf data set’.

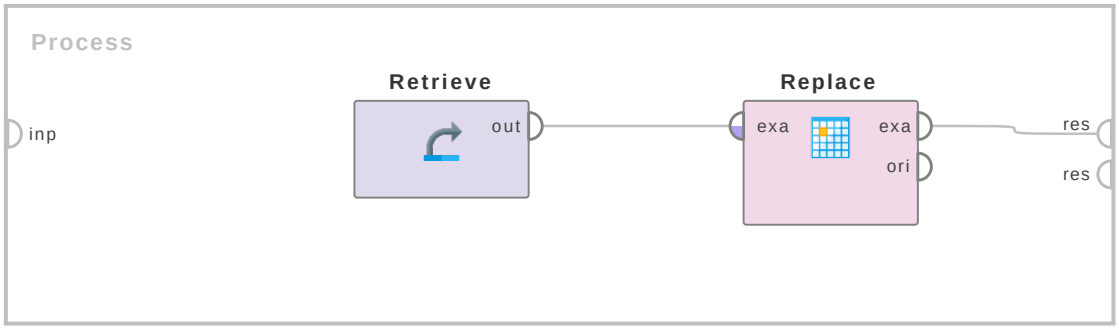



Figure 2.73: Tutorial process ‘Use of replace what and replace by parameters’.

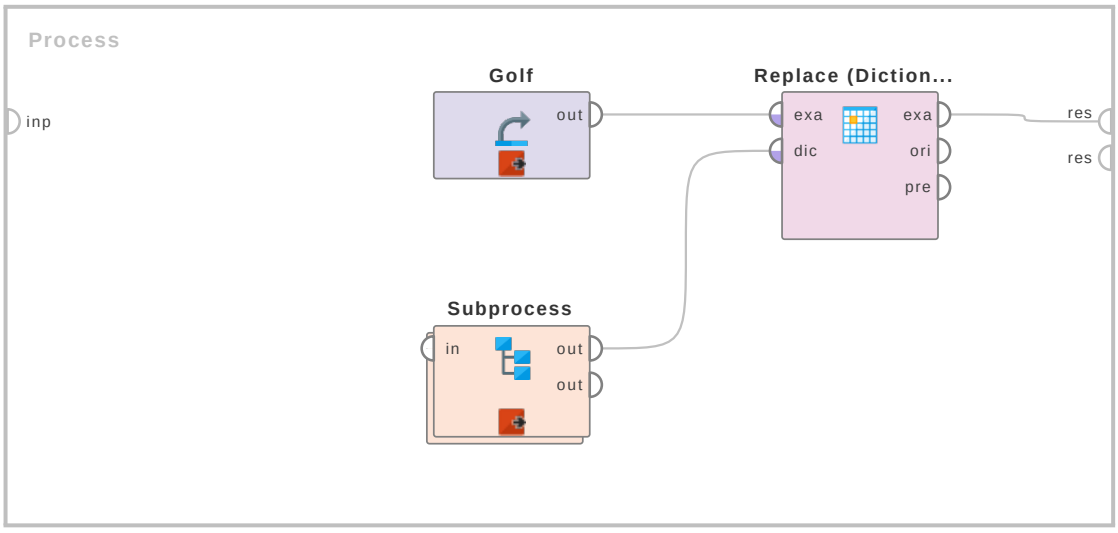



Figure 2.74: Tutorial process ‘Replacing substrings by using a dictionary’.

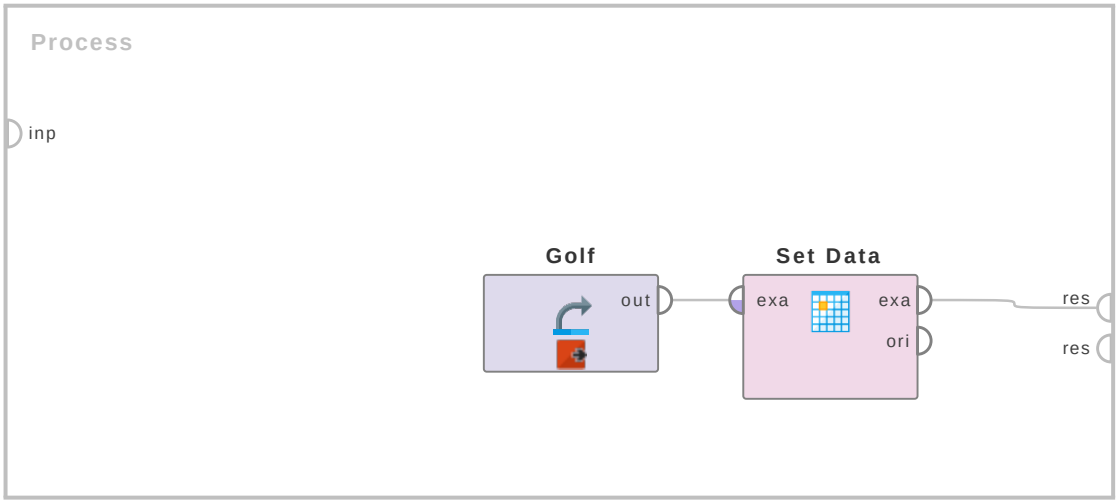



Figure 2.75: Tutorial process ‘Introduction to the Set Data operator’.

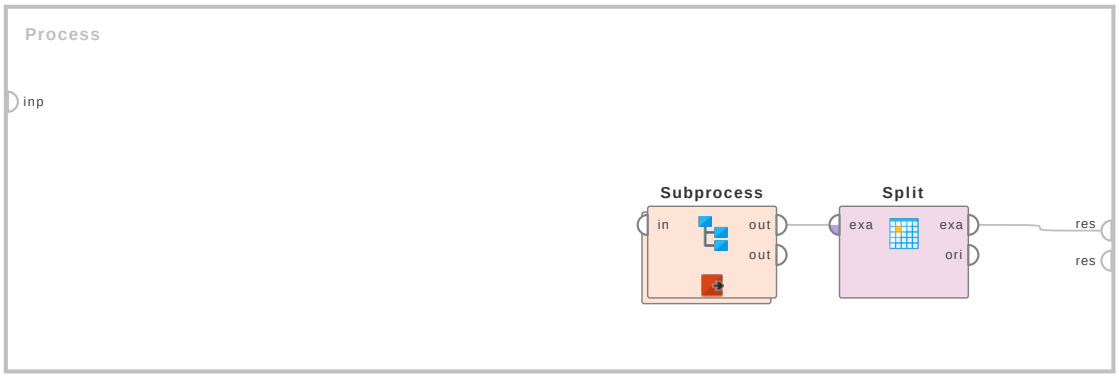



Figure 2.76: Tutorial process ‘Ordered and unordered splits’.

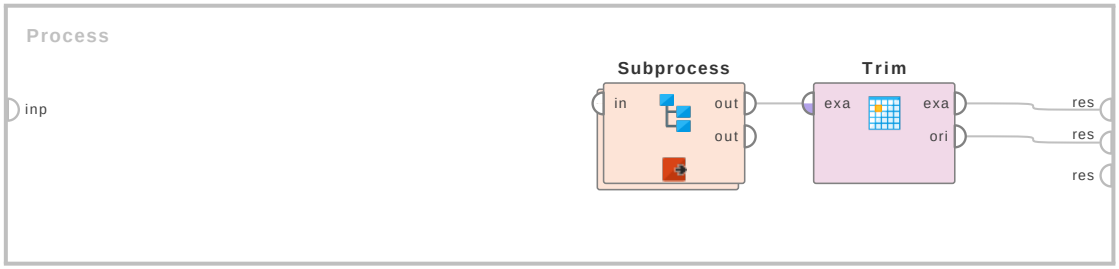
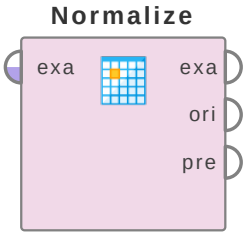



Figure 2.77: Tutorial process ‘Removing leading and trailing spaces from attribute values’.

# 3Cleansing

## 3.1 Normalization

### Normalize




This Operator normalizes the values of the selected Attributes.

#### Description

Normalization is used to scale values so they fit in a specific range. Adjusting the value range is very important when dealing with Attributes of different units and scales. For example, when using the Euclidean distance all Attributes should have the same scale for a fair comparison. Normalization is useful to compare Attributes that vary in size. This Operator performs normalization of the selected Attributes. Four normalization methods are provided. These methods are explained in the parameters.

#### Differentiation

- **Scale by Weights**

  This Operator can be used to scale Attributes by pre-calculated weights. Instead of adjusting the value range to a common scale, this Operator can be used to give important Attributes even more weight.

  See page 332 for details.

- **De-Normalize**

  This Operator can be used to revert a previously applied normalization. It requires the preprocessing model returned by a Normalization Operator.

  See page **??** for details.

#### Input Ports

**example set *(exa)*** This input port expects an ExampleSet.

#### Output Ports

**example set *(exa)*** The ExampleSet with the selected Attributes in normalized form is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed through without changes.

**preprocessing model *(pre)*** This port delivers the preprocessing model. It can be used by the Apply Model Operator to perform the specified normalization on another ExampleSet. This is helpful for example if the normalization is used during training and the same transformation has to be applied on test or actual data.

The preprocessing model can also be grouped together with other preprocessing models and learning models by the Group Models Operator.

## Parameters

**create view** Create a View instead of changing the underlying data. If this option is checked, the normalization is delayed until the transformations are needed. This parameter can be considered a legacy option.

**attribute filter type** This parameter allows you to select the Attribute selection filter; the method you want to use for selecting Attributes. It has the following options:

- **all** This option selects all the Attributes of the ExampleSet, so that no Attributes are removed. This is the default option.
- **single** This option allows the selection of a single Attribute. The required Attribute is selected by the *attribute* parameter.
- **subset** This option allows the selection of multiple Attributes through a list (see parameter *attributes*). If the meta data of the ExampleSet is known, all Attributes are present in the list and the required ones can easily be selected.
- **regular_expression** This option allows you to specify a regular expression for the Attribute selection. The regular expression filter is configured by the parameters *regular expression, use except expression and except expression*.
- **value_type** This option allows selection of all the Attributes of a particular type. It should be noted that types are hierarchical. For example, both real and integer types belong to the numeric type. The value type filter is configured by the parameters *value type, use value type exception, except value type*.
- **block_type** This option allows the selection of all the Attributes of a particular block type. It should be noted that block types may be hierarchical. For example, value_series_start and value_series_end block types both belong to the value_series block type. The block type filter is configured by the parameters *block type, use block type exception, except block type*.
- **no_missing_values** This option selects all Attributes of the ExampleSet, which do not contain a missing value in any Example. Attributes that have even a single missing value are removed.
- **numeric_value_filter** All numeric Attributes whose Examples all match a given numeric condition are selected. The condition is specified by the *numeric condition* parameter. Please note that all nominal Attributes are also selected irrespective of the given numerical condition.

**attribute** The required Attribute can be selected from this option. The Attribute name can be selected from the drop down box of the parameter if the meta data is known.

**attributes** The required Attributes can be selected from this option. This opens a new window with two lists. All Attributes are present in the left list. They can be shifted to the right list, which is the list of selected Attributes that will make it to the output port.

**regular expression** Attributes whose names match this expression will be selected. The expression can be specified through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression** If enabled, an exception to the first regular expression can be specified. This exception is specified by the *except regular expression* parameter.

**except regular expression** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in *regular expression* parameter).

**value type** This option allows to select a type of Attribute. One of the following types can be chosen: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date, time.

**use value type exception** If enabled, an exception to the selected type can be specified. This exception is specified by the *except value type* parameter.

**except value type** The Attributes matching this type will be removed from the final output even if they matched the type selected before, specified by the *value type* parameter. One of the following types can be selected here: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date and time.

**block type** This option allows to select a block type of Attribute. One of the following types can be chosen: single_value, value_series, value_series_start, value_series_end, value_matrix, value_matrix_start, value_matrix_end and value_matrix_row_start.

**use block type exception** If enabled, an exception to the selected block type can be specified. This exception is specified by the *except block type* parameter.

**except block type** The Attributes matching this block type will be removed from the final output even if they matched the type selected before by the *block type* parameter. One of the following block types can be selected here: single_value, value_series, value_series_start, value_series_end, value_matrix, value_matrix_start, value_matrix_end and value_matrix_row_start.

**numeric condition** The numeric condition used by the numeric condition filter type. A numeric Attribute is kept if all Examples match the specified condition for this Attribute. For example, the numeric condition '$> 6$' will keep all numeric Attributes having a value of greater than 6 in every Example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Nominal Attributes are always kept, regardless of the specified numeric condition.

**invert selection** If this parameter is set to true, the selection is reversed. In this case, all Attributes matching the specified condition are removed and the other Attributes remain in the output ExampleSet. Special Attributes are kept independent of the *invert selection* parameter as along as the *include special attributes* parameter is not set to true. If so, the condition is also applied to the special Attributes and the selection is reversed if this parameter is checked.

**include special attributes** Special Attributes are Attributes with special roles. These are: id, label, prediction, cluster, weight and batch. Also custom roles can be assigned to Attributes.

By default, all special Attributes are delivered to the output port irrespective of the conditions in the Select Attributes Operator. If this parameter is set to true, special Attributes are also tested against conditions specified in the Select Attributes Operator and only those Attributes are selected that match the conditions.

**method** Four methods are provided here for normalizing data. These methods are also explained in the attached tutorial Process.

- **z_transformation** This is also called statistical normalization. This normalization subtracts the mean of the data from all values and then divides them by the standard deviation. Afterwards, the distribution of the data has a mean of zero and a variance of one. This is a common and very useful normalization technique. It preserves the original distribution of the data and is less influenced by outliers.
- **range_transformation** Range transformation normalizes all Attribute values to a specified value range. When this method is selected, two other parameters (min, max) appear in the Parameters panel. So the largest value is set to 'max' and the smallest value is set to 'min'. All other values are scaled, so they fit into the given range. This method can be influenced by outliers, because the bounds move towards them. On the other hand, this method keeps the original distribution of the data points, so it can also be used for data anonymization, for example to obfuscate the true range of observations.
- **proportion_transformation** This normalization is based on the proportion each Attribute value has on the complete Attribute. This means each value is divided by the total sum of that Attribute values. The sum is only formed from finite values, ignoring NaN/missing values as well as positive and negative infinity. When this method is selected, another parameter (allow negative values) appears in the Parameters panel. If checked, negative values will be treated as absolute values, otherwise they will produce an error when executed.
- **interquartile_range** Normalization is performed using the interquartile range. The interquartile range is the distance between the 25th and 75th percentile, which are also called lower and upper quartile, or Q1 and Q3. They are calculated by first sorting the data and then taking the data value that separates the first (or the last) 25% of the Examples from the rest. The median is the 50th percentile, so it is the value that separates the sorted values in half. The interquartile range (IQR) is the difference between Q3 and Q1. The final formula for the interquartile range normalization is then: (value median) / IQR The IQR is the range between the middle 50% of the data, so this normalization method is less influenced by outliers. NaN/missing values, as well as infinite values will be ignored for this method. Also, if no finite values could be found, the corresponding Attribute will be ignored.

**min** This parameter is available only when the *method* parameter is set to 'range transformation'. It is used to specify the minimum point of the range.

**max** This parameter is available only when the *method* parameter is set to 'range transformation'. It is used to specify the maximum point of the range.

**allow negative values** This parameter is available only when the *method* parameter is set to 'proportion transformation'. It is used to allow or disallow negative values in the processed Attributes. Negative values then will be counted as their absolute values.

## Tutorial Processes

### Normalizing Age and Passenger Fare for the Titanic data

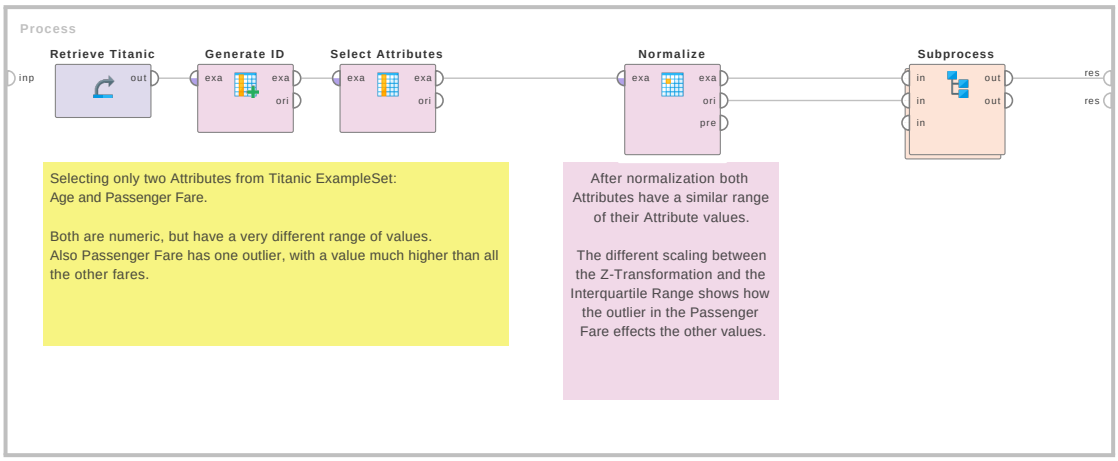
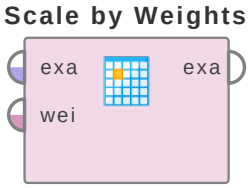



Figure 3.1: Tutorial process ‘Normalizing Age and Passenger Fare for the Titanic data’.

This tutorial Process takes the Age and the Passenger Fare Attributes from the Titanic data and performs a normalization on them. The Attributes have a very different range of values (the highest Age is 80 and the highest fare is around 500). Also, the Passenger Fare has one value that is much higher than all the other fares. So it can be considered as an outlier. When applying the Z-Transformation, both Attributes are centered around 0. When changing the method to Interquartile Range, the values of the Passenger Fare are spread out a bit more evenly, as the one outlier does not have so much influence. For visualization, it is best to use the Histogram charts view.

# Scale by Weights




This operator scales the input ExampleSet according to the given weights. This operator deselects attributes with weight 0 and calculates new values for numeric attributes according to the given weights.

## Description

The Scale by Weights operator selects attributes with non zero weight. The values of the remaining numeric attributes are recalculated based on the weights delivered at the weights input port. The new values of numeric attributes are calculated by multiplying the original values by the weight of that attribute. This operator can hardly be used for selecting a subset of attributes according to weights determined by a former weighting scheme. For this purpose the Select by Weights operator should be used which selects only those attributes that fulfill a specified weight relation.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Weight by Chi Squared Statistic operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.

**weights *(wei)*** This port expects the attribute weights. There are numerous operators that provide the attribute weights. The Weight by Chi Squared Statistic operator is used in the Example Process.

## Output Ports

**example set *(exa)*** The attributes with weight 0 are removed from the input ExampleSet. The values of the remaining numeric attributes are recalculated based on the weights provided at the weights input port. The resultant ExampleSet is delivered through this port.

## Tutorial Processes

### Applying the Scale by Weights operator on the Golf data set

The 'Golf' data set is loaded using the Retrieve operator. The Weight by Chi Squared Statistic operator is applied on it to generate attribute weights. A breakpoint is inserted here. You can see the attributes with their weights here. You can see that the Wind, Humidity, Outlook and Temperature attributes have weights 0, 0.438, 0.450 and 1 respectively. The Scale by Weights operator is applied next. The 'Golf' data set is provided at the example set input port and weights calculated by the Weight by Chi Squared Statistic operator are provided at the weights input port. The Scale by Weights operator removes the attributes with weight 0 i.e. the Wind attribute is removed. The values of the remaining numeric attributes (i.e. the Temperature and Humidity attribute) are recalculated based on their weights. The weight of the Temperature attribute is 1 thus its values remain unchanged. The weight of the Humidity attribute is 0.438 thus its new

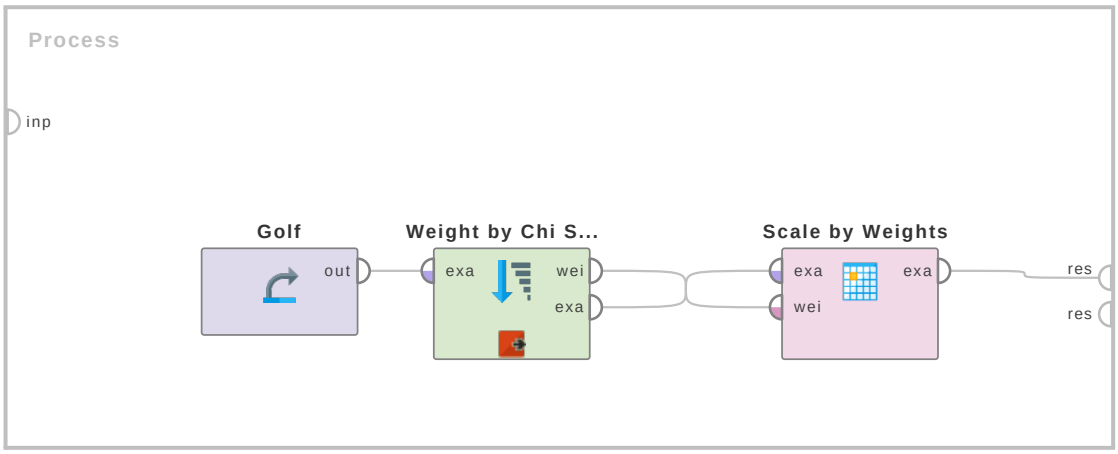
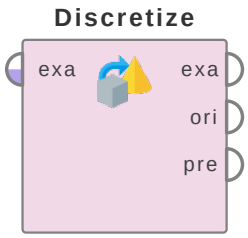



Figure 3.2: Tutorial process ‘Applying the Scale by Weights operator on the Golf data set’.

values are calculated by multiplying the original values by 0.438. This can be verified by viewing the results in the Results Workspace.

# 3.2 Binning

## Discretize by Binning




This operator discretizes the selected numerical attributes into user-specified number of bins. Bins of equal range are automatically generated, the number of the values in different bins may vary.

### Description

This operator discretizes the selected numerical attributes to nominal attributes. The *number of bins* parameter is used to specify the required number of bins. This discretization is performed by simple binning. The range of numerical values is partitioned into segments of equal size. Each segment represents a bin. Numerical values are assigned to the bin representing the segment covering the numerical value. Each range is named automatically. The naming format for range can be changed using the *range name type* parameter. Values falling in the range of a bin are named according to the name of that range. This operator also allows you to apply binning only on a range of values. This can be enabled by using the *define boundaries* parameter. The *min value* and *max value* parameter are used for defining the boundaries of the range. If there are any values that are less than the *min value* parameter, a separate range is created for them. Similarly if there are any values that are greater than the *max value* parameter, a separate range is created for them. Then, the discretization by binning is performed only on the values that are within the specified boundaries.

### Differentiation

- **Discretize by Frequency** The Discretize By Frequency operator creates bins in such a way that the number of unique values in all bins are (almost) equal. See page 343 for details.
- **Discretize by Size** The Discretize By Size operator creates bins in such a way that each bin has user-specified size (i.e. number of examples). See page 348 for details.
- **Discretize by Entropy** The discretization is performed by selecting bin boundaries such that the entropy is minimized in the induced partitions. See page 339 for details.
- **Discretize by User Specification** This operator discretizes the selected numerical attributes into user-specified classes. See page 352 for details.

### Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. the output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with the data. Note that there should be at least one numerical attribute in the input ExampleSet, otherwise the use of this operator does not make sense.

## Output Ports

**example set *(exa)*** The selected numerical attributes are converted into nominal attributes by binning and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model *(pre)*** This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

## Parameters

**create view *(boolean)*** It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples

all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *parameter* attribute if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list, which is the list of selected attributes.

**regular expression** ***(string)*** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression** ***(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter *(except regular expression*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will not be selected even if they match the previously mentioned type i.e. *value type* parameter's value.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will not be selected even if they match the previously mentioned block type i.e. *block type* parameter's value.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '($> 0$ && $< 2$) || ($>10$ && $< 12$)' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Select Attribute operator.

If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**number of bins** ***(integer)*** This parameter specifies the number of bins which should be used for each attribute.

**define boundaries:** ***(boolean)*** The Discretize by Binning operator allows you to apply binning only on a range of values. This can be enabled by using the *define boundaries* parameter. If this is set to true, discretization by binning is performed only on the values that are within the specified boundaries. The lower and upper limit of the boundary is specified by the *min value* and *max value* parameters respectively.

**min value** ***(real)*** This parameter is only available when the *define boundaries* parameter is set to true. It is used to specify the lower limit value for the binning range.

**max value** ***(real)*** This parameter is only available when the *define boundaries* parameter is set to true. It is used to specify the upper limit value for the binning range.

**range name type** ***(selection)*** This parameter is used to change the naming format for range. 'long', 'short' and 'interval' formats are available.

**automatic number of digits** ***(boolean)*** This is an expert parameter. It is only available when the *range name type* parameter is set to 'interval'. It indicates if the number of digits should be automatically determined for the range names.

**number of digits** ***(integer)*** This is an expert parameter. It is used to specify the minimum number of digits used for the interval names.

## Related Documents

- **Discretize by Frequency** (page 343)
- **Discretize by Size** (page 348)
- **Discretize by Entropy** (page 339)
- **Discretize by User Specification** (page 352)

## Tutorial Processes

### Discretizing numerical attributes of the 'Golf' data set by Binning

The focus of this Example Process is the binning procedure. For understanding the parameters related to attribute selection please study the Example Process of the Select Attributes operator.

The 'Golf' data set is loaded using the Retrieve operator. The Discretize by Binning operator is applied on it. The 'Temperature' and 'Humidity' attributes are selected for discretization. The number of bins parameter is set to 2. The define boundaries parameter is set to true. The min value and max value parameters are set to 70 and 80 respectively. Thus binning will be performed

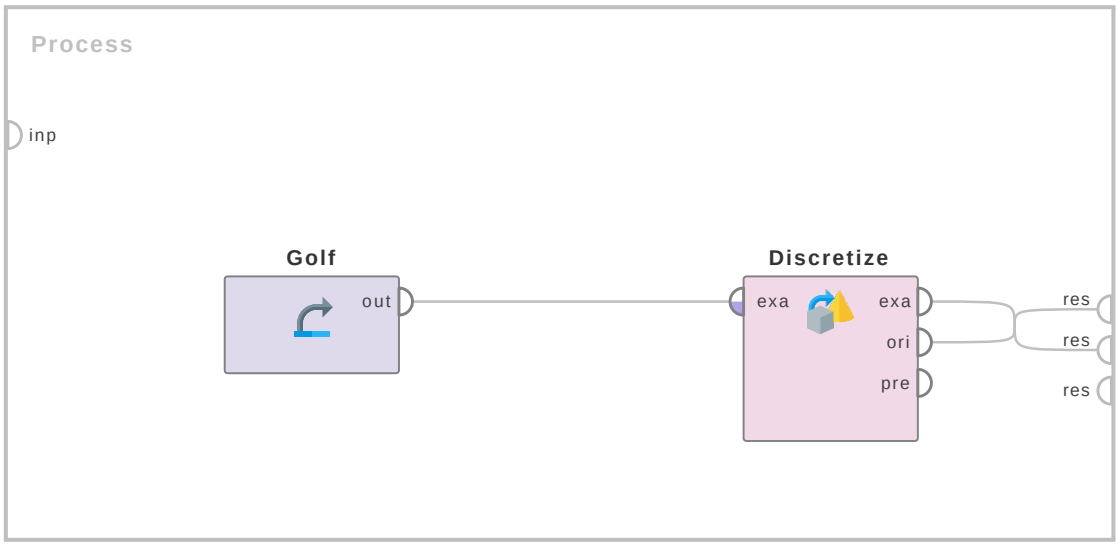
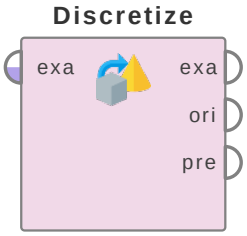



Figure 3.3: Tutorial process ‘Discretizing numerical attributes of the ’Golf’ data set by Binning’.

only in the range from 70 to 80. As the number of bins parameter is set to 2, the range will be divided into two equal segments. Approximately speaking, the first segment of the range will be from 70 to 75 and the second segment of the range will be from 76 to 80. These are not exact values, but they are good enough for the explanation of this process. There will be a separate range for all those values that are less than the min value parameter i.e. less than 70. This range is automatically named ‘range1’. The first and second segment of the binning range are named ‘range2’ and ‘range3’ respectively. There will be a separate range for all those values that are greater than the max value parameter i.e. greater than 80. This range is automatically named ‘range4’. Run the process and compare the original data set with the discretized one. You can see that the values less than or equal to 70 in the original data set are named ‘range1’ in the discretized data set. The values greater than 70 and less than or equal to 75 in the original data set are named ‘range2’ in the discretized data set. The values greater than 75 and less than or equal to 80 in the original data set are named ‘range3’ in the discretized data set. The values greater than 80 in the original data set are named ‘range4’ in the discretized data set.

# Discretize by Entropy




This operator converts the selected numerical attributes into nominal attributes. The boundaries of the bins are chosen so that the entropy is minimized in the induced partitions.

## Description

This operator discretizes the selected numerical attributes to nominal attributes. The discretization is performed by selecting a bin boundary that minimizes the entropy in the induced partitions. Each bin range is named automatically. The naming format of the range can be changed using the *range name type* parameter. The values falling in the range of a bin are named according to the name of that range.

The discretization is performed by selecting a bin boundary that minimizes the entropy in the induced partitions. The method is then applied recursively for both new partitions until the stopping criterion is reached. For more information please study:

- Multi-interval discretization of continued-values attributes for classification learning (Fayyad,Irani)
- Supervised and Unsupervised Discretization (Dougherty,Kohavi,Sahami).

This operator can automatically remove all attributes with only one range i.e. those attributes which are not actually discretized since the entropy criterion is not fulfilled. This behavior can be controlled by the *remove useless* parameter.

## Differentiation

- **Discretize by Binning** The Discretize By Binning operator creates bins in such a way that the range of all bins is (almost) equal. See page 334 for details.
- **Discretize by Frequency** The Discretize By Frequency operator creates bins in such a way that the number of unique values in all bins are (almost) equal. See page 343 for details.
- **Discretize by Size** The Discretize By Size operator creates bins in such a way that each bin has user-specified size (i.e. number of examples). See page 348 for details.
- **Discretize by User Specification** This operator discretizes the selected numerical attributes into user-specified classes. See page 352 for details.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. Please note that there should be at least one numerical attribute in the input ExampleSet, otherwise the use of this operator does not make sense.

## Output Ports

**example set output *(exa)*** The selected numerical attributes are converted into nominal attributes by discretization and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model *(pre)*** This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

## Parameters

**create view *(boolean)*** It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute *(string)*** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes *(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression *(string)*** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression *(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except regular expression *(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type *(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception *(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type *(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type *(selection)*** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception *(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type *(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition *(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**remove useless** ***(boolean)*** This parameter indicates if the useless attributes, i.e. attributes containing only a single range, should be removed. If this parameter is set to true then all those attributes that are not actually discretized since the entropy criterion is not fulfilled are removed.

**range name type** ***(selection)*** This parameter is used for changing the naming format for range. 'long', 'short' and 'interval' formats are available.

**automatic number of digits** ***(boolean)*** This is an expert parameter. It is only available when the *range name type* parameter is set to 'interval'. It indicates if the number of digits should be automatically determined for the range names.

**number of digits** ***(integer)*** This is an expert parameter. It is used to specify the minimum number of digits used for the interval names.

## Related Documents

- **Discretize by Binning** (page 334)
- **Discretize by Frequency** (page 343)
- **Discretize by Size** (page 348)
- **Discretize by User Specification** (page 352)

## Tutorial Processes

### Discretizing the 'Sonar' data set by entropy

The focus of this Example Process is the discretization procedure. For understanding the parameters related to attribute selection please study the Example Process of the Select Attributes operator.

The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can gave a look at the ExampleSet. You can see that this data set has 60 regular attributes (all of real type). The Discretize by Entropy operator is applied on it. The attribute filter type parameter is set to 'all', thus all the numerical attributes will be discretized.The remove useless parameter is set to true, thus attributes with only one range are removed from the ExampleSet. Run the process and switch to the Results Workspace. You can see that the 'Sonar' data set has been reduced to just 22 regular attributes. These numerical attributes have been discretized to nominal attributes.

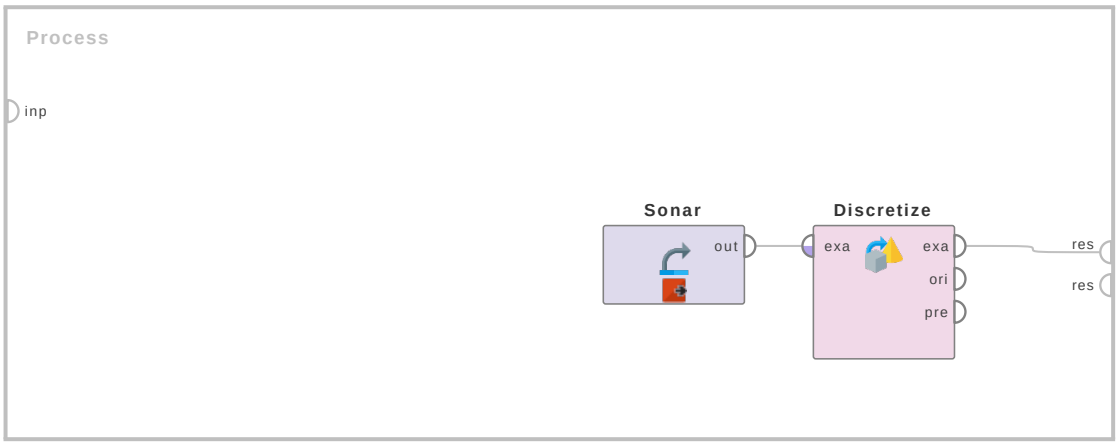
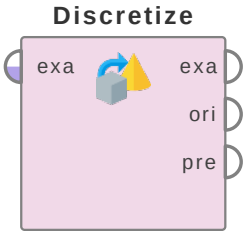



Figure 3.4: Tutorial process 'Discretizing the 'Sonar' data set by entropy'.

## Discretize by Frequency




This operator converts the selected numerical attributes into nominal attributes by discretizing the numerical attribute into a user-specified number of bins. Bins of equal frequency are automatically generated, the range of different bins may vary.

### Description

This operator discretizes the selected numerical attributes to nominal attributes. The *number of bins* parameter is used to specify the required number of bins. The number of bins can also be specified by using the *use sqrt of examples* parameter. If the *use sqrt of examples* parameter is set to true, then the number of bins is calculated as the square root of the number of examples with non-missing values (calculated for every single attribute). This discretization is performed by equal frequency binning i.e. the thresholds of all bins is selected in a way that all bins contain the same number of numerical values. Numerical values are assigned to the bin representing the range segment covering the numerical value. Each range is named automatically. The naming format for the range can be changed using the *range name type* parameter. Values falling in the range of a bin are named according to the name of that range.

Other discretization operators are also available in RapidMiner. The Discretize By Frequency operator creates bins in such a way that the number of unique values in all bins are (almost) equal. In contrast, the Discretize By Binning operator creates bins in such a way that the range of all bins is (almost) equal.

### Differentiation

- **Discretize by Binning** The Discretize By Binning operator creates bins in such a way that the range of all bins is (almost) equal. See page 334 for details.

- **Discretize by Size** The Discretize By Size operator creates bins in such a way that each bin has user-specified size (i.e. number of examples). See page 348 for details.
- **Discretize by Entropy** The discretization is performed by selecting bin boundaries such that the entropy is minimized in the induced partitions. See page 339 for details.
- **Discretize by User Specification** This operator discretizes the selected numerical attributes into user-specified classes. See page 352 for details.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in attached Example Process. The output of other operators can also be used as input. Please note that there should be at least one numerical attribute in the input ExampleSet, otherwise use of this operator does not make sense.

## Output Ports

**example set *(exa)*** The selected numerical attributes are converted into nominal attributes by discretization (frequency) and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model *(pre)*** This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

## Parameters

**create view *(boolean)*** It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *parameter* attribute if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list, which is the list of selected attributes.

**regular expression** ***(string)*** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression** ***(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter *(except regular expression*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will not be selected even if they match the previously mentioned type i.e. *value type* parameter's value.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be not be selected even if they match the previously mentioned block type i.e. *block type* parameter's value.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '>', '=' and '<' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**use sqrt of examples** ***(boolean)*** If set to true, the number of bins is determined by the square root of the number of non-missing values instead of using the *number of bins* parameter.

**number of bins** ***(integer)*** This parameter is available only when the *use sqrt of examples* parameter is not set to true. This parameter specifies the number of bins which should be used for each attribute.

**range name type** ***(selection)*** This parameter is used for changing the naming format for range. 'long', 'short' and 'interval' formats are available.

**automatic number of digits** ***(boolean)*** This is an expert parameter. It is only available when the *range name type* parameter is set to 'interval'. It indicates if the number of digits should be automatically determined for the range names.

**number of digits** ***(integer)*** This is an expert parameter. It is used to specify the minimum number of digits used for the interval names.

## Related Documents

- **Discretize by Binning** (page 334)
- **Discretize by Size** (page 348)
- **Discretize by Entropy** (page 339)
- **Discretize by User Specification** (page 352)

## Tutorial Processes

### Discretizing the Temperature attribute of the 'Golf' data set by Frequency

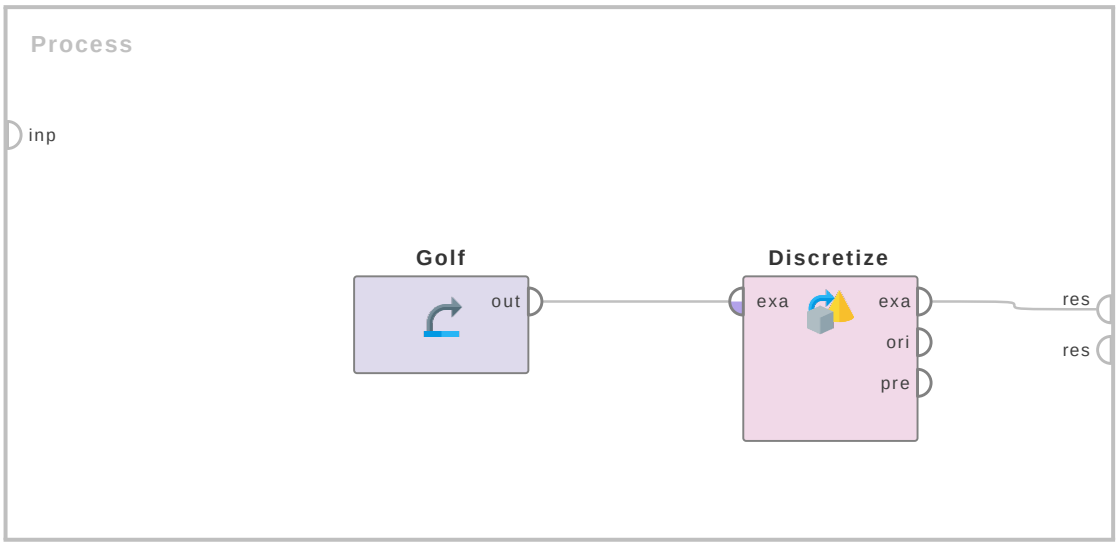
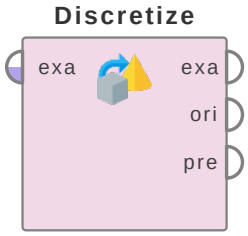



Figure 3.5: Tutorial process 'Discretizing the Temperature attribute of the 'Golf' data set by Frequency'.

The focus of this Example Process is the discretization (by frequency) procedure. For understanding the parameters related to attribute selection please study the Example Process of the Select Attributes operator.

The 'Golf' data set is loaded using the Retrieve operator. The Discretize by Frequency operator is applied on it. The 'Temperature' attribute is selected for discretization. The number of bins parameter is set to 3. Run the process and switch to the Results Workspace. You can see that the 'Temperature' attribute has been changed from numerical to nominal form. The values of the 'Temperature' attribute have been divided into three ranges. Each range has an equal number of unique values. You can see that 'range1' and 'range3' have 4 examples while the 'range2' has 6 examples. But in 'range2' the 'Temperature' values 72 and 75 occur twice. Thus essentially 4 unique numerical values are present in 'range2'.

# Discretize by Size




This operator converts the selected numerical attributes into nominal attributes by discretizing the numerical attribute into bins of user-specified size. Thus each bin contains a user-defined number of examples.

## Description

This operator discretizes the selected numerical attributes to nominal attributes. The *size of bins* parameter is used for specifying the required size of bins. This discretization is performed by binning examples into bins containing the same, user-specified number of examples. Each bin range is named automatically. The naming format of the range can be changed by using the *range name type* parameter. The values falling in the range of a bin are named according to the name of that range.

It should be noted that if the number of examples is not evenly divisible by the requested number of examples per bin, the actual result may slightly differ from the requested bin size. Similarly, if a range of examples cannot be split, because the numerical values are identical within this set, only all or none can be assigned to a bin. This may lead to further deviations from the requested bin size.

This operator is closely related to the Discretize By Frequency operator. There you have to specify the number of bins you need (say $x$) and the operator automatically creates it with an almost equal number of values (i.e. $n/x$ values where $n$ is the total number of values). In the Discretize by Size operator you have to specify the number of values you need in each bin (say $y$) and the operator automatically creates $n/y$ bins with $y$ values.

## Differentiation

- **Discretize by Binning** The Discretize By Binning operator creates bins so their range is (almost) equal. See page 334 for details.
- **Discretize by Frequency** The Discretize By Frequency operator creates bins so the number of unique values in all bins are (almost) equal. See page 343 for details.
- **Discretize by Entropy** The discretization is performed by selecting bin boundaries so the entropy is minimized in the induced partitions. See page 339 for details.
- **Discretize by User Specification** This operator discretizes the selected numerical attributes into user-specified classes. See page 352 for details.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. Please note that there should be at least one numerical attribute in the input ExampleSet, otherwise the use of this operator does not make sense.

## Output Ports

**example set output *(exa)*** The selected numerical attributes are converted into nominal attributes by discretization and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model *(pre)*** This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

## Parameters

**create view *(boolean)*** It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value type* option. It allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression** ***(string)*** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**size of bins** ***(integer)*** This parameter specifies the required size of bins i.e. number of examples contained in a bin.

**sorting direction** ***(selection)*** This parameter indicates if the values should be sorted in increasing or decreasing order.

**range name type** ***(selection)*** This parameter is used for changing the naming format for range. 'long', 'short' and 'interval' formats are available.

**automatic number of digits** ***(boolean)*** This is an expert parameter. It is only available when the *range name type* parameter is set to 'interval'. It indicates if the number of digits should be automatically determined for the range names.

**number of digits** ***(integer)*** This is an expert parameter. It is used to specify the minimum number of digits used for the interval names.

## Related Documents

- **Discretize by Binning** (page 334)
- **Discretize by Frequency** (page 343)
- **Discretize by Entropy** (page 339)
- **Discretize by User Specification** (page 352)

## Tutorial Processes

### Discretizing the Temperature attribute of the 'Golf' data set

The focus of this Example Process is the discretization procedure. For understanding the parameters related to attribute selection please study the Example Process of the Select Attributes operator.

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can gave a look at the ExampleSet. You can see that the 'Temperature' attribute is a numerical attribute. The Discretize by Size operator is applied on it. The 'Temperature' attribute is selected for discretization. The size of bins parameter is set to 5. Run the process and switch to the Results Workspace. You can see that the 'Temperature' attribute has been changed from numerical to nominal form. The values of the 'Temperature' attribute have been divided into three ranges. Each range has an equal number of unique values. You can see that 'range1' and 'range3' have 4 examples while the 'range2' has 6 examples. All bins do not have exactly equal values because 14 examples cannot be grouped by 5 examples per bin. But in 'range2' the 'Temperature' values 72 and 75 occur twice. Thus essentially 4 unique numerical values are present in 'range2'.

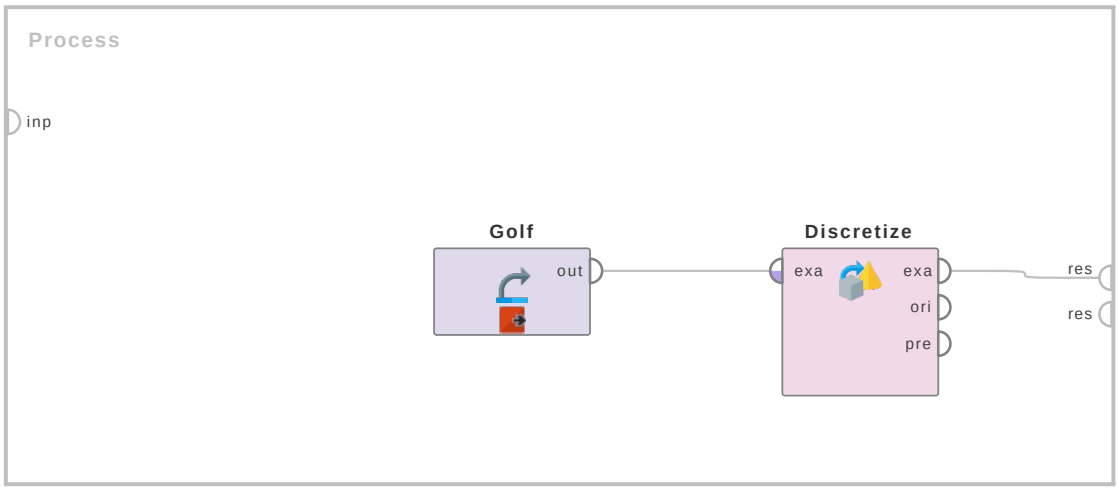
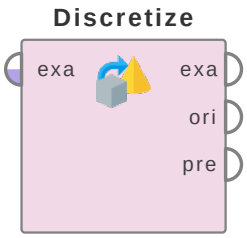



Figure 3.6: Tutorial process 'Discretizing the Temperature attribute of the 'Golf' data set'.

## Discretize by User Specification




This operator discretizes the selected numerical attributes into user-specified classes. The selected numerical attributes will be changed to nominal attributes.

### Description

This operator discretizes the selected numerical attributes to nominal attributes. The numerical values are mapped to the classes according to the thresholds specified by the user in the *classes* parameter. The user can define the classes by specifying the upper limit of each class. The lower limit of every class is automatically defined as the upper limit of the previous class. The lower limit of the first class is assumed to be negative infinity. 'Infinity' can be used to specify positive infinity as upper limit in the classes parameter. This is usually done in the last class. If a class is named as '?', the numerical values falling in this class will be replaced by unknown values in the resulting attributes.

### Differentiation

- **Discretize by Binning** The Discretize By Binning operator creates bins in such a way that the range of all bins is (almost) equal. See page 334 for details.
- **Discretize by Frequency** The Discretize By Frequency operator creates bins in such a way that the number of unique values in all bins are (almost) equal. See page 343 for details.
- **Discretize by Size** The Discretize By Size operator creates bins in such a way that each bin has user-specified size (i.e. number of examples). See page 348 for details.

- **Discretize by Entropy** The discretization is performed by selecting bin boundaries such that the entropy is minimized in the induced partitions. See page 339 for details.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with data. Note that there should be at least one numerical attribute in the input ExampleSet, otherwise use of this operator does not make sense.

## Output Ports

**example set *(exa)*** The selected numerical attributes are converted into nominal attributes according to the user specified classes and the resultant ExampleSet is delivered through this port.

**original *(ori)*** ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model *(pre)*** This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

## Parameters

**create view *(boolean)*** It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some

other parameters (*value type, use value type exception*) become visible in the Parameters panel.

- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type.When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose all examples satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute *(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *parameter* attribute if the meta data is known.

**attributes *(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list. Attributes can be shifted to the right list, which is the list of selected attributes.

**regular expression *(string)*** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression *(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter *(except regular expression*) becomes visible in the Parameters panel.

**except regular expression *(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the *regular expression* parameter).

**value type *(selection)*** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception *(boolean)*** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type *(selection)*** The attributes matching this type will not be selected even if they match the previously mentioned type i.e. *value type* parameter's value.

**block type *(selection)*** The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception *(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be not be selected even if they match the previously mentioned block type i.e. *block type* parameter's value.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '> 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '> 6 && < 11' or '<= 5 || < 0'. But && and || cannot be used together in one numeric condition. Conditions like '(> 0 && < 2) || (>10 && < 12)' are not allowed because they use both && and ||. Use a blank space after '>', '=' and '<' e.g. '<5' will not work, so use '< 5' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles. Special attributes are those attributes which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**classes** This is the most important parameter of this operator. It is used to specify the classes into which the numerical values will be mapped. The names and upper limits of the classes are specified here. The numerical values are mapped to the classes according to the defined thresholds. The user can define the classes by specifying the upper limit of each class. The lower limit of every class is automatically defined as the upper limit of the previous class. The lower limit of the first class is assumed to be negative infinity. 'Infinity' can be used to specify positive infinity as upper limit in the classes parameter. This is usually done in the last class. If a class is named as '?', the numerical values falling in this class will be replaced by unknown values in the resulting attributes.

## Related Documents

- **Discretize by Binning** (page 334)
- **Discretize by Frequency** (page 343)
- **Discretize by Size** (page 348)
- **Discretize by Entropy** (page 339)

## Tutorial Processes

### Discretizing numerical attributes of the Golf data set

The focus of this Example Process is the classes parameter. Almost all parameters other than the classes parameter are for selection of attributes on which discretization is to be performed. For understanding these parameters please study the Example Process of the Select Attributes operator.

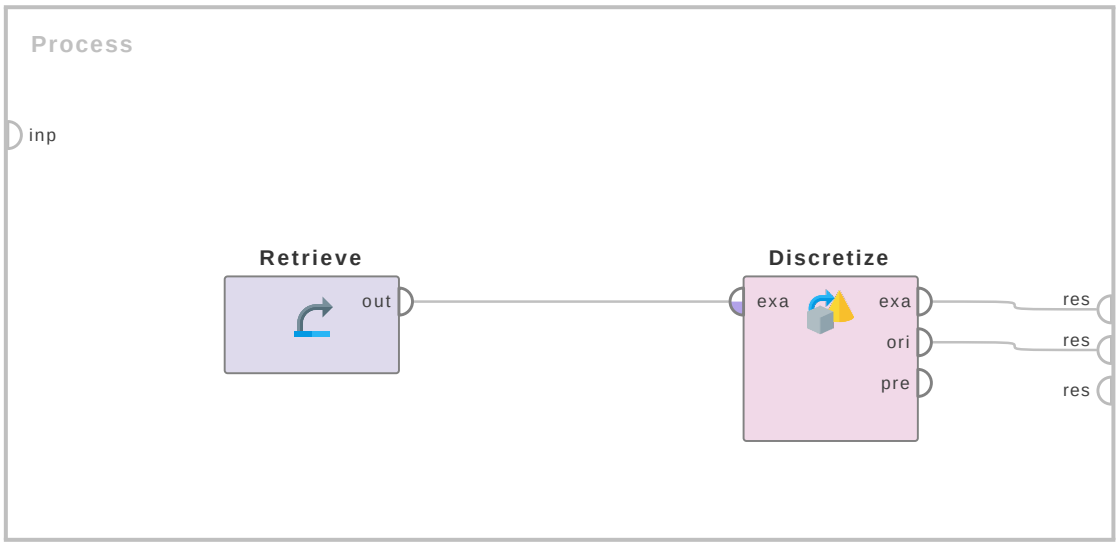
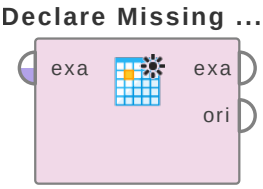



Figure 3.7: Tutorial process 'Discretizing numerical attributes of the Golf data set'.

The 'Golf' data set is loaded using the Retrieve operator. The Discretize by User Specification operator is applied on it. The 'Temperature' and 'Humidity' attributes are selected for discretization. As you can see in the classes parameter, four classes have been specified. The values from negative infinity to 70 will be mapped to 'low' class. The values above 70 to 80 will be mapped to 'average' class. The values above 80 to 90 will be mapped to 'high' class. The values above 90 will be considered as unknown (missing) values. This can be verified by running the process and viewing the results in the Results Workspace. Note that value of the 'Humidity' attribute was 96 and 95 in Row No. 4 and 8 respectively. In the discretized attributes these values are replaced by unknown values because of the last class defined in the classes parameter.

# 3.3 Missing

## Declare Missing Value




This operator declares the specified values of the selected attributes as missing values.

### Description

The Declare Missing Value operator replaces the specified values of the selected attributes by Double.NaN, thus these values will become missing values. These values will be treated as missing values by the subsequent operators. The desired values can be selected through nominal, numeric or regular expression mode. This behavior can be controlled by the *mode* parameter.

### Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

### Output Ports

**example set output *(exa)*** The specified values of the selected attributes are replaced by missing values and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

### Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When it is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression** ***(string)*** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '> 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '> 6 && < 11' or '<= 5 || < 0'. But && and || cannot be used together in one numeric condition. Conditions like '(> 0 && < 2) || (>10 && < 12)' are not allowed because they use both && and ||. Use a blank space after '>', '=' and '<' e.g. '<5' will not work, so use '< 5' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**mode** ***(selection)*** This parameter specifies the type of the value that should be set to missing value. The type can be nominal or numeric or it can be specified through a regular expression.

**numeric value** ***(real)*** This parameter specifies the numerical value that should be declared as missing value.

**nominal value** ***(string)*** This parameter specifies the nominal value that should be declared as missing value.

**expression value** ***(string)*** This parameter specifies the value that should be declared as missing value through an expression.

## Tutorial Processes

### Declaring a nominal value as missing value

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the 'Outlook' attribute has three possible values i.e. 'sunny', 'rain' and 'overcast'. The Declare Missing Value operator is applied on this ExampleSet to change the 'overcast' value of the 'Outlook' attribute to a missing value. The attribute filter type parameter is set to 'single' and the attribute parameter is set to 'Outlook'. The mode parameter is set to 'nominal' and the nominal value parameter is set to 'overcast'. Run the process and compare the resultant ExampleSet with the original ExampleSet. You can clearly see that the value 'overcast' has been replaced by missing values.

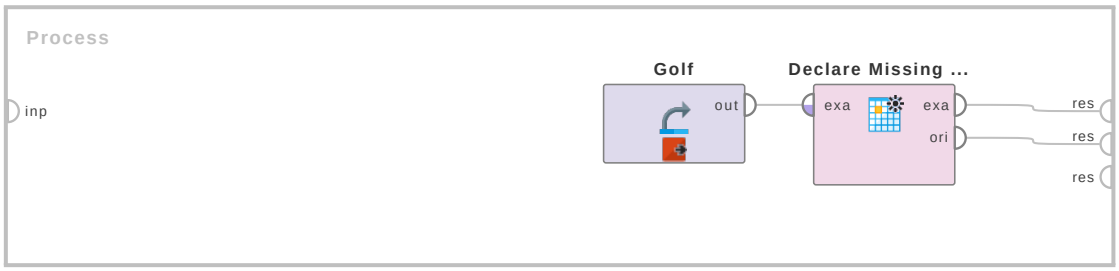
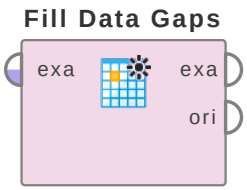



Figure 3.8: Tutorial process 'Declaring a nominal value as missing value'.

# Fill Data Gaps




This operator fills the gaps (based on the ID attribute) in the given ExampleSet by adding new examples in the gaps. The new example will have null values.

## Description

The Fill Data Gaps operator fills the gaps (based on gaps in the ID attribute) in the given ExampleSet by adding new examples in the gaps. The new examples will have null values for all attributes (except the id attribute) which can be replenished by operators like the Replace Missing Values operator. It is ideal that the ID attribute should be of integer type. This operator performs the following steps:

- The data is sorted according to the ID attribute
- All occurring distances between consecutive ID values are calculated.
- The greatest common divisor (GCD) of all distances is calculated.
- All rows which would have an ID value which is a multiple of the GCD but are missing are added to the data set.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.

## Output Ports

**example set output *(exa)*** The gaps in the ExampleSet are filled with new examples and the resulting ExampleSet is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**use gcd for step size *(boolean)*** This parameter indicates if the greatest common divisor (GCD) should be calculated and used as the underlying distance between all data points.

**step size *(integer)*** This parameter is only available when the *use gcd for step size* parameter is set to false. This parameter specifies the step size to be used for filling the gaps.

**start *(integer)*** This parameter can be used for filling the gaps at the beginning (if they occur) before the first data point. For example, if the ID attribute of the given ExampleSet starts with 3 and the *start* parameter is set to 1. Then this operator will fill the gaps in the beginning by adding rows with ids 1 and 2.

**end *(integer)*** This parameter can be used for filling the gaps at the end (if they occur) after the last data point. For example, if the ID attribute of the given ExampleSet ends with 100 and the *end* parameter is set to 105. Then this operator will fill the gaps at the end by adding rows with ids 101 to 105.

## Tutorial Processes

### Introduction to the Fill Data Gaps operator

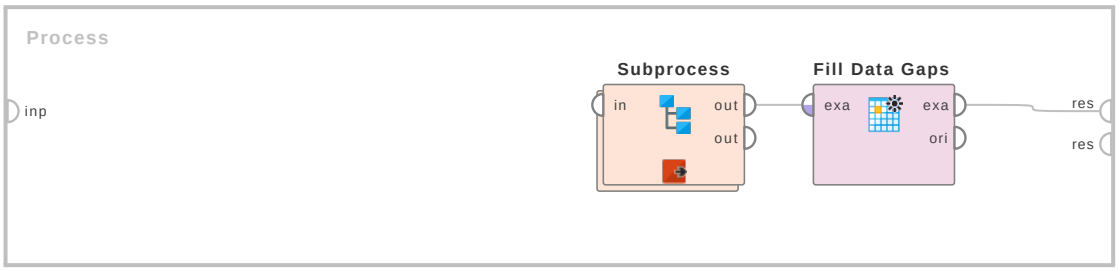
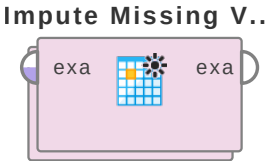



Figure 3.9: Tutorial process 'Introduction to the Fill Data Gaps operator'.

This Example Process starts with the Subprocess operator which delivers an ExampleSet. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 10 examples. Have a look at the id attribute of the ExampleSet. You will see that certain ids are missing: ids 3, 6, 8 and 10. The Fill Data Gaps operator is applied on this ExampleSet to fill these data gaps with examples that have the appropriate ids. You can see the resultant ExampleSet in the Results Workspace. You can see that this ExampleSet has 14 examples. New examples with ids 3, 6, 8 and 10 have been added. But these examples have missing values for all attributes (except the id attribute) which can be replenished by using operators like the Replace Missing Values operator.

# Impute Missing Values




This operator estimates values for the missing values of the selected attributes by applying a model learned for missing values.

## Description

This is a nested operator i.e. it has a subprocess. This subprocess should always accept an ExampleSet and return a model. The Impute Missing Values operator estimates values for missing values by learning models for each attribute (except the label) and applying those models to the ExampleSet. The learner for estimating missing values should be placed in the subprocess of this operator. Please note that depending on the ability of the inner learner to handle missing values this operator might not be able to impute all missing values in some cases. This behavior leads to a warning. It might hence be useful to combine this operator with a subsequent Replace Missing Values operator.

## Input Ports

**example set in *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with data.

## Output Ports

**example set out *(exa)*** The missing values in the ExampleSet are replaced by the values estimated by the given model and the resultant ExampleSet is output of this port.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes in which you want to replace missing values. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in Parameters panel.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to *numeric* type. User should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in Parameters panel.
- **block_type** This option is similar in working to *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to *value_series* block type. When this option is selected some other parameters (block type, use block type exception) become visible in Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are not selected.
- **numeric_value_filter** When this option is selected another parameter (numeric condition) becomes visible in Parameters panel. All numeric attributes whose all examples satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *parameter* attribute if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes.

**regular expression** ***(string)*** Attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. It also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (except regular expression) becomes visible in Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in *regular expression* parameter).

**value type** ***(selection)*** Type of attributes to be selected can be chosen from drop down list.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in Parameters panel.

**except value type** ***(selection)*** Attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type*parameter's value.

**block type** ***(selection)*** Block type of attributes to be selected can be chosen from drop down list.

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in Parameters panel.

**except block type** ***(selection)*** Attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** ***(string)*** Numeric condition for testing examples of numeric attributes is mention here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '>', '=' and '<' e.g. '$<5$' will not work, so use '$< 5$' instead.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is removed prior to selection of this parameter. After selection of this parameter 'att1' will be removed and 'att2' will be selected.

**include special attributes** ***(boolean)*** Special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are delivered to the output port irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**iterate** ***(boolean)*** Set this parameter to true if you want to impute the missing values immediately (after having learned the corresponding concept) and iterate afterwards.

**learn on complete cases** ***(boolean)*** If this parameter is set to true, concepts are learned for estimating missing values only on the basis of complete cases. This option should be used when the inner learning approach cannot handle missing values.

**order** ***(selection)*** This parameter specifies the order of attributes in which missing values should be estimated.

**sort** ***(selection)*** This parameter specifies the sort direction to be used in order strategy.

**use local random seed** ***(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of the *local random seed* will produce the same randomization.

**local random seed** ***(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Using the K-NN scheme for estimating missing values

The 'Labor-Negotiations' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the ExampleSet. You can see that there are numerous missing values

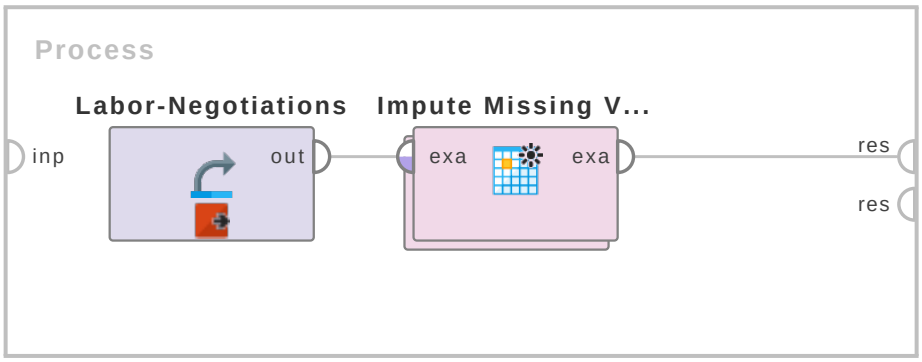
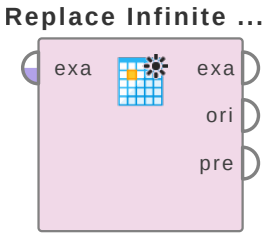



Figure 3.10: Tutorial process 'Using the K-NN scheme for estimating missing values'.

in this ExampleSet. The Impute Missing Values operator is applied on this ExampleSet for estimating missing values. Have a look at the subprocess of this operator. The K-NN operator is applied there for estimating the missing values. The attribute filter type parameter is set to 'all', thus missing values of all attributes will be estimated using the K-NN scheme. All parameters are used with default values. The resultant ExampleSet can be seen in the Results Workspace. You can see that there are no missing values in this ExampleSet because they have been estimated using the K-NN scheme.

# Replace Infinite Values




This operator replaces infinite values of the selected attributes by the specified replacements.

## Description

The Replace Infinite Values operator replaces positive or negative infinite values by the specified replacements. The following replacements are available: none, zero, max_byte, max_int, max_double and missing. The 'max_byte', 'max_int', 'max_double' replacements replace positive infinity by the upper bound and negative infinity by the lower bound of the range of the byte, int and double Java types respectively. If 'missing' replacement is used then the infinite values are replaced by nan (not a number), which is internally used to represent missing values. These missing values can be replenished by the Replace Missing Values operator. Different replacements can be specified for different attributes by using the *columns* parameter. If an attribute's name is not in the list of the *columns* parameter, the replacement specified by the *default* parameter is used.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.

## Output Ports

**example set output *(exa)*** The infinite values are replaced by the specified replacement and the resultant ExampleSet is output of this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model *(pre)*** This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

## Parameters

**create view *(boolean)*** It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.

**attribute filter type** ***(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes in which you want to replace infinite values. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to *numeric* type. User should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in Parameters panel.
- **block_type** This option is similar in working to *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to *value_series* block type. When this option is selected some other parameters (block type, use block type exception) become visible in Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are not selected.
- **numeric_value_filter** When this option is selected another parameter (numeric condition) becomes visible in Parameters panel. All numeric attributes whose all examples satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *parameter* attribute if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes.

**regular expression** ***(string)*** Attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. It also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (except regular expression) becomes visible in Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in *regular expression* parameter).

**value type** ***(selection)*** Type of attributes to be selected can be chosen from drop down list.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in Parameters panel.

**except value type** ***(selection)*** Attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type*parameter's value.

**block type** ***(selection)*** Block type of attributes to be selected can be chosen from drop down list.

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in Parameters panel.

**except block type** ***(selection)*** Attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** ***(string)*** Numeric condition for testing examples of numeric attributes is mention here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '>', '=' and '<' e.g. '$<5$' will not work, so use '$< 5$' instead.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is removed prior to selection of this parameter. After selection of this parameter 'att1' will be removed and 'att2' will be selected.

**include special attributes** ***(boolean)*** Special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are delivered to the output port irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**default** ***(selection)*** This parameter specifies the replacement to apply to all attributes that are not explicitly specified by the *columns* parameter. The following options are available: none, zero, max_byte, max_int, max_double, missing, value.

**columns** ***(list)*** Different attributes can be provided with different types of replacements through this parameter. The default replacement selected by the *default* parameter is applied on attributes that are not explicitly mentioned in the *columns* parameter

**replenish what** ***(selection)*** This parameter specifies if positive or negative infinity values should be replaced.

**replenishment value *(real)*** This parameter is only available when the *default* parameter is set to 'value'. This value will be inserted instead of infinity.

## Tutorial Processes

### Replacing infinite values by missing values

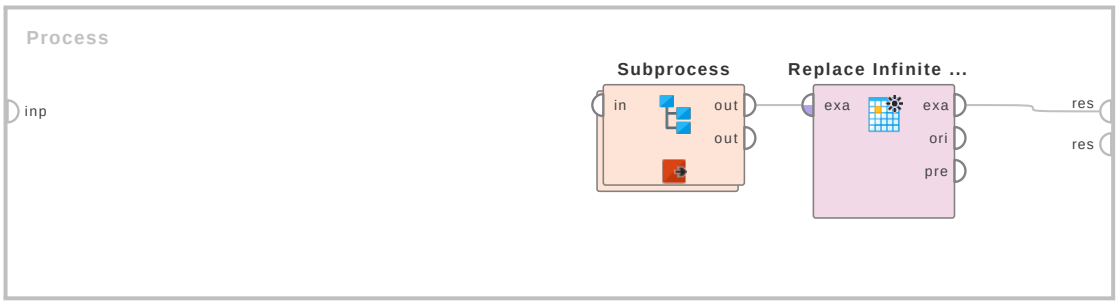
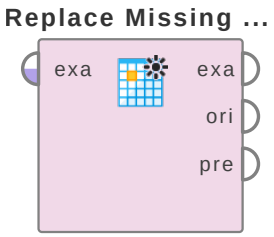



Figure 3.11: Tutorial process 'Replacing infinite values by missing values'.

This Example Process starts with the Subprocess operator which delivers an ExampleSet. A breakpoint is inserted here so that you can have a look at the ExampleSet. Have a look at the Ratio attribute of the ExampleSet. You will see that it has a positive infinity value in the first example. The Replace Infinite Values operator is applied on this ExampleSet to replace infinite values by missing values. The default parameter is set to 'missing' and all other parameters are used with default values. You can see the resultant ExampleSet in the Results Workspace. You can see that the infinite values of the Ratio attribute have been replaced by missing values. These missing values can be replenished by using operators like the Replace Missing Values operator.

# Replace Missing Values




This Operator replaces missing values in Examples of selected Attributes by a specified replacement.

## Description

Missing values can be replaced by the minimum, maximum or average value of that Attribute. Zero can also be used to replace missing values. Any replenishment value can also be specified as a replacement of missing values.

## Differentiation

- **Impute Missing Values**

  This Operator estimates values for the missing values by applying a model learned for missing values.

  See page 362 for details.

- **Replace Infinite Values**

  This Operator replaces infinte values by specified replacements.

  See page 366 for details.

- **Declare Missing Value**

  In contrast to the Replace Missing Values Operators, this Operator set specific values of selected Attributes to missing values.

  See page 357 for details.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet.

## Output Ports

**example set *(exa)*** The ExampleSet with missing values replaced.

**original *(ori)*** The ExampleSet that was given as input is passed through without changes.

**preprocessing model *(pre)*** This port delivers the preprocessing model. It can be used by the Apply Model Operator to perform the specified replacement of missing values on another ExampleSet. This is helpful for example if the Replace Missing Values Operator is used during training and the same replacement has to be applied on test or actual data.

The preprocessing model can also be grouped together with other preprocessing models and learning models by the Group Models Operator.

## Parameters

**create view** Create a View instead of changing the underlying data. If this option is checked, the replacement is delayed until the transformations are needed. This parameter can be considered a legacy option.

**attribute filter type** This parameter allows you to select the Attribute selection filter; the method you want to use for selecting Attributes. It has the following options:

- **all** This option selects all the Attributes of the ExampleSet, no Attributes are removed. This is the default option.
- **single** This option allows the selection of a single Attribute. The required Attribute is selected by the *attribute* parameter.
- **subset** This option allows the selection of multiple Attributes through a list (see parameter *attributes*). If the meta data of the ExampleSet is known all Attributes are present in the list and the required ones can easily be selected.
- **regular_expression** This option allows you to specify a regular expression for the Attribute selection. The regular expression filter is configured by the parameters *regular expression, use except expression and except expression*.
- **value_type** This option allows selection of all the Attributes of a particular type. It should be noted that types are hierarchical. For example real and integer types both belong to the numeric type. The value type filter is configured by the parameters *value type, use value type exception, except value type*.
- **block_type** This option allows the selection of all the Attributes of a particular block type. It should be noted that block types may be hierarchical. For example value_series_start and value_series_end block types both belong to the value_series block type. The block type filter is configured by the parameters *block type, use block type exception, except block type*.
- **no_missing_values** This option selects all Attributes of the ExampleSet which do not contain a missing value in any Example. Attributes that have even a single missing value are removed.
- **numeric_value_filter** All numeric Attributes whose Examples all match a given numeric condition are selected. The condition is specified by the *numeric condition* parameter. Please note that all nominal Attributes are also selected irrespective of the given numerical condition.

**attribute** The required Attribute can be selected from this option. The Attribute name can be selected from the drop down box of the parameter if the meta data is known.

**attributes** The required Attributes can be selected from this option. This opens a new window with two lists. All Attributes are present in the left list. They can be shifted to the right list, which is the list of selected Attributes that will make it to the output port.

**regular expression** Attributes whose names match this expression will be selected. The expression can be specified through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression** If enabled, an exception to the first regular expression can be specified. This exception is specified by the *except regular expression* parameter.

**except regular expression** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in *regular expression* parameter).

**value type** This option allows to select a type of Attribute. One of the following types can be chosen: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date, time.

**use value type exception** If enabled, an exception to the selected type can be specified. This exception is specified by the *except value type* parameter.

**except value type** The Attributes matching this type will be removed from the final output even if they matched the before selected type, specified by the *value type* parameter. One of the following types can be selected here: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date, time.

**block type** This option allows to select a block type of Attribute. One of the following types can be chosen: single_value, value_series, value_series_start, value_series_end, value_matrix, value_matrix_start, value_matrix_end, value_matrix_row_start.

**use block type exception** If enabled, an exception to the selected block type can be specified. This exception is specified by the *except block type* parameter.

**except block type** The Attributes matching this block type will be removed from the final output even if they matched the before selected type by the *block type* parameter. One of the following block types can be selected here: single_value, value_series, value_series_start, value_series_end, value_matrix, value_matrix_start, value_matrix_end, value_matrix_row_start.

**numeric condition** The numeric condition used by the numeric condition filter type. A numeric Attribute is kept if all Examples match the specified condition for this Attribute. For example the numeric condition '$> 6$' will keep all numeric Attributes having a value of greater than 6 in every Example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Nominal Attributes are always kept, regardless of the specified numeric condition.

**include special attributes** Special Attributes are Attributes with special roles. These are: id, label, prediction, cluster, weight and batch. Also custom roles can be assigned to Attributes. By default all special Attributes are delivered to the output port irrespective of the conditions in the Select Attribute Operator. If this parameter is set to true, special Attributes are also tested against conditions specified in the Select Attribute Operator and only those Attributes are selected that match the conditions.

**invert selection** If this parameter is set to true the selection is reversed. In that case all Attributes matching the specified condition are removed and the other Attributes remain in the output ExampleSet. Special Attributes are kept independent of the *invert selection* parameter as along as the *include special attributes* parameter is not set to true. If so the condition is also applied to the special Attributes and the selection is reversed if this parameter is checked.

**default** This parameter specifies how missing values are replaced by default. This default option is used for all Attributes which are not specified by the *columns* parameter.

- **none** Missing values are not replaced by default.

- **minimum** Missing values are replaced by the minimum value of that Attribute.
- **maximum** Missing values are replaced by the maximum value of that Attribute.
- **average** Missing values are replaced by the average value of that Attribute.
- **zero** Missing values are replaced by zero.
- **value** Missing values are replaced by the value specified in the *replenishment value* parameter.

**columns** Different Attributes can be provided with a different type of replacements through this parameter. The default function selected by the *default* parameter is applied on Attributes that are not explicitly mentioned in the *columns* parameter.

**replenishment value** If the *default* parameter is set to value, this parameter specifies the value which is used to replace missing values.

## Tutorial Processes

### Replacing missing values of the Labor Negotiations data set

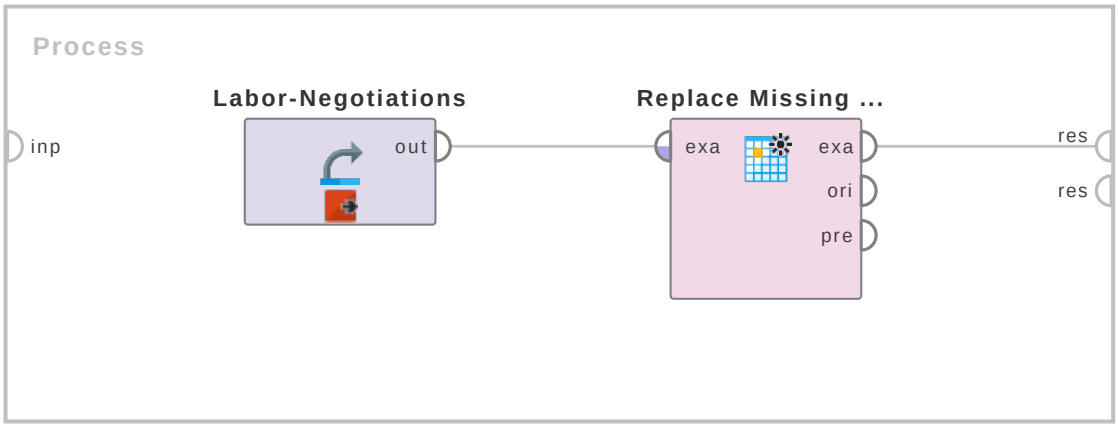



Figure 3.12: Tutorial process 'Replacing missing values of the Labor Negotiations data set'.

This Process shows the usage of the Replace Missing Values Operator on the Labor-Negotiations data set from the Samples folder.

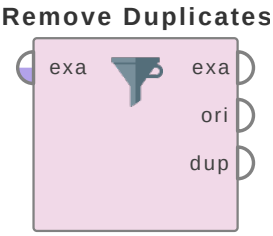
The Operator is configured that it applies the replacement on all Attributes which have at least one missing value (attribute filter type is no_missing_values and invert selection is true). In the columns parameter several Attributes are set to different replacement methods:

wage-inc-1st: minimum
wage-inc-2nd: maximum
wage-inc-3rd: zero
working-hours: value

The parameter replenishment value is set to 35, so that all missing values of the Attribute working-hours are replaced by 35. The missing values of the remaining Attributes are replaced by the average of the Attribute (parameter default).

# 3.4 Duplicates

## Remove Duplicates




This operator removes duplicate examples from an ExampleSet by comparing all examples with each other on the basis of the specified attributes. Two examples are considered duplicate if the selected attributes have the same values in them.

### Description

The Remove Duplicates operator removes duplicate examples from an ExampleSet by comparing all examples with each other on the basis of the specified attributes. This operator removes duplicate examples such that only one of all the duplicate examples is kept. Two examples are considered duplicate if the selected attributes have the same values in them. Attributes can be selected from the *attribute filter type* parameter and other associated parameters. Suppose two attributes 'att1' and 'att2' are selected and 'att1' and 'att2' have three and two possible values respectively. Thus there are total 6 (i.e. 3 x 2) unique combinations of these two attribute. Thus the resultant ExampleSet can have 6 examples at most. This operator works on all attribute types.

### Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

### Output Ports

**example set output *(exa)*** The duplicate examples are removed from the given ExampleSet and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**duplicates *(dup)*** The duplicated examples from the given ExampleSet are delivered through this port.

### Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.

- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to *numeric* type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression** ***(string)*** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type *(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception *(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type *(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type *(selection)*** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception *(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type *(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition *(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Use a blank space after '>', '=' and '<' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes *(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection *(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**treat missing values as duplicates *(boolean)*** This parameter specifies if missing values should be treated as duplicates or not. If set to true, missing values are considered as duplicate values.

## Tutorial Processes

### Removing duplicate values from the Golf data set on the basis of the Outlook and Wind attributes

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the Outlook attribute has three possible values i.e. 'sunny', 'rain' and 'overcast'. The Wind attribute has two possible values i.e. 'true' and 'false'. The Remove Duplicates operator is applied on this ExampleSet to remove duplicate

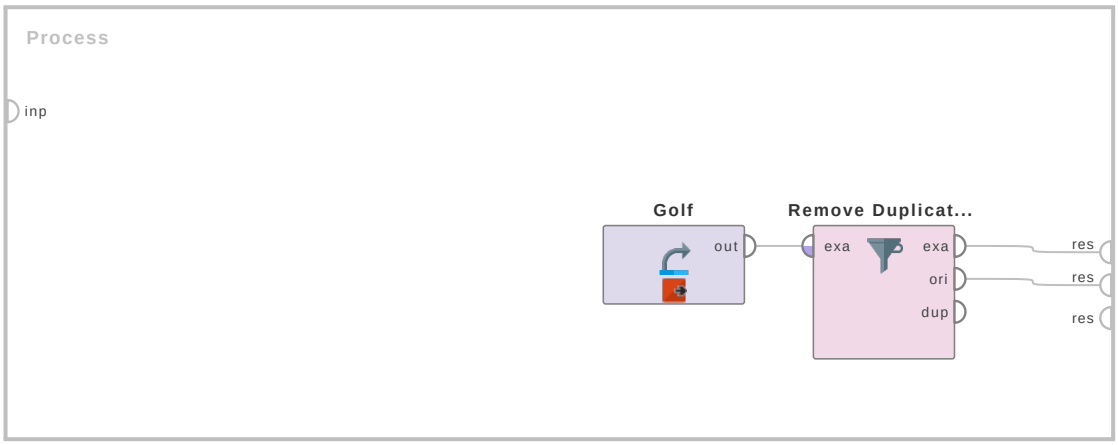
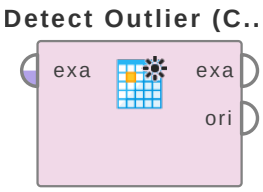



Figure 3.13: Tutorial process 'Removing duplicate values from the Golf data set on the basis of the Outlook and Wind attributes'.

examples on the basis of the Outlook and Wind attributes. The attribute filter type parameter is set to 'value type' and the value type parameter is set to 'nominal', thus two examples that have same values in their Outlook and Wind attributes are considered as duplicate. Note that the Play attribute is not selected although its value type is nominal because it is a special attribute (because it has label role). To select attributes with special roles the include special attributes parameter should be set to true. The Outlook and Wind attributes have 3 and 2 possible values respectively. Thus the resultant ExampleSet will have 6 examples at most i.e. one example for each possible combination of attribute values. You can see the resultant ExampleSet in the Results Workspace. You can see that it has 6 examples and all examples have a different combination of the Outlook and Wind attribute values.

# 3.5 Outliers

## Detect Outlier (COF)




This operator identifies outliers in the given ExampleSet based on the Class Outlier Factors (COF).

### Description

The main concept of an ECODB (Enhanced Class Outlier - Distance Based) algorithm is to rank each instance in the ExampleSet given the parameters *N* (top *N* class outliers), and *K* (the number of nearest neighbors). The rank of each instance is found using the formula:
*COF = PCL(T,K) - norm(deviation(T)) + norm(kDist(T))*

- *PCL(T,K)* is the Probability of the Class Label of the instance *T* with respect to the class labels of its *K* nearest neighbors.
- *norm(Deviation(T))* and *norm(KDist(T))* are the normalized values of *Deviation(T)* and *KDist(T)* respectively and their values fall in the range [0 - 1].
- *Deviation(T)* is how much the instance *T* deviates from instances of the same class. It is computed by summing the distances between the instance *T* and every instance belonging to the same class.
- *KDist(T)* is the summation of the distance between the instance *T* and its *K* nearest neighbors.

This operator adds a new boolean attribute named 'outlier' to the given ExampleSet. If the value of this attribute is true, that example is an outlier and vice versa. Another special attribute 'COF Factor' is also added to the ExampleSet. This attribute measures the degree of being Class Outlier for an example.

An outlier is an example that is numerically distant from the rest of the examples of the ExampleSet. An outlying example is one that appears to deviate markedly from other examples of the ExampleSet. Outliers are often (not always) indicative of measurement error. In this case such examples should be discarded.

### Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Generate Data operator in the attached Example Process. The output of other operators can also be used as input.

### Output Ports

**example set output *(exa)*** A new boolean attribute 'outlier' and a real attribute 'COF Factor' is added to the given ExampleSet and the ExampleSet is delivered through this output port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**number of neighbors *(integer)*** This parameter specifies the *k* value for the *k* nearest neighbors to be the analyzed. The minimum and maximum values for this parameter are 1 and 1 million respectively.

**number of class outliers *(integer)*** This parameter specifies the number of top-n Class Outliers to be looked for. The resultant ExampleSet will have *n* number of examples that are considered outliers. The minimum and maximum values for this parameter are 2 and 1 million respectively.

**measure types *(selection)*** This parameter is used for selecting the type of measure to be used for measuring the distance between points.The following options are available: *mixed measures*, *nominal measures*, *numerical measures* and *Bregman divergences*.

**mixed measure *(selection)*** This parameter is available when the *measure type* parameter is set to 'mixed measures'. The only available option is the 'Mixed Euclidean Distance'

**nominal measure *(selection)*** This parameter is available when the *measure type* parameter is set to 'nominal measures'. This option cannot be applied if the input ExampleSet has numerical attributes. In this case the 'numerical measure' option should be selected.

**numerical measure *(selection)*** This parameter is available when the *measure type* parameter is set to 'numerical measures'. This option cannot be applied if the input ExampleSet has nominal attributes. If the input ExampleSet has nominal attributes the 'nominal measure' option should be selected.

**divergence *(selection)*** This parameter is available when the *measure type* parameter is set to 'Bregman divergences'.

**kernel type *(selection)*** This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance'. The type of the kernel function is selected through this parameter. Following kernel types are supported:

- **dot** The dot kernel is defined by*k(x,y)=x*y* i.e.it is inner product of*x* and *y*.
- **radial** The radial kernel is defined by *exp(-g ||x-y||^2)* where *g* is the *gamma* that is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by *k(x,y)=(x*y+1)^d* where *d* is the degree of the polynomial and it is specified by the *kernel degree* parameter. The Polynomial kernels are well suited for problems where all the training data is normalized.
- **neural** The neural kernel is defined by a two layered neural net *tanh(a x*y+b)* where *a* is *alpha* and *b* is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is 1/N, where N is the data dimension. Note that not all choices of *a* and *b* lead to a valid kernel function.
- **sigmoid** This is the sigmoid kernel. Please note that the *sigmoid* kernel is not valid under some parameters.

- **anova** This is the anova kernel. It has adjustable parameters *gamma* and *degree*.
- **epachnenikov** The Epanechnikov kernel is this function *(3/4)(1-u2)* for *u* between -1 and 1 and zero for *u* outside that range. It has two adjustable parameters *kernel sigma1* and *kernel degree*.
- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters *kernel sigma1, kernel sigma2* and *kernel sigma3*.
- **multiquadric** The multiquadric kernel is defined by the square root of *||x-y||^2 + c^2*. It has adjustable parameters *kernel sigma1* and *kernel sigma shift*.

**kernel gamma *(real)*** This is the SVM kernel parameter gamma. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1 *(real)*** This is the SVM kernel parameter sigma1. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2 *(real)*** This is the SVM kernel parameter sigma2. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3 *(real)*** This is the SVM kernel parameter sigma3. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel shift *(real)*** This is the SVM kernel parameter shift. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *multiquadric*.

**kernel degree *(real)*** This is the SVM kernel parameter degree. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a *(real)*** This is the SVM kernel parameter a. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

**kernel b *(real)*** This is the SVM kernel parameter b. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

## Tutorial Processes

### Detecting outliers from an ExampleSet

The Generate Data operator is used for generating an ExampleSet. The target function parameter is set to 'gaussian mixture clusters'. The number examples and number of attributes parameters are set to 200 and 2 respectively. A breakpoint is inserted here so that you can view the ExampleSet in the Results Workspace. A good plot of the ExampleSet can be seen by switching to the 'Plot View' tab. Set Plotter to 'Scatter', x-Axis to 'att1' and y-Axis to 'att2' to view the scatter plot of the ExampleSet.

The Detect Outlier (COF) operator is applied on the ExampleSet. The number of neighbors and number of class outliers parameters are set to 7. The resultant ExampleSet can be viewed

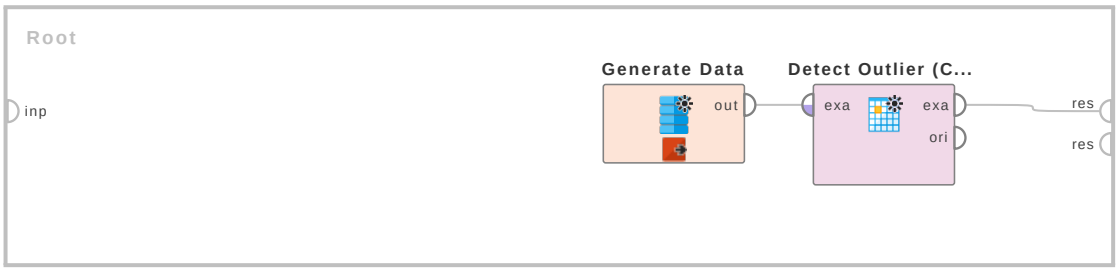
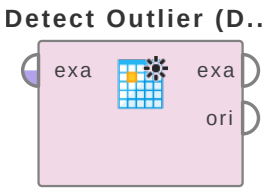



Figure 3.14: Tutorial process ‘Detecting outliers from an ExampleSet’.

in the Results Workspace. For better understanding, switch to the ‘Plot View’ tab. Set Plotter to ‘Scatter’, x-Axis to ‘att1’, y-Axis to ‘att2’ and Color Column to ‘outlier’ to view the scatter plot of the ExampleSet (the outliers are marked red).

# Detect Outlier (Densities)




This operator identifies outliers in the given ExampleSet based on the data density. All objects that have at least *p* proportion of all objects farther away than distance *D* are considered outliers.

## Description

The Detect Outlier (Densities) operator is an outlier detection algorithm that calculates the *DB(p,D)-outliers* for the given ExampleSet. A *DB(p,D)-outlier* is an object which is at least *D* distance away from at least *p* proportion of all objects. The two real-valued parameters *p* and *D* can be specified through the *proportion* and *distance* parameters respectively. The *DB(p,D)-outliers* are distance-based outliers according to Knorr and Ng. This operator implements a global homogenous outlier search.

This operator adds a new boolean attribute named 'outlier' to the given ExampleSet. If the value of this attribute is true, that example is an outlier and vice versa. Different distance functions are supported by this operator. The desired distance function can be selected by the *distance function* parameter.

An outlier is an example that is numerically distant from the rest of the examples of the ExampleSet. An outlying example is one that appears to deviate markedly from other examples of the ExampleSet. Outliers are often (not always) indicative of measurement error. In this case such examples should be discarded.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Generate Data operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set output *(exa)*** A new boolean attribute 'outlier' is added to the given ExampleSet and the ExampleSet is delivered through this output port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**distance *(real)*** This parameter specifies the distance *D* parameter for calculation of the *DB(p,D)-outliers*.

**proportion *(real)*** This parameter specifies the proportion *p* parameter for calculation of the *DB(p,D)-outliers*.

**distance function *(selection)*** This parameter specifies the distance function that will be used for calculating the distance between two examples.

## Tutorial Processes

### Detecting outliers from an ExampleSet

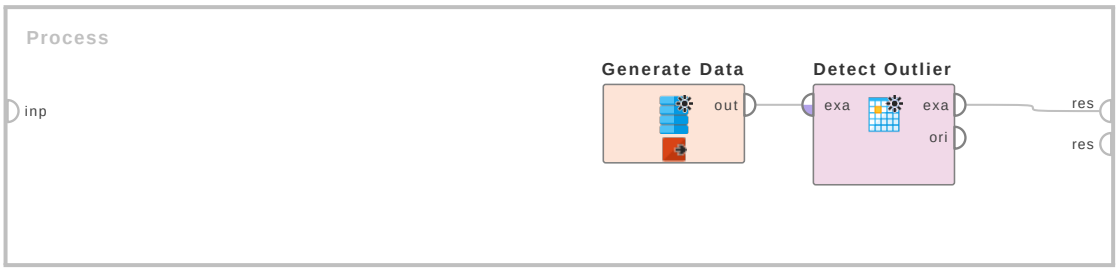
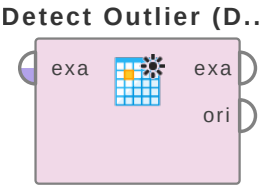



Figure 3.15: Tutorial process ‘Detecting outliers from an ExampleSet’.

The Generate Data operator is used for generating an ExampleSet. The target function parameter is set to ‘gaussian mixture clusters’. The number examples and number of attributes parameters are set to 200 and 2 respectively. A breakpoint is inserted here so that you can view the ExampleSet in the Results Workspace. A good plot of the ExampleSet can be seen by switching to the ‘Plot View’ tab. Set Plotter to ‘Scatter’, x-Axis to ‘att1’ and y-Axis to ‘att2’ to view the scatter plot of the ExampleSet.

The Detect Outlier (Densities) operator is applied on the ExampleSet. The distance and proportion parameters are set to 4.0 and 0.8 respectively. The resultant ExampleSet can be viewed in the Results Workspace. For better understanding switch to the ‘Plot View’ tab. Set Plotter to ‘Scatter’, x-Axis to ‘att1’, y-Axis to ‘att2’ and Color Column to ‘outlier’ to view the scatter plot of the ExampleSet (the outliers are marked red). The number of outliers may differ depending on the randomization, if the random seed parameter of the process is set to 1997, you will see 5 outliers.

# Detect Outlier (Distances)




This operator identifies *n* outliers in the given ExampleSet based on the distance to their *k* nearest neighbors. The variables *n* and *k* can be specified through parameters.

## Description

This operator performs outlier search according to the outlier detection approach recommended by Ramaswamy, Rastogi and Shim in "Efficient Algorithms for Mining Outliers from Large Data Sets". In their paper, a formulation for distance-based outliers is proposed that is based on the distance of a point from its *k-th* nearest neighbor. Each point is ranked on the basis of its distance to its *k-th* nearest neighbor and the top *n* points in this ranking are declared to be outliers. The values of *k* and *n* can be specified by the *number of neighbors* and *number of outliers* parameters respectively. This search is based on simple and intuitive distance-based definitions for outliers by Knorr and Ng which in simple words is: 'A point *p* in a data set is an outlier with respect two parameters *k* and *d* if no more than *k* points in the data set are at a distance of *d* or less from *p*'.

This operator adds a new boolean attribute named 'outlier' to the given ExampleSet. If the value of this attribute is true that example is an outlier and vice versa. *n* examples will have the value true in the 'outlier' attribute (where *n* is the value specified in the *number of outliers* parameter). Different distance functions are supported by this operator. The desired distance function can be selected by the *distance function* parameter.

An outlier is an example that is numerically distant from the rest of the examples of the ExampleSet. An outlying example is one that appears to deviate markedly from other examples of the ExampleSet. Outliers are often (not always) indicative of measurement error. In this case such examples should be discarded.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Generate Data operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set output *(exa)*** A new boolean attribute 'outlier' is added to the given ExampleSet and the ExampleSet is delivered through this output port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**number of neighbors *(integer)*** This parameter specifies the *k* value for the *k-th* nearest neighbors to be the analyzed. The minimum and maximum values for this parameter are 1 and 1 million respectively.

**number of outliers *(integer)*** This parameter specifies the number of top-n outliers to be looked for. The resultant ExampleSet will have *n* number of examples that are considered outliers. The minimum and maximum values for this parameter are 2 and 1 million respectively.

**distance function *(selection)*** This parameter specifies the distance function that will be used for calculating the distance between two examples.

## Tutorial Processes

### Detecting outliers from an ExampleSet

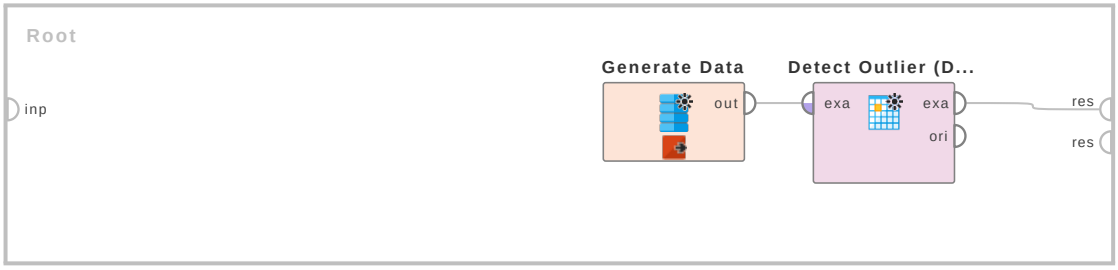
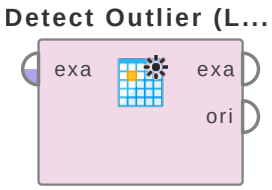



Figure 3.16: Tutorial process 'Detecting outliers from an ExampleSet'.

The Generate Data operator is used for generating an ExampleSet. The target function parameter is set to 'gaussian mixture clusters'. The number examples and number of attributes parameters are set to 200 and 2 respectively. A breakpoint is inserted here so that you can view the ExampleSet in the Results Workspace. A good plot of the ExampleSet can be seen by switching to the 'Plot View' tab. Set Plotter to 'Scatter', x-Axis to 'att1' and y-Axis to 'att2' to view the scatter plot of the ExampleSet.

The Detect Outlier (Distances) operator is applied on this ExampleSet. The number of neighbors and number of outliers parameters are set to 4 and 12 respectively. Thus 12 examples of the resultant ExampleSet will have true value in the 'outlier' attribute. This can be verified by viewing the ExampleSet in the Results Workspace. For better understanding switch to the 'Plot View' tab. Set Plotter to 'Scatter', x-Axis to 'att1', y-Axis to 'att2' and Color Column to 'outlier' to view the scatter plot of the ExampleSet (the outliers are marked red).

# Detect Outlier (LOF)




This operator identifies outliers in the given ExampleSet based on local outlier factors (LOF). The LOF is based on a concept of a local density, where locality is given by the *k* nearest neighbors, whose distance is used to estimate the density. By comparing the local density of an object to the local densities of its neighbors, one can identify regions of similar density, and points that have a substantially lower density than their neighbors. These are considered to be outliers

## Description

This operator performs a LOF outlier search. LOF outliers or outliers with a local outlier factor per object are density based outliers according to Breunig, Kriegel, et al. As indicated by the name, the local outlier factor is based on a concept of a local density, where locality is given by *k* nearest neighbors, whose distance is used to estimate the density. By comparing the local density of an object to the local densities of its neighbors, one can identify regions of similar density, and points that have a substantially lower density than their neighbors. These are considered to be outliers. The local density is estimated by the typical distance at which a point can be 'reached' from its neighbors. The definition of 'reachability distance' used in LOF is an additional measure to produce more stable results within clusters.

The approach to find the outliers is based on measuring the density of objects and its relation to each other (referred to as local reachability density). Based on the average ratio of the local reachability density of an object and its k-nearest neighbors (i.e. the objects in its k-distance neighborhood), a local outlier factor (LOF) is computed. The approach takes a parameter *MinPts* (actually specifying the 'k') and it uses the maximum LOFs for objects in a *MinPts* range (lower bound and upper bound to *MinPts*).

This operator supports cosine, inverted cosine, angle and squared distance in addition to the usual euclidian distance which can be specified by the *distance function* parameter. In the first step, the objects are grouped into containers. For each object, using a radius screening of all other objects, all the available distances between that object and another object (or group of objects) on the same radius given by the distance are associated with a container. That container then has the distance information as well as the list of objects within that distance (usually only a few) and the information about how many objects are in the container.

In the second step, three things are done:

1. The containers for each object are counted in ascending order according to the cardinality of the object list within the container (= that distance) to find the k-distances for each object and the objects in that k-distance (all objects in all the subsequent containers with a smaller distance).

2. Using this information, the local reachability densities are computed by using the maximum of the actual distance and the k-distance for each object pair (object and objects in k-distance) and averaging it by the cardinality of the k-neighborhood and then taking the reciprocal value.

3. The LOF is computed for each *MinPts* value in the range (actually for all up to upper bound) by averaging the ratio between the MinPts-local reachability-density of all objects in the k-neighborhood and the object itself. The maximum LOF in the *MinPts* range is passed as final LOF to each object.

Afterwards LOFs are added as values for a special real-valued outlier attribute in the ExampleSet which the operator will return.

An outlier is an example that is numerically distant from the rest of the examples of the ExampleSet. An outlying example is one that appears to deviate markedly from other examples of the ExampleSet. Outliers are often (not always) indicative of measurement error. In this case such examples should be discarded.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Generate Data operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set output *(exa)*** A new attribute 'outlier' is added to the given ExampleSet which is then delivered through this output port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**minimal points lower bound *(integer)*** This parameter specifies the lower bound for *MinPts* for the Outlier test.

**minimal points upper bound *(integer)*** This parameter specifies the upper bound for *MinPts* for the Outlier test.

**distance function *(selection)*** This parameter specifies the distance function that will be used for calculating the distance between two objects.

## Tutorial Processes

### Detecting outliers from an ExampleSet

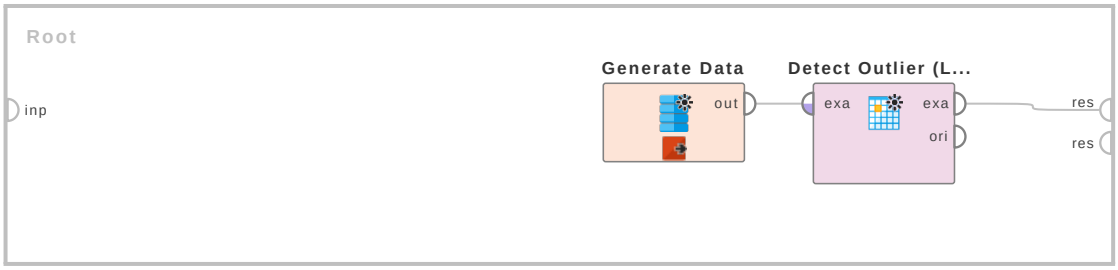



Figure 3.17: Tutorial process 'Detecting outliers from an ExampleSet'.

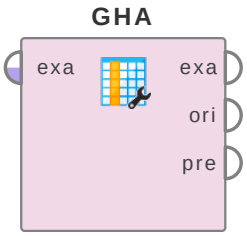
The Generate Data operator is used for generating an ExampleSet. The target function parameter is set to 'gaussian mixture clusters'. The number examples and number of attributes

parameters are set to 200 and 2 respectively. A breakpoint is inserted here so that you can view the ExampleSet in the Results Workspace. A good plot of the ExampleSet can be seen by switching to the 'Plot View' tab. Set Plotter to 'Scatter', x-Axis to 'att1' and y-Axis to 'att2' to view the scatter plot of the ExampleSet.

The Detect Outlier (LOF) operator is applied on this ExampleSet with default values for all parameters. The minimal points lower bound and minimal points upper bound parameters are set to 10 and 20 respectively. The resultant ExampleSet can be seen in the Results Workspace. For better understanding switch to the 'Plot View' tab. Set Plotter to 'Scatter', x-Axis to 'att1', y-Axis to 'att2' and Color Column to 'outlier' to view the scatter plot of the ExampleSet.

# 3.6 Dimensionality Reduction

## Generalized Hebbian Algorithm




This operator is an implementation of the Generalized Hebbian Algorithm (GHA) which is an iterative method for computing principal components. The user can specify manually the required number of principal components.

### Description

The Generalized Hebbian Algorithm (GHA) is a linear feedforward neural network model for unsupervised learning with applications primarily in principal components analysis. From a computational point of view, it can be advantageous to solve the eigenvalue problem by iterative methods which do not need to compute the covariance matrix directly. This is useful when the ExampleSet contains many attributes (hundreds or even thousands).

Principal Component Analysis (PCA) is an attribute reduction procedure. It is useful when you have obtained data on a number of attributes (possibly a large number of attributes), and believe that there is some redundancy in those attributes. In this case, redundancy means that some of the attributes are correlated with one another, possibly because they are measuring the same construct. Because of this redundancy, you believe that it should be possible to reduce the observed attributes into a smaller number of principal components (artificial attributes) that will account for most of the variance in the observed attributes. Principal Component Analysis is a mathematical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated attributes into a set of values of uncorrelated attributes called principal components. The number of principal components is less than or equal to the number of original attributes. This transformation is defined in such a way that the first principal component's variance is as high as possible (accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it should be orthogonal to (uncorrelated with) the preceding components.

### Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along with the data. Please note that this operator cannot handle nominal attributes; it works on numerical attributes.

### Output Ports

**example set *(exa)*** The Generalized Hebbian Algorithm is performed on the input ExampleSet and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model *(pre)*** This port delivers the GHA model.

## Parameters

**number of components *(integer)*** The number of components to keep is specified by the *number of components* parameter. If set to -1 the number of principal components in the resultant ExampleSet is equal to the number of attributes in the original ExampleSet.

**number of iterations *(integer)*** This parameter specifies the number of iterations to apply the update rule.

**learning rate *(real)*** This parameter specifies the learning rate of the GHA.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. It is available only if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Dimensionality reduction of the Polynomial data set using the GHA operator

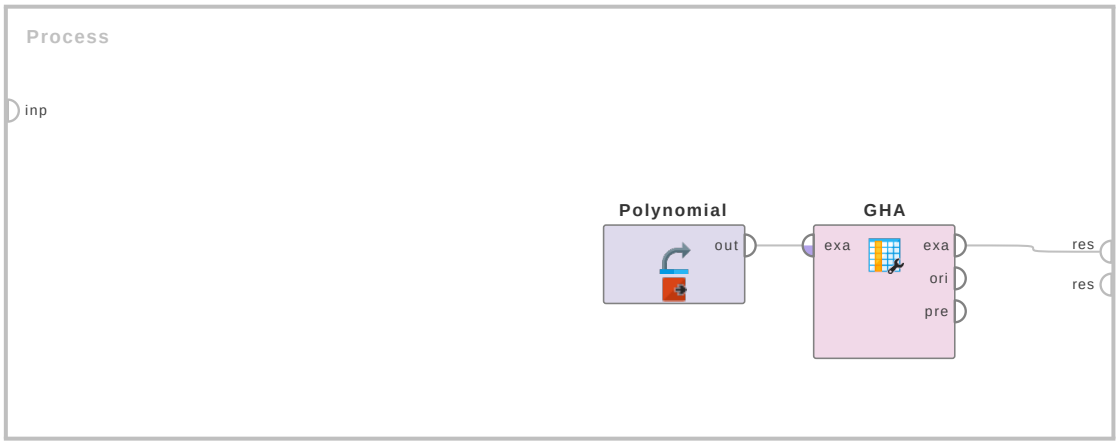
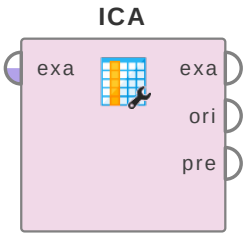



Figure 3.18: Tutorial process 'Dimensionality reduction of the Polynomial data set using the GHA operator'.

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes. The Generalized Hebbian Algorithm operator is applied on the 'Polynomial' data set. The number of components parameter is set to 3. Thus the resultant ExampleSet will be composed of 3 principal components. All other parameters are used with default values. Run the process, you will see that the ExampleSet that had 5 attributes has been reduced to an ExampleSet with 3 principal components.

# Independent Component Analysis




This operator performs the Independent Component Analysis (ICA) of the given ExampleSet using the FastICA-algorithm of Hyvärinen and Oja.

## Description

Independent component analysis (ICA) is a very general-purpose statistical technique in which observed random data are linearly transformed into components that are maximally independent from each other, and simultaneously have "interesting" distributions. Such a representation seems to capture the essential structure of the data in many applications, including feature extraction. ICA is used for revealing hidden factors that underlie sets of random variables or measurements. ICA is superficially related to principal component analysis (PCA) and factor analysis. ICA is a much more powerful technique, however, capable of finding the underlying factors or sources when these classic methods fail completely. This operator implements the FastICA-algorithm of A. Hyvärinen and E. Oja. The FastICA-algorithm has most of the advantages of neural algorithms: It is parallel, distributed, computationally simple, and requires little memory space.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along with the data. Please note that this operator cannot handle nominal attributes; it works on numerical attributes.

## Output Ports

**example set output *(exa)*** The Independent Component Analysis is performed on the input ExampleSet and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model *(pre)*** This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

## Parameters

**dimensionality reduction *(selection)*** This parameter indicates which type of dimensionality reduction (reduction in number of attributes) should be applied.

- **none** if this option is selected, dimensionality reduction is not performed.

- **fixed_number** if this option is selected, only a fixed number of components are kept. The number of components to keep is specified by the *number of components* parameter.

**number of components *(integer)*** This parameter is only available when the *dimensionality reduction* parameter is set to 'fixed number'. The number of components to keep is specified by the *number of components* parameter.

**algorithm type *(selection)*** This parameter specifies the type of algorithm to be used.

- **parallel** If parallel option is selected, the components are extracted simultaneously.
- **deflation** If deflation option is selected, the components are extracted one at a time.

**function *(selection)*** This parameter specifies the functional form of the G function to be used in the approximation to neg-entropy.

**alpha *(real)*** This parameter specifies the alpha constant in range [1, 2] which is used in approximation to neg-entropy.

**row norm *(boolean)*** This parameter indicates whether rows of the data matrix should be standardized beforehand.

**max iteration *(integer)*** This parameter specifies the maximum number of iterations to perform.

**tolerance *(real)*** This parameter specifies a positive scalar giving the tolerance at which the un-mixing matrix is considered to have converged.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Dimensionality reduction of the Sonar data set using the Independent Component Analysis operator

The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 60 attributes. The Independent Component Analysis operator is applied on the 'Sonar' data set. The dimensionality reduction parameter is set to 'fixed number' and the number_of_components parameter is set to 10. Thus the resultant ExampleSet will be composed of 10 components (artificial attributes). You can see the resultant ExampleSet in the Results Workspace and verify that it has only 10 attributes. Please note that these attributes are not original attributes of the 'Sonar' data set. These attributes were created using the ICA procedure.

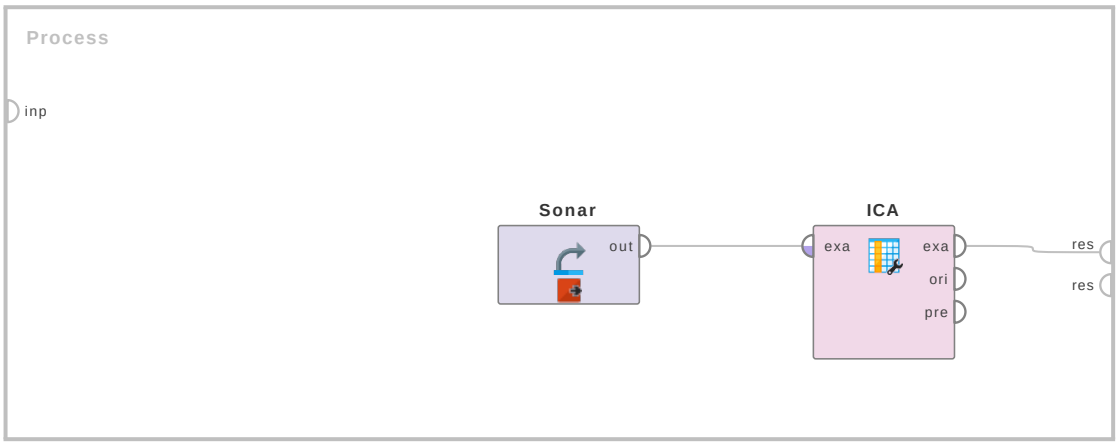
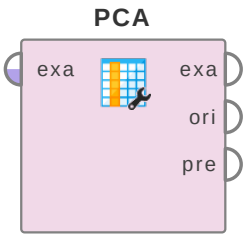



Figure 3.19: Tutorial process 'Dimensionality reduction of the Sonar data set using the Independent Component Analysis operator'.

## Principal Component Analysis




This operator performs a Principal Component Analysis (PCA) using the covariance matrix. The user can specify the amount of variance to cover in the original data while retaining the best number of principal components. The user can also specify manually the number of principal components.

### Description

Principal component analysis (PCA) is an attribute reduction procedure. It is useful when you have obtained data on a number of attributes (possibly a large number of attributes), and believe that there is some redundancy in those attributes. In this case, redundancy means that some of the attributes are correlated with one another, possibly because they are measuring the same construct. Because of this redundancy, you believe that it should be possible to reduce the observed attributes into a smaller number of principal components (artificial attributes) that will account for most of the variance in the observed attributes.

Principal Component Analysis is a mathematical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated attributes into a set of values of uncorrelated attributes called principal components. The number of principal components is less than or equal to the number of original attributes. This transformation is defined in such a way that the first principal component's variance is as high as possible (accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it should be orthogonal to (uncorrelated with) the preceding components.

Please note that PCA is sensitive to the relative scaling of the original attributes. This means that whenever different attributes have different units (like temperature and mass); PCA is a somewhat arbitrary method of analysis. Different results would be obtained if one used Fahren-

heit rather than Celsius for example.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along with the data. Please note that this operator cannot handle nominal attributes; it works on numerical attributes.

## Output Ports

**example set *(exa)*** The Principal Component Analysis is performed on the input ExampleSet and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model *(pre)*** This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

## Parameters

**dimensionality reduction *(selection)*** This parameter indicates which type of dimensionality reduction (reduction in number of attributes) should be applied.

- **none** if this option is selected, no component is removed from the ExampleSet.
- **keep_variance** if this option is selected, all the components with a cumulative variance greater than the given threshold are removed from the ExampleSet. The threshold is specified by the *variance threshold* parameter.
- **fixed_number** if this option is selected, only a fixed number of components are kept. The number of components to keep is specified by the *number of components* parameter.

**variance threshold *(real)*** This parameter is available only when the *dimensionality reduction* parameter is set to 'keep variance'. All the components with a cumulative variance greater than the *variance threshold* are removed from the ExampleSet.

**number of components *(integer)*** This parameter is only available when the *dimensionality reduction* parameter is set to 'fixed number'. The number of components to keep is specified by the *number of components* parameter.

## Tutorial Processes

### Dimensionality reduction of the Polynomial data set using the Principal Component Analysis operator

The 'Polynomial' data set is loaded using the Retrieve operator. The Covariance Matrix operator is applied on it. A breakpoint is inserted here so that you can have a look at the ExampleSet and its covariance matrix. For this purpose the Covariance Matrix operator is applied otherwise it is

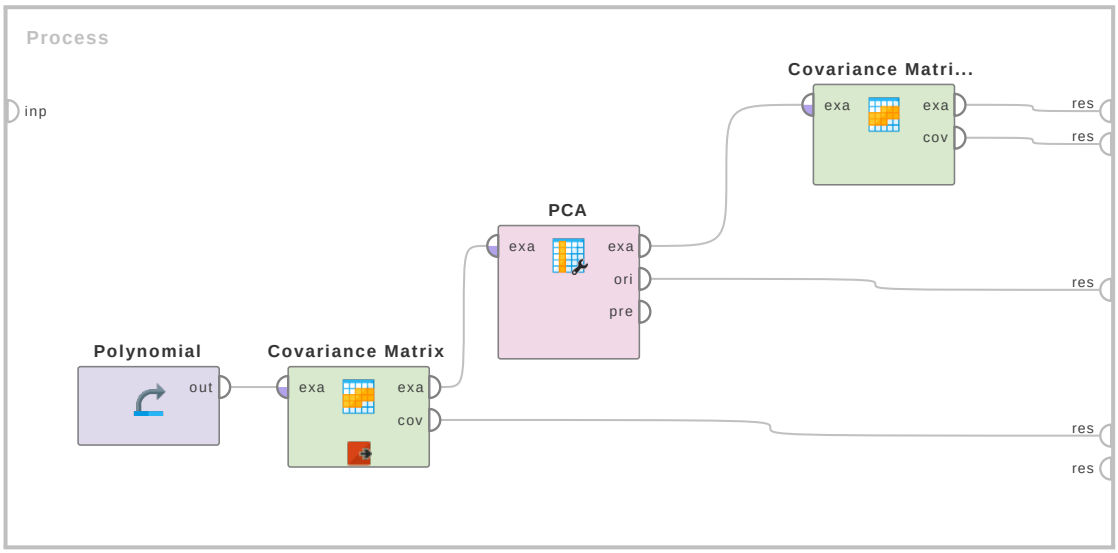
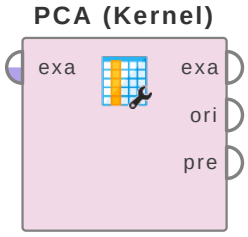



Figure 3.20: Tutorial process 'Dimensionality reduction of the Polynomial data set using the Principal Component Analysis operator'.

not required here. The Principal Component Analysis operator is applied on the 'Polynomial' data set. The dimensionality reduction parameter is set to 'fixed number' and the number of components parameter is set to 4. Thus the resultant ExampleSet will be composed of 4 principal components. As mentioned in the description, the principal components are uncorrelated with each other thus their covariance should be zero. The Covariance Matrix operator is applied on the output of the Principal Component Analysis operator. You can see the covariance matrix of the resultant ExampleSet in the Results Workspace. As you can see that the covariance of the components is zero.

# Principal Component Analysis (Kernel)




This operator performs Kernel Principal Component Analysis (PCA) which is a non-linear extension of PCA.

## Description

Kernel principal component analysis (kernel PCA) is an extension of principal component analysis (PCA) using techniques of kernel methods. Using a kernel, the originally linear operations of PCA are done in a reproducing kernel Hilbert space with a non-linear mapping. By the use of integral operator kernel functions, one can efficiently compute principal components in high-dimensional feature spaces, related to input space by some nonlinear map. The result will be the set of data points in a non-linearly transformed space. Please note that in contrast to the usual linear PCA the kernel variant also works for large numbers of attributes but will become slow for large number of examples.

RapidMiner provides the Principal Component Analysis operator for applying linear PCA. Principal Component Analysis is a mathematical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated attributes into a set of values of uncorrelated attributes called principal components. This transformation is defined in such a way that the first principal component's variance is as high as possible (accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it should be orthogonal to (uncorrelated with) the preceding components.

## Differentiation

- **Principal Component Analysis** Kernel principal component analysis (kernel PCA) is an extension of principal component analysis (PCA) using techniques of kernel methods. In contrast to the usual linear PCA the kernel variant also works for large numbers of attributes but will become slow for large number of examples. See page 393 for details.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along with the data. Please note that this operator cannot handle nominal attributes; it works on numerical attributes.

## Output Ports

**example set output *(exa)*** The kernel-based Principal Component Analysis is performed on the input ExampleSet and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model *(pre)*** This port delivers the preprocessing model, which has the information regarding the parameters of this operator in the current process.

## Parameters

**kernel type *(selection)*** The type of the kernel function is selected through this parameter. Following kernel types are supported: *dot, radial, polynomial, neural, anova, epachnenikov, gaussian combination, multiquadric*

- **dot** The dot kernel is defined by*k(x,y)=x*y* i.e. it is inner product of*x* and *y*.
- **radial** The radial kernel is defined by *exp(-g ||x-y||^2)* where *g* is the *gamma*, it is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by *k(x,y)=(x*y+1)^d* where *d* is the degree of polynomial and it is specified by the *kernel degree* parameter. The polynomial kernels are well suited for problems where all the training data is normalized.
- **neural** The neural kernel is defined by a two layered neural net *tanh(a x*y+b)* where *a* is *alpha* and *b* is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is 1/N, where N is the data dimension. Note that not all choices of *a* and *b* lead to a valid kernel function.
- **anova** The anova kernel is defined by raised to power *d* of summation of *exp(-g (x-y))* where *g* is *gamma* and *d* is *degree*. gamma and degree are adjusted by the *kernel gamma* and *kernel degree* parameters respectively.
- **epachnenikov** The epachnenikov kernel is this function *(3/4)(1-u2)* for *u* between -1 and 1 and zero for *u* outside that range. It has two adjustable parameters *kernel sigma1* and *kernel degree*.
- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters *kernel sigma1, kernel sigma2* and *kernel sigma3*.
- **multiquadric** The multiquadric kernel is defined by the square root of *||x-y||^2 + c^2*. It has adjustable parameters *kernel sigma1* and *kernel sigma shift*.

**kernel gamma *(real)*** This is the kernel parameter gamma. This is only available when the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1 *(real)*** This is the kernel parameter sigma1. This is only available when the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2 *(real)*** This is the kernel parameter sigma2. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3 *(real)*** This is the kernel parameter sigma3. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel shift *(real)*** This is the kernel parameter shift. This is only available when the *kernel type* parameter is set to *multiquadric*.

**kernel degree *(real)*** This is the kernel parameter degree. This is only available when the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a *(real)*** This is the kernel parameter a. This is only available when the *kernel type* parameter is set to *neural*.

**kernel b *(real)*** This is the kernel parameter b. This is only available when the *kernel type* parameter is set to *neural*.

## Related Documents

- **Principal Component Analysis** (page 393)

## Tutorial Processes

### Introduction to the Principal Component Analysis (Kernel) operator

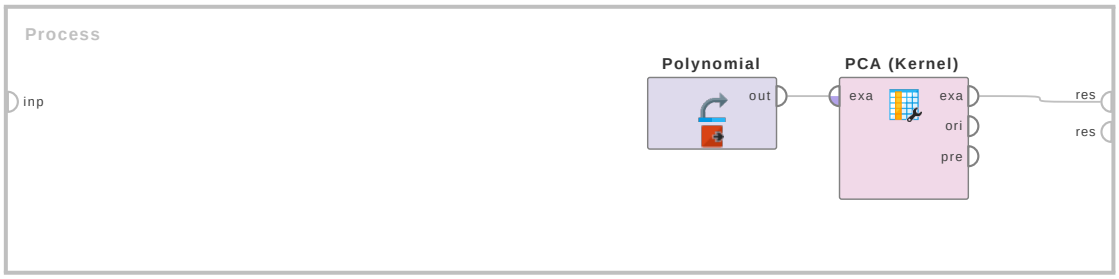
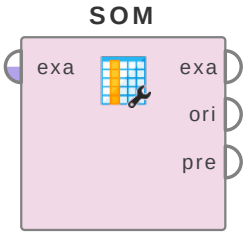



Figure 3.21: Tutorial process 'Introduction to the Principal Component Analysis (Kernel) operator'.

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes. The Principal Component Analysis (Kernel) operator is applied on this ExampleSet with default values of all parameters. The kernel type parameter is set to 'radial' and the kernel gamma parameter is set to 1.0. The resultant ExampleSet can be seen in the Results Workspace. You can see that this ExampleSet has a different set of attributes.

# Self-Organizing Map




This operator performs a dimensionality reduction of the given ExampleSet based on a self-organizing map (SOM). The user can specify the required number of dimensions.

## Description

A self-organizing map (SOM) or self-organizing feature map (SOFM) is a type of artificial neural network that is trained using unsupervised learning to produce a low-dimensional (typically two-dimensional), discretized representation of the input space of the training samples, called a map. Self-organizing maps are different from other artificial neural networks in the sense that they use a neighborhood function to preserve the topological properties of the input space. This makes SOMs useful for visualizing low-dimensional views of high-dimensional data, akin to multidimensional scaling. The model was first described as an artificial neural network by Teuvo Kohonen, and is sometimes called a Kohonen map.

Like most artificial neural networks, SOMs operate in two modes: training and mapping. Training builds the map using input examples. Mapping automatically classifies a new input vector. A self-organizing map consists of components called nodes or neurons. Associated with each node is a weight vector of the same dimension as the input data vectors and a position in the map space. The usual arrangement of nodes is a regular spacing in a hexagonal or rectangular grid. The self-organizing map describes a mapping from a higher dimensional input space to a lower dimensional map space. The procedure for placing a vector from data space onto the map is to first find the node with the closest weight vector to the vector taken from data space. Once the closest node is located it is assigned the values from the vector taken from the data space.

While it is typical to consider this type of network structure as related to feed-forward networks where the nodes are visualized as being attached, this type of architecture is fundamentally different in arrangement and motivation.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along with the data. Please note that this operator cannot handle nominal attributes; it works on numerical attributes.

## Output Ports

**example set output *(exa)*** The dimensionality reduction of the given ExampleSet is performed based on a self-organizing map and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators

or to view the ExampleSet in the Results Workspace.

**preprocessing model *(pre)*** This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

## Parameters

**return preprocessing model *(boolean)*** This parameter indicates if the preprocessing model should be returned.

**number of dimensions *(integer)*** This parameter specifies the number of dimensions to keep i.e. the number of attributes of the resultant ExampleSet.

**net size *(integer)*** This parameter specifies the size of the SOM net, by setting the length of every edge of the net. In total, there will be *net size* to the power of *number of dimensions* nodes in the net.

**training rounds *(integer)*** This parameter specifies the number of training rounds.

**learning rate start *(real)*** This parameter specifies the strength of an adaption in the first round. The strength will decrease every round until it reaches the *learning rate end* in the last round.

**learning rate end *(real)*** This parameter specifies the strength of an adaption in the last round. The strength will decrease to this value in last round, beginning with *learning rate start* in the first round.

**adaption radius start *(real)*** This parameter specifies the radius of the sphere around a stimulus in the first round. This radius decreases every round, starting by *adaption radius start* in the first round, to *adaption radius end* in the last round.

**adaption radius end *(real)*** This parameter specifies the radius of the sphere around a stimulus in the last round. This radius decreases every round, starting by *adaption radius start* in the first round, to *adaption radius end* in the last round.

## Tutorial Processes

### Dimensionality reduction of the Sonar data set using the Self-Organizing Map operator

The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 60 attributes. The Self-Organizing Map operator is applied on the 'Sonar' data set. The number of dimensions parameter is set to 2. Thus the resultant ExampleSet will be composed of 2 dimensions (artificial attributes). You can see the resultant ExampleSet in the Results Workspace and verify that it has only 2 attributes. Please note that these attributes are not original attributes of the 'Sonar' data set. These attributes were created using the SOM procedure.

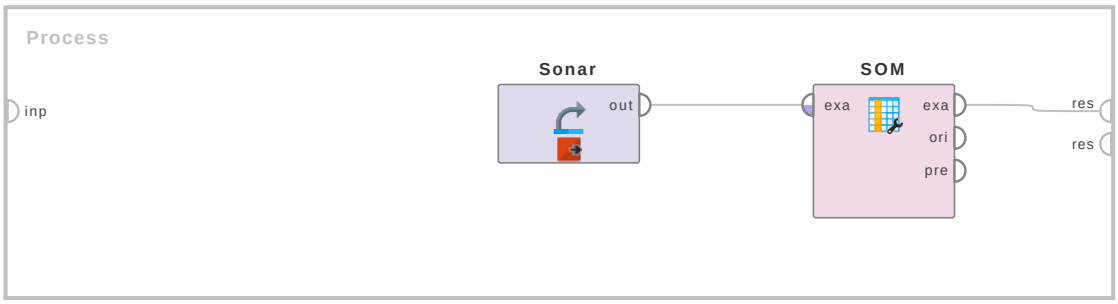
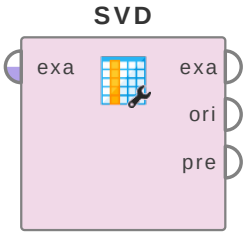



Figure 3.22: Tutorial process 'Dimensionality reduction of the Sonar data set using the Self-Organizing Map operator'.

## Singular Value Decomposition




This operator performs a dimensionality reduction of the given ExampleSet based on Singular Value Decomposition (SVD). The user can specify the required number of dimensions or specify the cumulative variance threshold. In the latter case all components having cumulative variance above this threshold are discarded.

### Description

Singular Value Decomposition (SVD) can be used to better understand an ExampleSet by showing the number of important dimensions. It can also be used to simplify the ExampleSet by reducing the number of attributes of the ExampleSet. This reduction removes unnecessary attributes that are linearly dependent in the point of view of Linear Algebra. It is useful when you have obtained data on a number of attributes (possibly a large number of attributes), and believe that there is some redundancy in those attributes. In this case, redundancy means that some of the attributes are correlated with one another, possibly because they are measuring the same construct. Because of this redundancy, you believe that it should be possible to reduce the observed attributes into a smaller number of components (artificial attributes) that will account for most of the variance in the observed attributes. For example, imagine an ExampleSet which contains an attribute that stores the water's temperature on several samples and another that stores its state (solid, liquid or gas). It is easy to see that the second attribute is dependent on the first attribute and, therefore, SVD could easily show us that it is not important for the analysis.

RapidMiner provides various dimensionality reduction operators e.g. the Principal Component Analysis operator. The Principal Component Analysis technique is a specific case of SVD. It is a mathematical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated attributes into a set of values of uncorrelated attributes called principal components. The number of principal components is less than or equal to the number of original attributes. This transformation is defined in such a way that the first principal component's variance is as high as possible (accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it should be orthogonal to (uncorrelated with) the preceding components.

## Differentiation

- **Principal Component Analysis** PCA is a dimensionality reduction procedure. PCA is a specific case of SVD. See page 393 for details.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along with the data. Please note that this operator cannot handle nominal attributes; it works on numerical attributes.

## Output Ports

**example set output *(exa)*** The Singular Value Decomposition is performed on the input ExampleSet and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model *(pre)*** This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

## Parameters

**dimensionality reduction *(selection)*** This parameter indicates which type of dimensionality reduction (reduction in number of attributes) should be applied.

- **none** if this option is selected, dimensionality reduction is not performed.
- **keep_percentage** if this option is selected, all the components with a cumulative variance greater than the given threshold are removed from the ExampleSet. The threshold is specified by the *percentage threshold* parameter.
- **fixed_number** if this option is selected, only a fixed number of components are kept. The number of components to keep is specified by the *dimensions* parameter.

**percentage threshold *(real)*** This parameter is only available when the *dimensionality reduction* parameter is set to 'keep percentage'. All the components with a cumulative variance greater than the *percentage threshold* are removed from the ExampleSet.

**dimensions *(integer)*** This parameter is only available when the *dimensionality reduction* parameter is set to 'fixed number'. The number of components to keep is specified by the *dimensions* parameter.

## Related Documents

- **Principal Component Analysis** (page 393)

## Tutorial Processes

### Dimensionality reduction of the Sonar data set using the Singular Value Decomposition operator

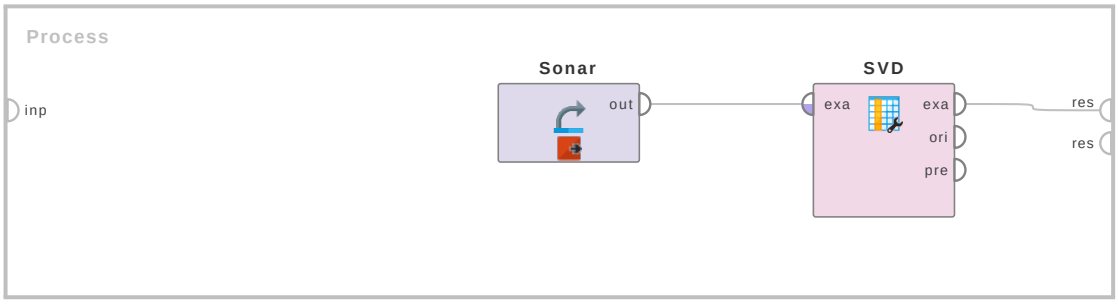


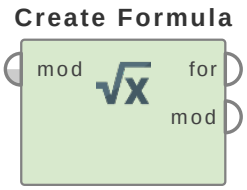

Figure 3.23: Tutorial process 'Dimensionality reduction of the Sonar data set using the Singular Value Decomposition operator'.

The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 60 attributes. The Singular Value Decomposition operator is applied on the 'Sonar' data set. The dimensionality reduction parameter is set to 'fixed number' and the dimensions parameter is set to 10. Thus the resultant ExampleSet will be composed of 10 dimensions (artificial attributes). You can see the resultant ExampleSet in the Results Workspace and verify that it has only 10 attributes. Please note that these attributes are not original attributes of the 'Sonar' data set. These attributes were created using the SVD procedure.

# 4Modeling

## 4.1 Predictive

### Create Formula




This operator generates a formula from the given model. This operator can generate formula only for models that are capable of producing formula.

#### Description

The Create Formula operator extracts a prediction calculation formula from the given model and stores the formula in a formula result object which can then be written into a file, e.g. with the Write operator. Please note that not all RapidMiner models provide a formula and this operator is applicable on only those models that are capable of producing formula.

#### Input Ports

**model *(mod)*** This input port expects a model. The model should be capable of providing a formula.

#### Output Ports

**formula *(for)*** The formula of the given model is passed to the output through this port.

**model *(mod)*** The model that was given as input is passed without changing to the output through this port. This is usually used to reuse the same model in further operators or to view the model in the Results Workspace.

#### Tutorial Processes

##### Formula of the Logistic Regression model

The 'Ripley-Set' data set is loaded using the Retrieve operator. The Logistic Regression operator is applied on this ExampleSet with default values of all parameters. The regression model generated by the Logistic Regression operator is provided as input to the Create Formula operator which returns a formula object. You can view this formula object in the Results Workspace. It is important to note that most RapidMiner operators do not provide a formula, thus this operator cannot be applied on them.

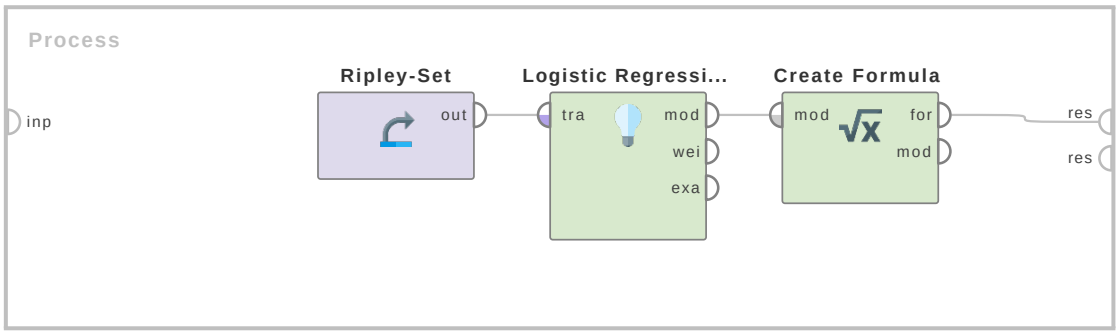
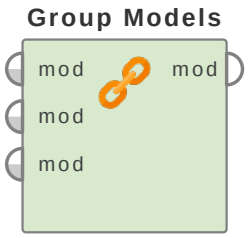



Figure 4.1: Tutorial process 'Formula of the Logistic Regression model'.

## Group Models




This operator groups the given models into a single combined model. When this combined model is applied, it is equivalent to applying the original models in their respective order.

### Description

The Group Models operator groups all input models together into a single combined model. This combined model can be applied on ExampleSets (using the Apply Model operator) like other models. When this combined model is applied, it is equivalent to applying the original models in their respective order. This combined model can also be written into a file using the Write Model operator. This operator is useful in cases where preprocessing and prediction models should be applied together on new and unseen data. A grouped model can be ungrouped with the Ungroup Models operator. Please study the attached Example Process for more information about the Group Models operator.

### Input Ports

**model in *(mod)*** This input port expects a model. This operator can have multiple inputs but it is mandatory to provide at least two models to this operator as input. When one input is connected, another *model in* port becomes available which is ready to accept another model(if any). The order of models remains the same i.e. the model supplied at the first *model in* port of this operator will be the first model to be applied when the resultant combined model is applied.

### Output Ports

**model out *(mod)*** The given models are grouped into a single combined model and the resultant grouped model is returned from this port.

## Tutorial Processes

### Grouping models and applying the resultant grouped model

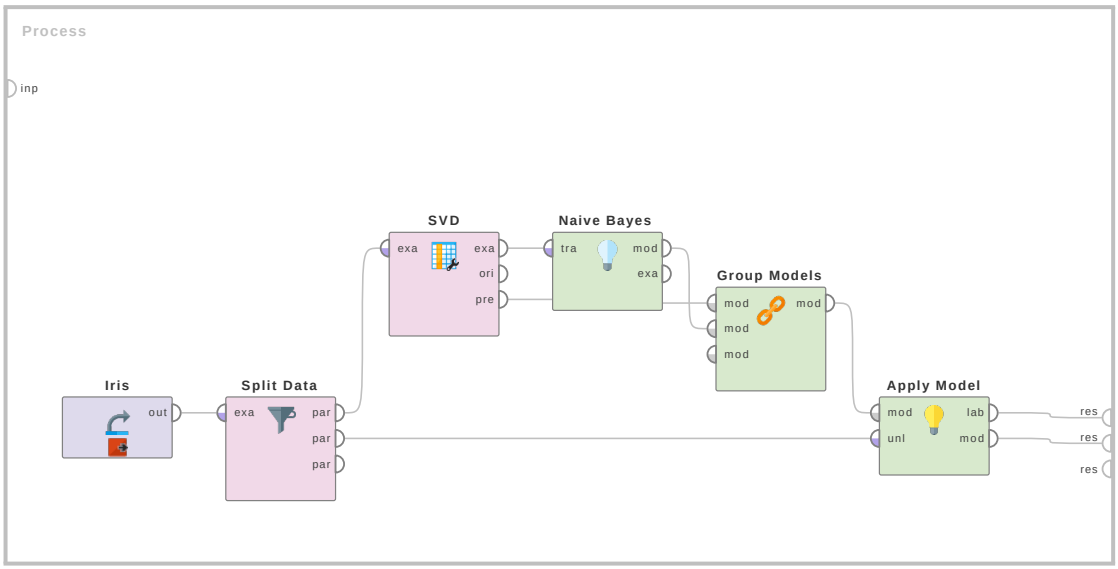
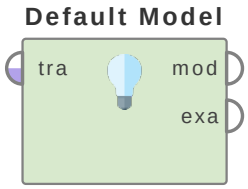



Figure 4.2: Tutorial process ‘Grouping models and applying the resultant grouped model’.

The ‘Iris’ data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has four regular attributes. The Split Data operator is applied on it to split the ExampleSet into training and testing data sets. The training data set (composed of 70% of examples) is passed to the SVD operator. The dimensionality reduction and dimensions parameters of the SVD operator are set to ‘fixed number’ and 2 respectively. Thus the given data set will be reduced to a data set with two dimensions (artificial attributes that represent the original attributes). The SVD model (model that reduces the dimensionality of the given ExampleSet) is provided as the first model to the Group Models operator. The Naive Bayes operator is applied on the resultant ExampleSet (i.e. the training data set with reduced dimensions). The classification model generated by the Naive Bayes operator is provided as the second model to the Group Models operator. Thus the Group Models operator combines two models SVD dimensionality reduction model Naive Bayes classification model. This combined model is applied on the testing data set (composed of 30% of the ‘Iris’ data set) using the Apply Model operator. When the combined model is applied, the SVD model is applied first on the testing data set. Then the Naive Bayes classification model is applied on the resultant ExampleSet (i.e. the testing data set with reduced dimensions). The combined model and the labeled ExampleSet can be seen in the Results Workspace after the execution of the process.

## 4.1.1 Lazy

## Default Model




This operator generates a model that provides the specified default value as prediction.

### Description

The Default Model operator generates a model that predicts the specified default value for the label in all examples. The method to use for generating a default value can be selected through the *method* parameter. For a numeric label, the default value can be median or average of the label values or a constant default value can be specified through the *constant* parameter. For nominal values the mode of the labels can be used. Values of an attribute can be used as predictions; the attribute can be selected through the *attribute* parameter. This operator should not be used for 'actual' prediction tasks, but it can be used for comparing the results of 'actual' learning schemes with guessing.

### Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

### Output Ports

**model *(mod)*** The default model is delivered from this output port. This model can now be applied on unseen data sets for the prediction of the *label* attribute. This model should not be used for 'actual' prediction tasks, but it can be used for comparing the results of 'actual' learning schemes with guessing.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

### Parameters

**method *(selection)*** This parameter specifies the method for computing the default values. For a numeric label, the default value can be median or average of the label values or a constant default value can be specified through the *constant* parameter. For nominal values the mode of the labels can be used. Values of an attribute can be used as predictions; the attribute can be selected through the *attribute* parameter.

**constant *(real)*** This parameter is only available when the *method* parameter is set to 'constant'. This parameter specifies a constant default value for a numeric label.

**attribute *(string)*** This parameter is only available when the *method* parameter is set to 'attribute'. This parameter specifies the attribute to get the predicted values from. If applied on a nominal label, it should be made sure that the selected attribute has the same set of possible values as the label.

## Tutorial Processes

### Using the Default Model operator with 'mode' method

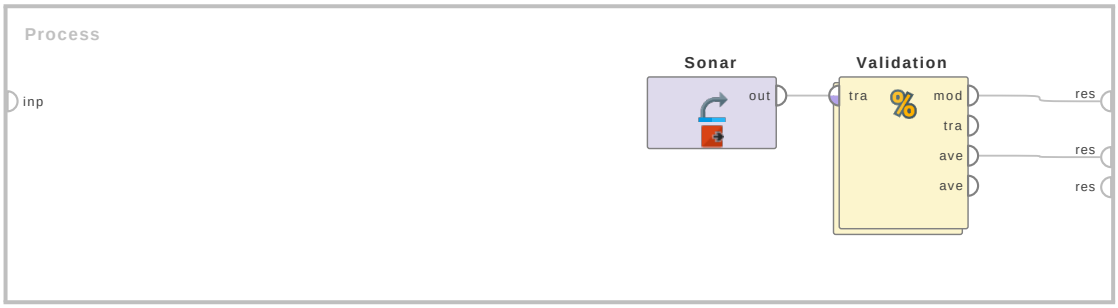
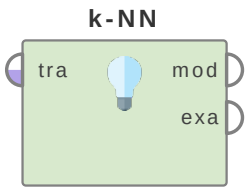



Figure 4.3: Tutorial process 'Using the Default Model operator with 'mode' method'.

The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that there are two possible label values i.e. 'Rock' and 'Mine'. The most frequently occurring label value is 'Mine'. The Split Validation operator is applied on this ExampleSet for training and testing a classification model. The Default Model operator is applied in the training subprocess of the Split Validation operator. The method parameter of the Default Model operator is set to 'mode', thus the most frequently occurring label value (i.e. 'Mine') will be used as prediction in all examples. The Apply Model operator is used in the testing subprocess for applying the model generated by the Default Model operator. A breakpoint is inserted here so that you can have a look at the labeled ExampleSet. You can see that all examples have been predicted as 'Mine'. This labeled ExampleSet is used by the Performance operator for measuring the performance of the model. The default model and its performance vector are connected to the output and they can be seen in the Results Workspace.

# k-NN




This Operator generates a k-Nearest Neighbor model, which is used for classification or regression.

## Description

The k-Nearest Neighbor algorithm is based on comparing an unknown Example with the *k* training Examples which are the nearest neighbors of the unknown Example.

The first step of the application of the k-Nearest Neighbor algorithm on a new Example is to find the k closest training Examples. "Closeness" is defined in terms of a distance in the n-dimensional space, defined by the n Attributes in the training ExampleSet.

Different metrices, such as the Euclidean distance, can be used to calculate the distance between the unknown Example and the training Examples. Due to the fact that distances often depends on absolut values, it is recommended to normalize data before training and applying the k-Nearest Neighbor algorithm. The metric used and its exact configuration are defined by the parameters of the Operator.

In the second step, the k-Nearest Neighbor algorithm classify the unknown Example by a majority vote of the found neighbors. In case of a regression, the predicted value is the average of the values of the found neighbors.

It can be useful to weight the contributions of the neighbors, so that the nearer neighbors contribute more to the average than the more distant ones.

## Differentiation

- **k-Means (Deprecated)**

  This Operator performs clustering on an unlabeled data set. It can be considered as the equivalent of the k-NN algorithm for unsupervised learning. A cluster calculated by the k-Means algorithm consists of similar Examples, where the similarity is calculated by a given measure in the Attribute space.

  See page **??** for details.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet.

## Output Ports

**model *(mod)*** The k-Nearest Neighbor model is delivered from this output port. This model can now be applied on unseen data sets for prediction of the label Attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed through without changes.

## Parameters

**k** Finding the k training Examples that are closest to the unknown Example is the first step of the k-NN algorithm. If k = 1, the Example is simply assigned to the class of its nearest neighbor. k is typically a small, positive and odd integer.

**weighted vote** If this parameter is set, the distance values between the Examples are also taken into account for the prediction. It can be useful to weight the contributions of the neighbors, so that nearer neighbors contribute more than more distant ones.

**measure types** This parameter is used for selecting the type of measure to be used for finding the nearest neighbors. The following options are available:

- **MixedMeasures** Mixed Measures are used to calculate distances in case of both nominal and numerical Attributes.
- **NominalMeasures** In case of only nominal Attributes different distance metrices can be used to calculate distances on this nominal Attributes.
- **NumericalMeasures** In case of only numerical Attributes different distance metrices can be used to calculate distances on this numerical Attributes.
- **BregmannDivergences** Bregmann divergences are more generic "closeness" measure types with does not satisfy the triangle inequality or symmetry. For more details see the parameter *divergence*.

**mixed measure** The only available option for *mixed measure* is the 'Mixed Euclidean Distance'. For numerical values the euclidean distance is calculated. For nomimal values, a distance of 0 is taken if both values are the same and a distance of one is taken otherwise. This parameter is available when the *measure type* parameter is set to 'mixed measures'.

**nominal measure** This parameters defines how to calculate distances for only nominal Attributes in the input ExampleSet, in case the *measure type* is set to nominal measure. In case of using a similarity as a distance measure, the actual distance is calculated as the negative similarity. For the different similarities the following variables are defined:

e: number of Attribute for which both Examples have equal and non-zero values

u: number of Attribute for which both Examples have not equal values

z: number of Attribute for which both Examples have zero values

- **NominalDistance** Distance of two values is 0 if both values are the same and 1 otherwise.
- **DiceSimilarity** With the above mentioned defintions the *DiceSimilarity* is: 2*e/(2*e+u)
- **JaccardSimilarity** With the above mentioned defintions the *JaccardSimilarity* is: e/(e+u)
- **KulczynskiSimilarity** With the above mentioned defintions the *KulczynskiSimilarity* is: e/u
- **RogersTanimotoSimilarity** With the above mentioned defintions the *RogersTanimotoSimilarity* is: (e+z)/(e+2*u+z)
- **RussellRaoSimilarity** With the above mentioned defintions the *RussellRaoSimilarity* is: e/(e+u+z)
- **SimpleMatchingSimilarity** With the above mentioned defintions the *SimpleMatchingSimilarity* is: (e+z)/(e+u+z)

**numerical measure** This parameters defines how to calculate distances for only numerical Attributes in the input ExampleSet, in case the *measure type* is set to numerical measure. For the different distance measures the following variable is defined:

y(i,j) : Value of the j.th Attribute of the i.th Example. Hence y(1,3) - y(2,3) is the difference of the values of the third Attribute of the first and second Example.

In case of using a similarity as a distance measure, the actual distance is calculated as the negative similarity.

- **EuclideanDistance** Square root of the sum of quadratic differences over all Attributes. Dist = Sqrt ( Sum_(j=1) [y(1,j)-y(2,j)]^2 )
- **CanberraDistance** Sum over all Attributes. The summand is the absolut of the difference of the value, divided by the sum of the absolute values. Dist = Sum_(j=1) |y(1,j)-y(2,j)| / (|y(1,j)|+|y(2,j)|) The CanberraDistance is often used to compare ranked list or for intrusion detection in computer security.
- **ChebychevDistance** Maximum of all differences of all Attributes. Dist = max_(j=1) (|y(1,j)-y(2,j)|)
- **CorrelationSimilarity** The similarity is calculated as the correlation between the Attribute vectors of the two Examples.
- **CosineSimilarity** Similarity measure measuring the cosine of the angle between the Attribute vectors of the two Examples.
- **DiceSimilarity** The DiceSimilarity for numerical Attributes is calculated as 2*Y1Y2/(Y1+Y2). Y1Y2 = Sum over product of values = Sum_(j=1) y(1,j)*y(2,j). Y1 = Sum over values of first Example = Sum_(j=1) y(1,j) Y2 = Sum over values of second Example = Sum_(j=1) y(2,j)
- **DynamicTimeWarpingDistance** Dynamic Time Warping is often use in Time Series analysis for measuring the distance between two temporal sequences. Here the distance on an optimal "warping" path from the Attribute vector of the first Example to the second Example is calculated.
- **InnerProductSimilarity** The similarity is calculated as the sum of the product of the Attribute vectors of the two Examples. Dist = -Similarity = -Sum_(j=1) y(1,j)*y(2,j)
- **JaccardSimilarity** The JaccardSimilarity is calculated as Y1Y2/(Y1+Y2-Y1Y2). See *DiceSimilarity* for the definition of Y1Y2, Y1 and Y2.
- **KernelEuclideanDistance** The distance is calculated by the euclidean distance of the two Examples, in a transformed space. The transformation is defined by the chosen kernel and configured by the parameters *kernel type, gamma, sigma1, sigma2, sigma 3, shift, degree, a, b*.
- **ManhattanDistance** Sum of the absolute distances of the Attribute values. Dist = Sum_(j=1) |y(1,j)-y(2,j)|
- **MaxProductSimilarity** The similarity is the maximum of all products of all Attribute values. If the maximum is less or equal to zero the similarity is not defined. Dist = -Similarity = -max_(j=1) (y(1,j)*y(2,j))
- **OverlapSimilarity** The similarity is a variant of simple matching for numerical Attributes and is calculated as minY1Y2 / min(Y1,Y2). See *DiceSimilarity* for the definition of Y1, Y2. minY1Y2 = Sum over the minimum of values = Sum_(j=1) min [y(1,j),y(2,j)] .

**divergence** This parameter defines which type of Bregmann divergence is used when the *measure type* parameter is set to 'bregman divergences'. For the different distance measures the following variable is defined:

y(i,j) : Value of the j.th Attribute of the i.th Example. Hence y(1,3) - y(2,3) is the difference of the values of the third Attribute of the first and second Example.

- **GeneralizedIDivergence** The distance is caluclated as Sum1 Sum2. It is not applicable if any Attribute value is less or equal to 0. Sum1 = Sum_(j=1) y(1,j)*ln[y(1,j)/y(2,j)] Sum2 = Sum_(j=1) [y(1,j)-y(2,j)]
- **ItakuraSaitoDistance** The ItakuraSaitoDistance can only be calculated for ExampleSets with 1 Attribute and values larger 0. Dist = y(1,1)/y(2,1)-ln[y(1,1)/y(2,1)]-1
- **KLDivergence** The Kullback-Leibler divergence is a measure of how one probability distribution diverges from a second expected probability distribution. Dist = Sum_(j=1) [y(1,j)*log_2(y(1,j)/y(2,j))]
- **LogarithmicLoss** The LogarithmicLoss can only be calculated for ExampleSets with 1 Attribute and values larger 0. Dist = y(1,1)*ln[y(1,1)/y(2,1)]-(y(1,1)-y(2,1))
- **LogisticLoss** The LogisticLoss can only be calculated for ExampleSets with 1 Attribute and values larger 0. Dist = y(1,1)*ln[y(1,1)/y(2,1)]+(1-y(1,1))*ln[(1-y(1,1))/(1-y(2,1))]
- **MahalanobisDistance** The Mahalanobis distance measures the distance between the two Examples under the assumption they are both random vectors of the same distribution. Therefore the covariance matrix S is calculated on the whole ExampleSet and the Distance is calculated as: Dist = Sqrt [ (vecY1-vecY2) S (vecY1-vecY2) ] vecY1 = Attribute vector of Example 1 vecY2 = Attribute vector of Example 2
- **SquaredEuclideanDistance** Sum of quadratic differences over all Attributes. Dist = Sum_(j=1) [y(1,j)-y(2,j)]^2
- **SquaredLoss** The SquaredLoss can only be calculated for ExampleSets with 1 Attribute. Dist = [y(1,1)-y(2,1)]^2

**kernel type** This parameter is available only when the *numerical measure* parameter is set to 'Kernel Euclidean Distance'. The type of the kernel function is selected through this parameter. Following kernel types are supported:

- **dot** The dot kernel is defined by k(x,y) = x*y i.e. it is the inner product of x and y.
- **radial** The radial kernel is defined by k(x,y) = exp(-g*||x-y||^2) where g is gamma, specified by the *kernel gamma* parameter. The adjustable parameter gamma plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by k(x,y) = (x*y+1)^d where d is the degree of the polynomial and is specified by the _kernel degree_ parameter. Polynomial kernels are well suited for problems where all the training data is normalized.
- **sigmoid** This is a hyperbolic tangent sigmoid kernel. The distance is calculated as tanh[a*Y1Y2+b] where Y1Y2 is the inner product of the Attribute vector of the two Examples. a and b can be adjusted using the *kernel a* and *kernel b* parameters. A common value for a is 1/N, where N is the data dimension. Note that not all choices of a and b lead to a valid kernel function.
- **anova** The anova kernel is defined by the raised to the power d of summation of exp(-g(x-y)) where g is gamma and d is degree. The two are adjusted by the *kernel gamma* and *kernel degree* parameters respectively.

- **epachnenikov** The Epanechnikov kernel is this function (3/4)(1-u2) for u between -1 and 1 and zero for u outside that range. It has the two adjustable parameters kernel sigma1 and kernel degree.
- **gaussian_combination** This is the gaussian combination kernel. It has the adjustable parameters kernel sigma1, kernel sigma2 and kernel sigma3.
- **multiquadric** The multiquadric kernel is defined by the square root of ||x-y||^2+c^2. It has the adjustable parameters kernel sigma1 and kernel sigma shift.

**kernel gamma** This is the SVM kernel parameter gamma. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1** This is the SVM kernel parameter sigma1. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2** This is the SVM kernel parameter sigma2. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3** This is the SVM kernel parameter sigma3. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel shift** This is the SVM kernel parameter shift. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *multiquadric*.

**kernel degree** This is the SVM kernel parameter degree. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a** This is the SVM kernel parameter a. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

**kernel b** This is the SVM kernel parameter b. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

## Tutorial Processes

### Classification of the Golf-Testset data set using the K-NN Operator

This tutorial process demonstrate the usage of the k-NN Operator to classify the Golf-Testset from the Samples folder. The k-NN is trained on the Golf data set and applied to the independent Golf-Testset. Both data sets are retrieved from the Samples folder.

k is set to 3, so each Example in the Golf-Testset is classified as the majority class of its 3 nearest neighbors in the training set. For the distance measure the MixedEuclideanDistance (which is the default setting) is used.

For more details, see the comments in the process.

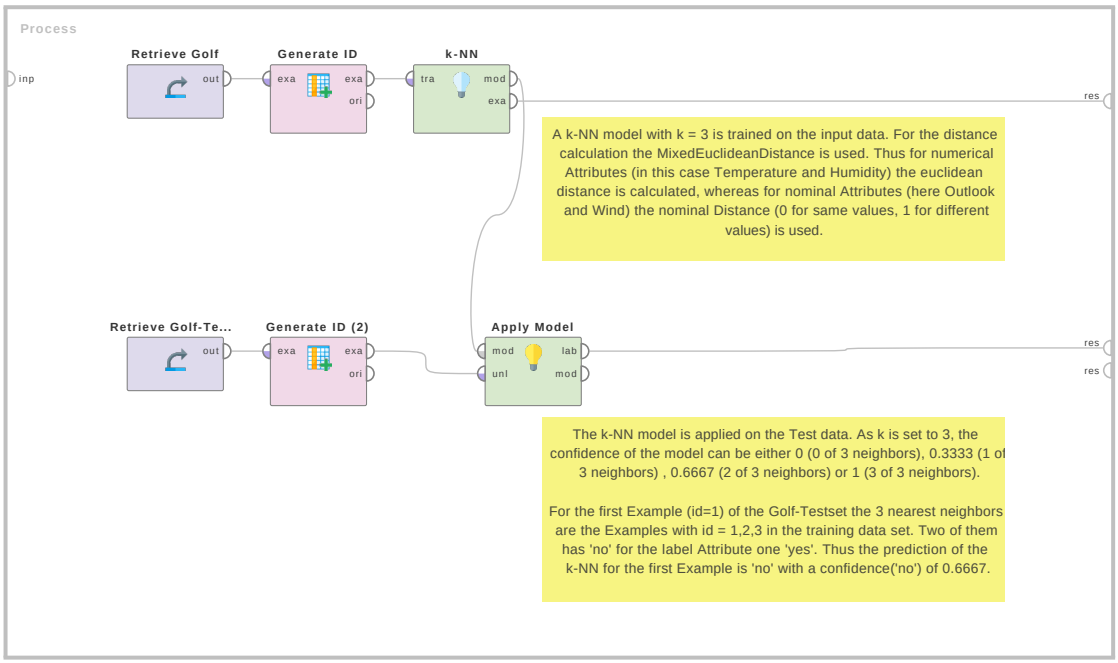



Figure 4.4: Tutorial process 'Classification of the Golf-Testset data set using the K-NN Operator'.

### Performance difference between normalized and not normalized data

This tutorial process demonstrate the enhancement of the performance of the k-NN in case the data is normalized before the k-NN is trained.

The Sonar data set is retrieved from the Samples folder and fed into two Cross Validation Operators. In both Cross Validations a k-NN model with k=3 is trained. But in the second the training data is normalized before the training of the k-NN. A Group Models Operator ensures that the test data is normalized in the same way, before the k-NN is applied on the test data.

Note that the accuracy of this k-NN is higher for the normalized values (84.24 % > 81.69 %).

### Optimizing k of a k-NN model and logging all results

In this tutorial process the parameter k of a k-NN model is optimized and results are logged to investigate the dependency of the performance on the parameter.

The Sonar data set is retrieved from the Samples folder. An Optimized Parameters (Grid) Operator is used to iterate over k (from 1 to 30) and calculate the best performing k-NN model. A Log Operator is used within the Cross Validation Operator in the Optimize Parameters (Grid) Operator to log the parameter k and the corresponding performance. The Log to Data Operator and the Aggregate Operator are used to convert this log entries into an ExampleSet, which can be investigated in the results view.

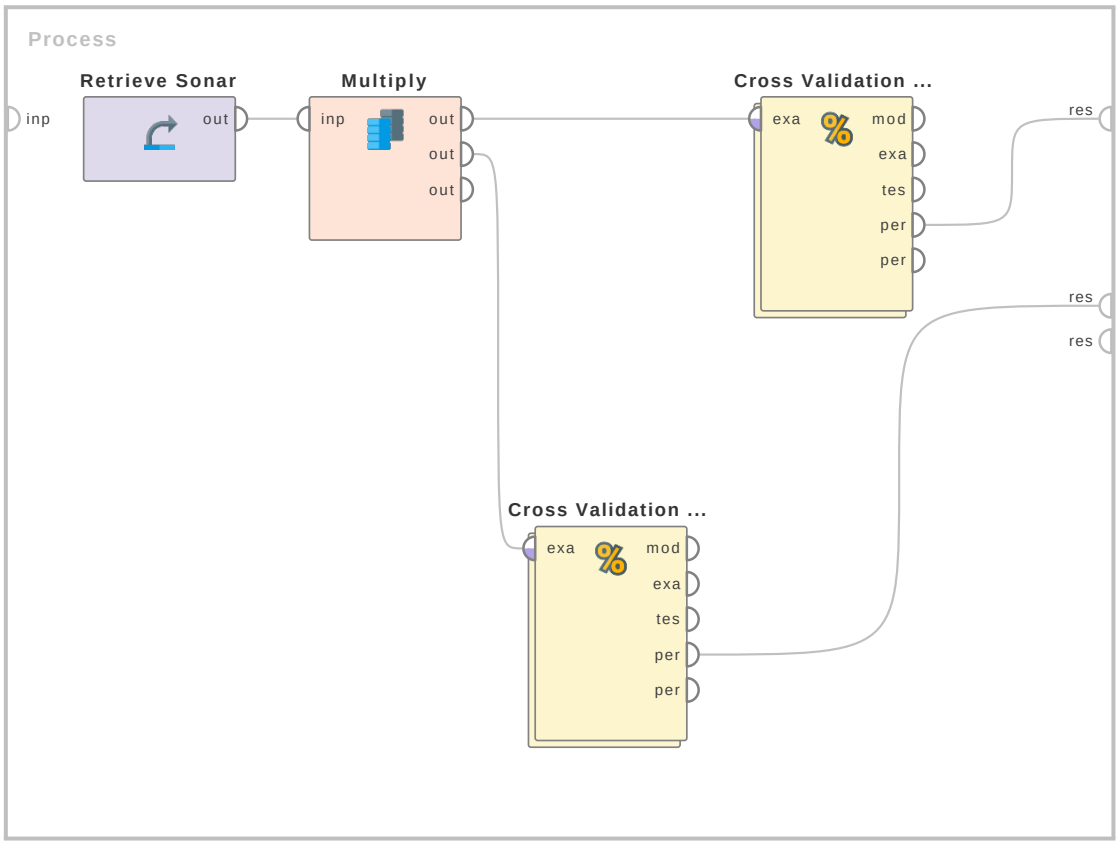
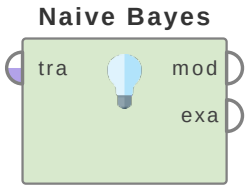



Figure 4.5: Tutorial process 'Performance difference between normalized and not normalized data'.

## 4.1.2 Bayesian

### Naive Bayes




This Operator generates a Naive Bayes classification model.

#### Description

Naive Bayes is a high-bias, low-variance classifier, and it can build a good model even with a small data set. It is simple to use and computationally inexpensive. Typical use cases involve text categorization, including spam detection, sentiment analysis, and recommender systems.

The fundamental assumption of Naive Bayes is that, given the value of the label (the class), the value of any Attribute is independent of the value of any other Attribute. Strictly speaking,

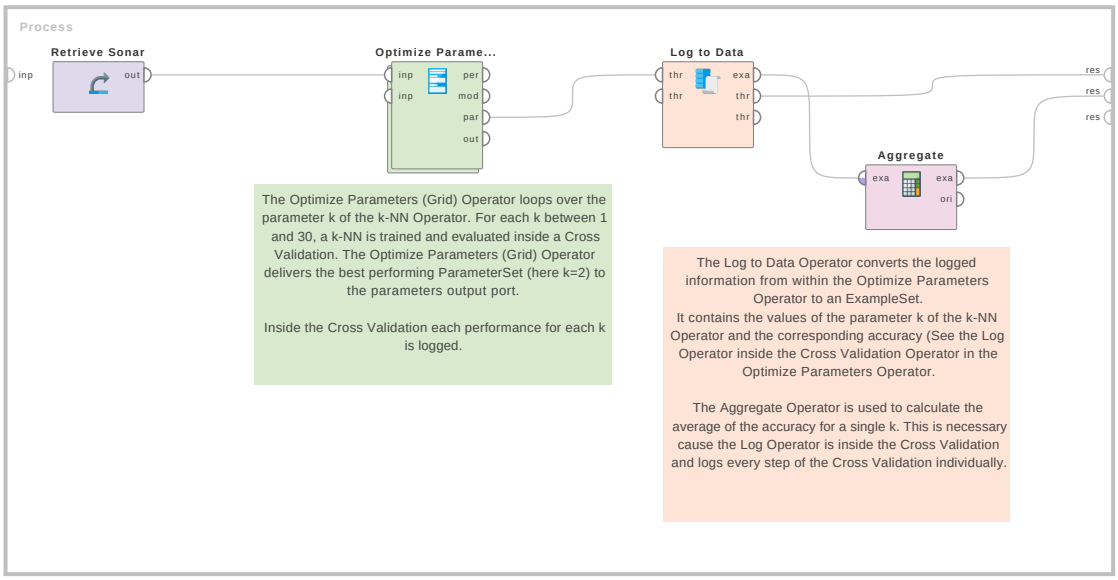



Figure 4.6: Tutorial process 'Optimizing k of a k-NN model and logging all results'.

this assumption is rarely true (it's "naive"!), but experience shows that the Naive Bayes classifier often works well. The independence assumption vastly simplifies the calculations needed to build the Naive Bayes probability model.

To complete the probability model, it is necessary to make some assumption about the conditional probability distributions for the individual Attributes, given the class. This Operator uses Gaussian probability densities to model the Attribute data.

## Differentiation

- **Naive Bayes (Kernel)**

  The alternative Operator Naive Bayes (Kernel) is a variant of Naive Bayes where multiple Gaussians are combined, to create a kernel density.

  See page 419 for details.

## Input Ports

**training set *(tra)*** The input port expects an ExampleSet.

## Output Ports

**model *(mod)*** The Naive Bayes classification model is delivered from this output port. The model can now be applied to unlabelled data to generate predictions.

**example set *(exa)*** The ExampleSet that was given as input is passed through without changes.

## Parameters

**laplace correction** The simplicity of Naive Bayes includes a weakness: if within the training data a given Attribute value never occurs in the context of a given class, then the conditional probability is set to zero. When this zero value is multiplied together with other probabilities, those values are also set to zero, and the results will be misleading. Laplace correction is a simple trick to avoid this problem, adding one to each count to avoid the occurrence of zero values. For most training sets, adding one to each count has only a negligible effect on the estimated probabilities.

## Tutorial Processes

### Apply Naive Bayes to the Iris Data Set

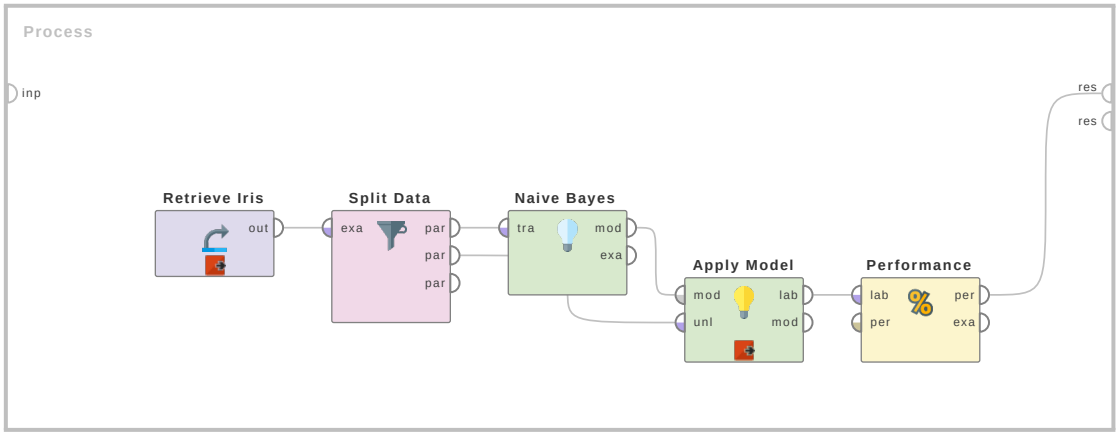



Figure 4.7: Tutorial process 'Apply Naive Bayes to the Iris Data Set'.

The Iris data set contains 150 Examples, corresponding to three different classes of Iris plant: Iris Setosa, Iris Versicolor, and Iris Virginica. There are 50 Examples for each class of Iris, and each Example includes 6 Attributes: the label, the id, and 4 real Attributes corresponding to physical characteristics of the plant.

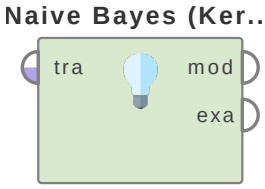
a1 = sepal length in cm
a2 = sepal width in cm
a3 = petal length in cm
a4 = petal width in cm

In the Tutorial Process, a predictive model for the Iris class is created, based on the plant's physical characteristics. When you run the Process, the output is displayed in three steps:

1. The whole Iris data set is displayed.
2. A subset of the Iris data set is displayed, together with the predictions based on Naive Bayes.
3. A confusion matrix is displayed, showing that the predictions are highly consistent with the data set (accuracy: 98.33%).

The Operator Split Data divides the original data set into two parts: one is used to train Naive Bayes, and the other to evaluate the model. The result shows that this simple model can generate a good fit to the Iris data set.

# Naive Bayes (Kernel)




This operator generates a Kernel Naive Bayes classification model using estimated kernel densities.

## Description

A Naive Bayes classifier is a simple probabilistic classifier based on applying Bayes' theorem (from Bayesian statistics) with strong (naive) independence assumptions. A more descriptive term for the underlying probability model would be the 'independent feature model'. In simple terms, a Naive Bayes classifier assumes that the presence (or absence) of a particular feature of a class (i.e. attribute) is unrelated to the presence (or absence) of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about 4 inches in diameter. Even if these features depend on each other or upon the existence of the other features, a Naive Bayes classifier considers all of these properties to independently contribute to the probability that this fruit is an apple. The Naive Bayes classifier performs reasonably well even if the underlying assumption is not true

The advantage of the Naive Bayes classifier is that it only requires a small amount of training data to estimate the means and variances of the variables necessary for classification. Because independent variables are assumed, only the variances of the variables for each *label* need to be determined and not the entire covariance matrix. In contrast to the Naive Bayes operator, the Naive Bayes (Kernel) operator can be applied on numerical attributes.

A kernel is a weighting function used in non-parametric estimation techniques. Kernels are used in kernel density estimation to estimate random variables' density functions, or in kernel regression to estimate the conditional expectation of a random variable.

Kernel density estimators belong to a class of estimators called non-parametric density estimators. In comparison to parametric estimators where the estimator has a fixed functional form (structure) and the parameters of this function are the only information we need to store, Non-parametric estimators have no fixed structure and depend upon all the data points to reach an estimate.

## Input Ports

**training set *(tra)*** The input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The Kernel Naive Bayes classification model is delivered from this output port. This classification model can now be applied on unseen data sets for prediction of the *label* attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**laplace correction** ***(boolean)*** This parameter indicates if Laplace correction should be used to prevent high influence of zero probabilities. There is a simple trick to avoid zero probabilities. We can assume that our training set is so large that adding one to each count that we need would only make a negligible difference in the estimated probabilities, yet would avoid the case of zero probability values. This technique is known as Laplace correction.

**estimation mode** ***(selection)*** This parameter specifies the kernel density estimation mode. Two options are available.

- **full** If this option is selected, you can select a bandwidth through heuristic or a fix bandwidth can be specified.
- **greedy** If this option is selected, you have to specify the minimum bandwidth and the number of kernels.

**bandwidth selection** ***(selection)*** This parameter is only available when the *estimation mode* parameter is set to 'full'. This parameter specifies the method to set the kernel bandwidth. The bandwidth can be selected through heuristic or a fix bandwidth can be specified. Please note that the bandwidth of the kernel is a free parameter which exhibits a strong influence on the resulting estimate. It is important to choose the most appropriate bandwidth as a value that is too small or too large is not useful.

**bandwidth** ***(real)*** This parameter is only available when the *estimation mode* parameter is set to 'full' and the *bandwidth selection* parameter is set to 'fix'. This parameter specifies the kernel bandwidth.

**minimum bandwidth** ***(real)*** This parameter is only available when the *estimation mode* parameter is set to 'greedy'. This parameter specifies the minimum kernel bandwidth.

**number of kernels** ***(integer)*** This parameter is only available when the *estimation mode* parameter is set to 'greedy'. This parameter specifies the number of kernels.

**use application grid** ***(boolean)*** This parameter indicates if the kernel density function grid should be used in the model application. It speeds up model application at the expense of the density function precision.

**application grid size** ***(integer)*** This parameter is only available when the *use application grid* parameter is set to true. This parameter specifies the size of the application grid.

## Tutorial Processes

### Introduction to the Naive Bayes (Kernel) operator

The 'Golf' data set is loaded using the Retrieve operator. The Naive Bayes (Kernel) operator is applied on it. All parameters of the Naive Bayes (Kernel) operator are used with default values. The model generated by the Naive Bayes (Kernel) operator is applied on the 'Golf-Testset' data set using the Apply Model operator. The results of the process can be seen in the Results Workspace. Please note that parameters should be carefully chosen for this operator to obtain better performance. Specially the bandwidth should be selected carefully.

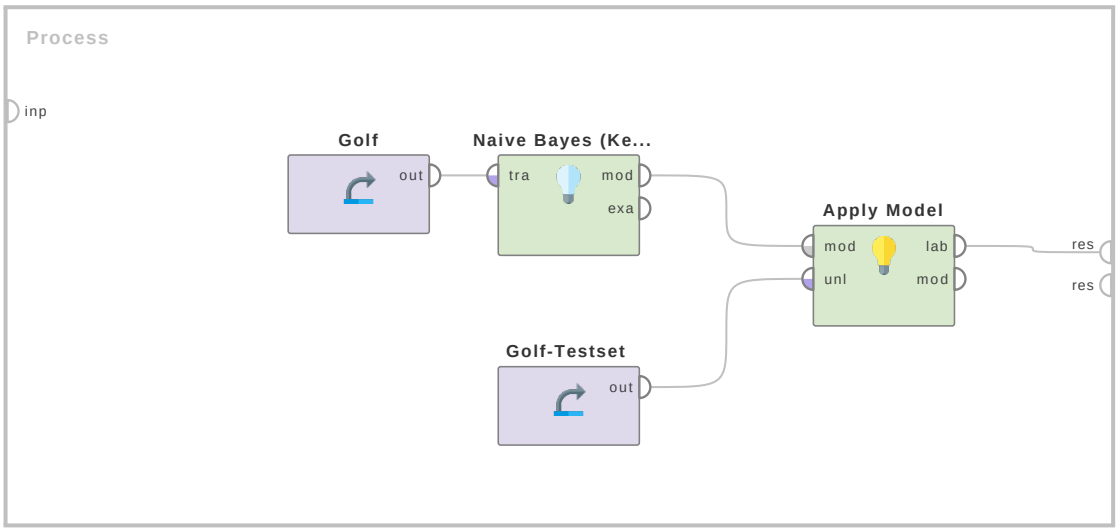
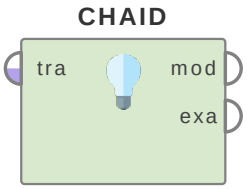



Figure 4.8: Tutorial process 'Introduction to the Naive Bayes (Kernel) operator'.

## 4.1.3 Trees

### CHAID




This operator generates a pruned decision tree based on the chi-squared attribute relevance test. This operator can be applied only on ExampleSets with nominal data.

#### Description

The CHAID decision tree operator works exactly like the Decision Tree operator with one exception: it uses a chi-squared based criterion instead of the information gain or gain ratio criteria. Moreover this operator cannot be applied on ExampleSets with numerical attributes. It is recommended that you study the documentation of the Decision Tree operator for basic understanding of decision trees.

CHAID stands for CHi-squared Automatic Interaction Detection. The chi-square statistic is a nonparametric statistical technique used to determine if a distribution of observed frequencies differs from the theoretical expected frequencies. Chi-square statistics use nominal data, thus instead of using means and variances, this test uses frequencies. CHAID's advantages are that its output is highly visual and easy to interpret. Because it uses multiway splits by default, it needs rather large sample sizes to work effectively, since with small sample sizes the respondent groups can quickly become too small for reliable analysis.

This representation of the data has the advantage compared with other approaches of being meaningful and easy to interpret. The goal is to create a classification model that predicts the value of the label based on several input attributes of the ExampleSet. Each interior node of the tree corresponds to one of the input attributes. The number of edges of an interior node is equal to the number of possible values of the corresponding input attribute. Each leaf node represents

a value of the label given the values of the input attributes represented by the path from the root to the leaf. This description can be easily understood by studying the Example Process of the Decision Tree operator.

Pruning is a technique in which leaf nodes that do not add to the discriminative power of the decision tree are removed. This is done to convert an over-specific or over-fitted tree to a more general form in order to enhance its predictive power on unseen datasets. Pre-pruning is a type of pruning performed parallel to the tree creation process. Post-pruning, on the other hand, is done after the tree creation process is complete.

## Differentiation

- The CHAID operator works exactly like the Decision Tree operator with one exception: it uses a chi-squared based criterion instead of the information gain or gain ratio criteria. Moreover this operator cannot be applied on ExampleSets with numerical attributes. See page **??** for details.
- **Decision Tree (Weight-Based)** If the Weight by Chi Squared Statistic operator is applied for attribute weighting in the subprocess of the Decision Tree (Weight-Based) operator, it works exactly like the CHAID operator. See page 433 for details.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Generate Nominal Data operator in the attached Example Process. The output of other operators can also be used as input. This operator cannot handle numerical data, therefore the ExampleSet should not have numerical attributes.

## Output Ports

**model *(mod)*** The CHAID Decision Tree is delivered from this output port. This classification model can now be applied on unseen data sets for the prediction of the *label* attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**minimal size for split *(integer)*** The size of a node is the number of examples in its subset. The size of the root node is equal to the total number of examples in the ExampleSet. Only those nodes are split whose size is greater than or equal to the *minimal size for split* parameter.

**minimal leaf size *(integer)*** The size of a leaf node is the number of examples in its subset. The tree is generated in such a way that every leaf node subset has at least the *minimal leaf size* number of instances.

**minimal gain *(real)*** The gain of a node is calculated before splitting it. The node is split if its gain is greater than the *minimal gain*. Higher values of minimal gain results in fewer splits and thus a smaller tree. A too high value will completely prevent splitting and a tree with a single node is generated.

**maximal depth** ***(integer)*** The depth of a tree varies depending upon size and nature of the ExampleSet. This parameter is used to restrict the size of the Decision Tree. The tree generation process is not continued when the tree depth is equal to the *maximal depth*. If its value is set to '-1', the *maximal depth* parameter puts no bound on the depth of the tree, a tree of maximum depth is generated. If its value is set to '1', a Tree with a single node is generated.

**confidence** ***(real)*** This parameter specifies the confidence level used for the pessimistic error calculation of pruning.

**number of prepruning alternatives** ***(integer)*** As prepruning runs parallel to the tree generation process, it may prevent splitting at certain nodes when splitting at that node does not add to the discriminative power of the entire tree. In such a case alternative nodes are tried for splitting. This parameter adjusts the number of alternative nodes tried for splitting when it is prevented by prepruning at a certain node.

**no prepruning** ***(boolean)*** By default the Decision Tree is generated with prepruning. Setting this parameter to true disables the prepruning and delivers a tree without any prepruning.

**no pruning** ***(boolean)*** By default the Decision Tree is generated with pruning. Setting this parameter to true disables the pruning and delivers an unpruned Tree.

## Related Documents

- (page **??**)
- **Decision Tree (Weight-Based)** (page 433)

## Tutorial Processes

### Introduction to the CHAID operator

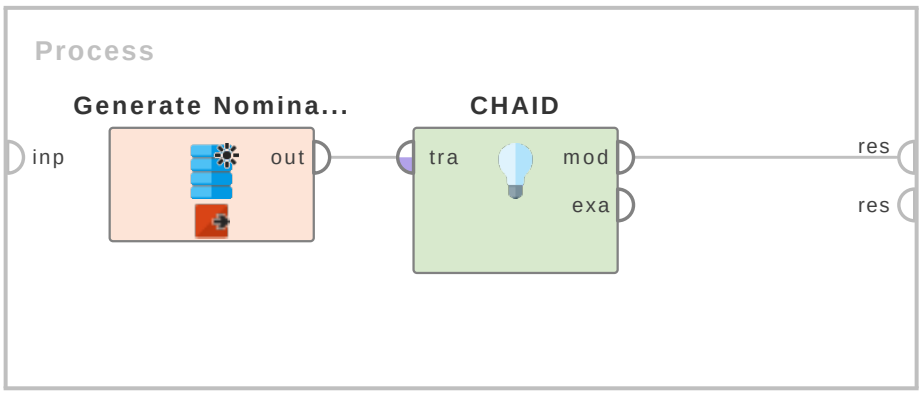
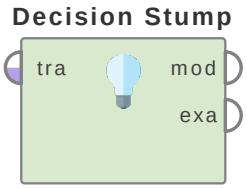



Figure 4.9: Tutorial process 'Introduction to the CHAID operator'.

The Generate Nominal Data operator is used for generating an ExampleSet with 100 examples. There are three nominal attributes in the ExampleSet and every attribute has three possible values. A breakpoint is inserted here so that you can have a look at the ExampleSet. The

CHAID operator is applied on this ExampleSet with default values of all parameters. The resultant model is connected to the result port of the process and it can be seen in the Results Workspace.

# Decision Stump




This operator learns a Decision Tree with only one single split. This operator can be applied on both nominal and numerical data sets.

## Description

The Decision Stump operator is used for generating a decision tree with only one single split. The resulting tree can be used for classifying unseen examples. This operator can be very efficient when boosted with operators like the AdaBoost operator. The examples of the given ExampleSet have several attributes and every example belongs to a class (like yes or no). The leaf nodes of a decision tree contain the class name whereas a non-leaf node is a decision node. The decision node is an attribute test with each branch (to another decision tree) being a possible value of the attribute. For more information about decision trees, please study the Decision Tree operator.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The Decision Tree with just a single split is delivered from this output port. This classification model can now be applied on unseen data sets for the prediction of the *label* attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**criterion *(selection)*** This parameter specifies the criterion on which attributes will be selected for splitting. It can have one of the following values:

- **information_gain** The entropy of all the attributes is calculated. The attribute with minimum entropy is selected for split. This method has a bias towards selecting attributes with a large number of values.
- **gain_ratio** It is a variant of information gain. It adjusts the information gain for each attribute to allow the breadth and uniformity of the attribute values.
- **gini_index** This is a measure of impurity of an ExampleSet. Splitting on a chosen attribute gives a reduction in the average gini index of the resulting subsets.
- **accuracy** Such an attribute is selected for split that maximizes the accuracy of the whole Tree.

**minimal leaf size *(integer)*** The size of a leaf node is the number of examples in its subset. The tree is generated in such a way that every leaf node subset has at least the *minimal leaf size* number of instances.

## Tutorial Processes

### Introduction to the Decision Stump operator

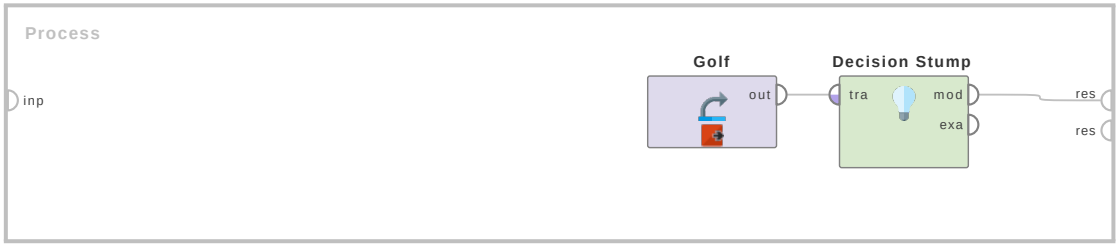
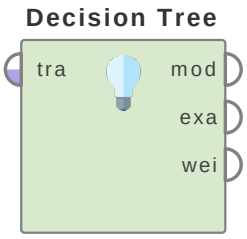



Figure 4.10: Tutorial process 'Introduction to the Decision Stump operator'.

To understand the basic terminology of trees, please study the Example Process of the Decision Tree operator.

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. The Decision Stump operator is applied on this ExampleSet. The criterion parameter is set to 'information gain' and the minimal leaf size parameter is set to 1. The resultant decision tree model is connected to the result port of the process and it can be seen in the Results Workspace. You can see that this decision tree has just a single split.

# Decision Tree




This Operator generates a decision tree model, which can be used for classification and regression.

## Description

A decision tree is a tree like collection of nodes intended to create a decision on values affiliation to a class or an estimate of a numerical target value. Each node represents a splitting rule for one specific Attribute. For classification this rule separates values belonging to different classes, for regression it separates them in order to reduce the error in an optimal way for the selected parameter *criterion*.

The building of new nodes is repeated until the stopping criteria are met. A prediction for the class label Attribute is determined depending on the majority of Examples which reached this leaf during generation, while an estimation for a numerical value is obtained by averaging the values in a leaf.

This Operator can process ExampleSets containing both nominal and numerical Attributes. The label Attribute must be nominal for classification and numerical for regression.

After generation, the decision tree model can be applied to new Examples using the Apply Model Operator. Each Example follows the branches of the tree in accordance to the splitting rule until a leaf is reached.

To configure the decision tree, please read the documentation on parameters as explained below.

## Differentiation

- **CHAID**

  The CHAID Operator provides a pruned decision tree that uses chi-squared based criterion instead of information gain or gain ratio criteria. This Operator cannot be applied on ExampleSets with numerical Attributes but only nominal Attributes.

  See page 421 for details.

- **ID3**

  The ID3 Operator provides a basic implementation of unpruned decision tree. It only works with ExampleSets with nominal Attributes.

  See page 442 for details.

- **Random Forest**

  The Random Forest Operator creates several random trees on different Example subsets. The resulting model is based on voting of all these trees. Due to this difference, it is less prone to overtraining.

  See page 444 for details.

- **Bagging**

  Bootstrap aggregating (bagging) is a machine learning ensemble meta-algorithm to improve classification and regression models in terms of stability and classification accuracy. It also reduces variance and helps to avoid 'overfitting'. Although it is usually applied to decision tree models, it can be used with any type of model.

  See page 537 for details.

## Input Ports

**training set *(tra)*** The input data which is used to generate the decision tree model.

## Output Ports

**model *(mod)*** The decision tree model is delivered from this output port.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port.

**weights *(wei)*** An ExampleSet containing Attributes and weight values, where each weight represents the feature importance for the given Attribute. A weight is given by the sum of improvements the selection of a given Attribute provided at a node. The amount of improvement is dependent on the chosen *criterion*.

## Parameters

**criterion** Selects the criterion on which Attributes will be selected for splitting. For each of these criteria the split value is optimized with regards to the chosen criterion. It can have one of the following values:

- **information_gain** The entropies of all the Attributes are calculated and the one with least entropy is selected for split. This method has a bias towards selecting Attributes with a large number of values.
- **gain_ratio** A variant of information gain that adjusts the information gain for each Attribute to allow the breadth and uniformity of the Attribute values.
- **gini_index** A measure of inequality between the distributions of label characteristics. Splitting on a chosen Attribute results in a reduction in the average gini index of the resulting subsets.
- **accuracy** An Attribute is selected for splitting, which maximizes the accuracy of the whole tree.
- **least_square** An Attribute is selected for splitting, that minimizes the squared distance between the average of values in the node with regards to the true value.

**maximal depth** The depth of a tree varies depending upon the size and characteristics of the ExampleSet. This parameter is used to restrict the depth of the decision tree. If its value is set to '-1', the *maximal depth* parameter puts no bound on the depth of the tree. In this case the tree is built until other stopping criteria are met. If its value is set to '1', a tree with a single node is generated.

**apply pruning** The decision tree model can be pruned after generation. If checked, some branches are replaced by leaves according to the *confidence* parameter.

**confidence** This parameter specifies the confidence level used for the pessimistic error calculation of pruning.

**apply prepruning** This parameter specifies if more stopping criteria than the *maximal depth* should be used during generation of the decision tree model. If checked, the parameters *minimal gain*, *minimal leaf size*, *minimal size for split* and *number of prepruning alternatives* are used as stopping criteria.

**minimal gain** The gain of a node is calculated before splitting it. The node is split if its gain is greater than the minimal gain. A higher value of *minimal gain* results in fewer splits and thus a smaller tree. A value that is too high will completely prevent splitting and a tree with a single node is generated.

**minimal leaf size** The size of a leaf is the number of Examples in its subset. The tree is generated in such a way that every leaf has at least the *minimal leaf size* number of Examples.

**minimal size for split** The size of a node is the number of Examples in its subset. Only those nodes are split whose size is greater than or equal to the *minimal size for split* parameter.

**number of prepruning alternatives** When split is prevented by prepruning at a certain node this parameter will adjust the number of alternative nodes tested for splitting. Occurs as prepruning runs parallel to the tree generation process. This may prevent splitting at certain nodes, when splitting at that node does not add to the discriminative power of the entire tree. In such a case, alternative nodes are tried for splitting.

## Tutorial Processes

### Train a Decision Tree model

Goal: RapidMiner Studio comes with a sample dataset called 'Golf'. This contains Attributes regarding the weather namely 'Outlook', 'Temperature', 'Humidity' and 'Wind'. These are important features to decide whether the game could be played or not. Our goal is to train a decision tree for predicting the 'Play' Attribute.

The 'Golf' dataset is retrieved using the Retrieve Operator. This data is fed to the Decision Tree Operator by connecting the output port of Retrieve to the input port of the Decision Tree Operator. Click on the Run button. This trains the decision tree model and takes you to the Results View, where you can examine it graphically as well as in textual description.

The tree shows that whenever the Attribute 'Outlook' has the value 'overcast', the Attribute 'Play' will have the value 'yes'. If the Attribute 'Outlook' has the value 'rain', then two outcomes are possible:

a) if the Attribute 'Wind' has the value 'false', the 'Play' Attribute has the value 'yes'

b) if the 'Wind' Attribute has the value 'true', the Attribute 'Play' is 'no'.

Finally, if the Attribute 'Outlook' has the value 'sunny', there are again two possibilities.

The Attribute 'Play' is 'yes' if the value of Attribute 'Humidity' is less than or equal to 77.5 and it is 'no' if 'Humidity' is greater than 77.5.

In this example, the leaf node led only to either of the two possible values for the label Attribute. The 'Play' Attribute is either 'yes' or 'no', which shows that the tree model fits the data very well.

### Train a Decision Tree model and apply it to predict the outcome

Goal: In this tutorial a predictive analytics process using a decision tree is shown. It is slightly advanced than the first tutorial. It also introduces basic but important concepts such as splitting

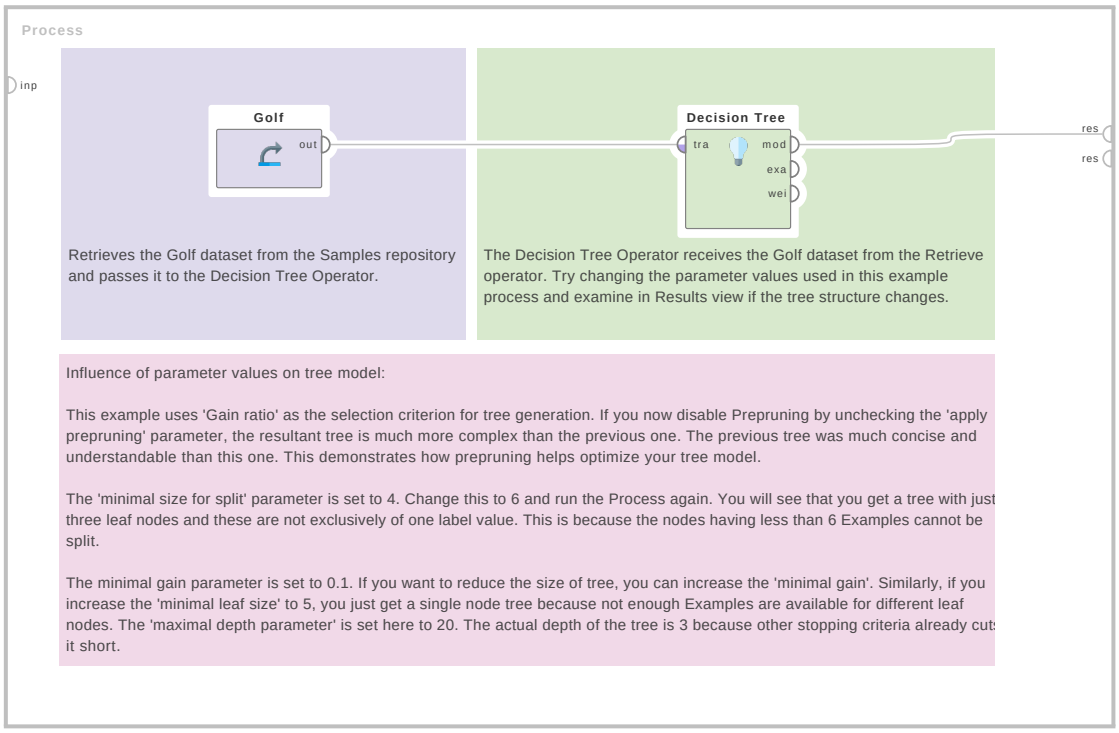



Figure 4.11: Tutorial process ‘Train a Decision Tree model’.

the dataset into two partitions. The larger half is used for training the decision tree model and the smaller half is used for testing it. Our goal is to see how good the tree model would be able to predict the fate of passengers in the test data set.

### Regression

ln this tutorial process a Decision Tree is used for regression. The ‘Polynominal’ data set with a numerical target Attribute is used as a label. Before training the model the data set is split into a training and a test set. Afterwards the regressed values are compared with the label values to obtain a performance measure using the Performance (Regression) Operator.

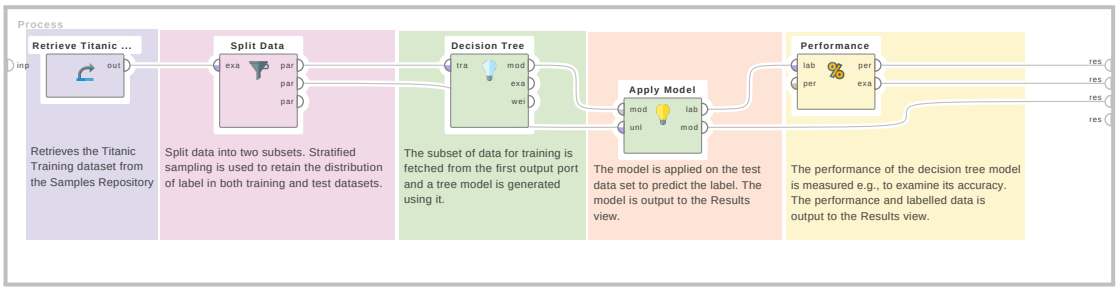
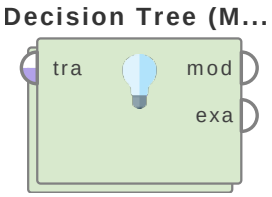



Figure 4.12: Tutorial process 'Train a Decision Tree model and apply it to predict the outcome'.

# Decision Tree (Multiway)




This operator generates a multiway decision tree.

## Description

The Decision Tree (Multiway) operator is a nested operator i.e. it has a subprocess. The subprocess must have a Tree learner i.e. an operator that expects an ExampleSet and generates a Tree model. You need to have basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

If we have only categorical attributes, we can use any C4.5-like algorithm in order to obtain a multi-way decision tree, although we will usually obtain a binary tree if our dataset includes continuous attributes. Using binary splits on numerical attributes implies that the attributes involved should be able to appear several times in the paths from the root of the tree to its leaves. Although these repetitions can be simplfied when converting the decision tree into a set of rules, they make the constructed tree more leafy, unnecessarily deeper, and harder to understand for human experts. The non-binary splits on continuous attributes make the trees easier to understand and also seem to lead to more accurate trees in some domains.

The representation of the data as Tree has the advantage compared with other approaches of being meaningful and easy to interpret. The goal is to create a classification model that predicts the value of the label based on several input attributes of the ExampleSet. Each interior node of tree corresponds to one of the input attributes. The number of edges of an interior node is equal to the number of possible values of the corresponding input attribute. Each leaf node represents a value of the label given the values of the input attributes represented by the path from the root to the leaf. This description can be easily understood by studying the Example Process of the Decision Tree operator.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as

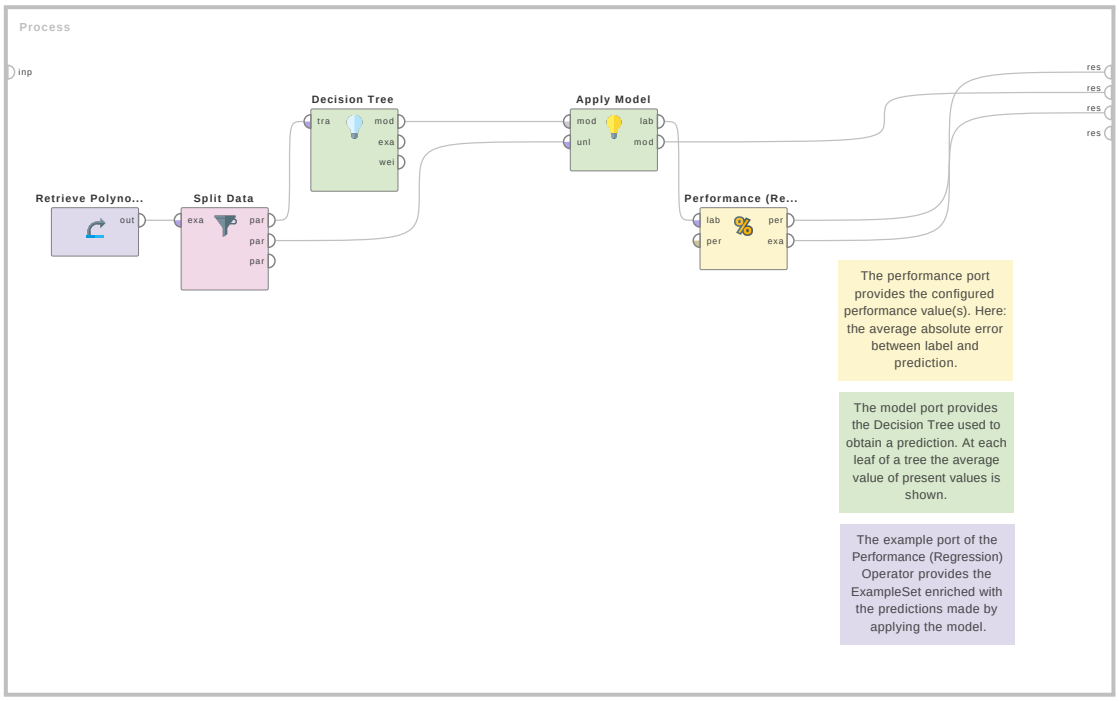



Figure 4.13: Tutorial process 'Regression'.

input.

## Output Ports

**model *(mod)*** The Decision Tree is delivered from this output port. This classification model can now be applied on unseen data sets for the prediction of the *label* attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Tutorial Processes

### Introduction to the Decision Tree (Multiway) operator

The Golf data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. The Decision Tree (Multiway) operator is applied on this ExampleSet. The Decision Tree operator is applied in the subprocess of the Decision Tree (Multiway) operator. The resultant Tree is connected to the result port of the process and it can be seen in the Results Workspace.

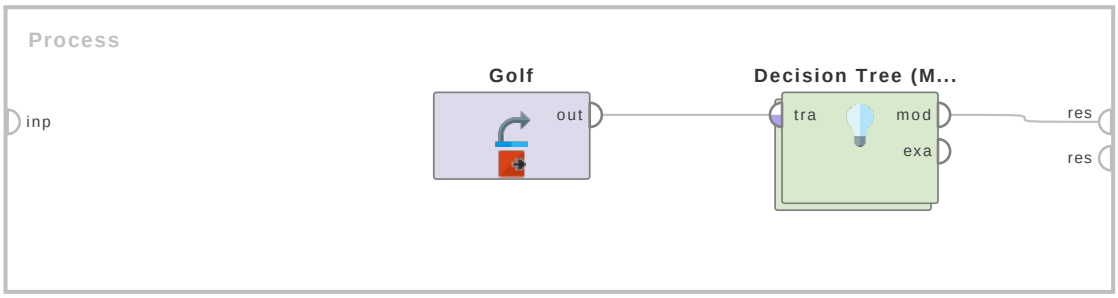
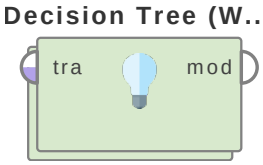



Figure 4.14: Tutorial process 'Introduction to the Decision Tree (Multiway) operator'.

# Decision Tree (Weight-Based)




This operator generates a pruned decision tree based on an arbitrary attribute relevance test. The attribute weighting scheme should be provided as inner operator. This operator can be applied only on ExampleSets with nominal data.

## Description

The Decision Tree (Weight-Based) operator is a nested operator i.e. it has a subprocess. The subprocess must have an attribute weighting scheme i.e. an operator that expects an ExampleSet and generates attribute weights. You need to have basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

The Decision Tree (Weight-Based) operator works exactly like the Decision Tree operator with one exception: it uses an arbitrary attribute relevance test criterion instead of the information gain or gain ratio criteria. Moreover this operator cannot be applied on ExampleSets with numerical attributes. It is recommended that you study the documentation of the Decision Tree operator for basic understanding of decision trees.

If the Weight by Chi Squared Statistic operator is supplied for attribute weighting, this operator acts as the CHAID operator. CHAID stands for CHi-squared Automatic Interaction Detection. The chi-square statistic is a nonparametric statistical technique used to determine if a distribution of observed frequencies differs from the theoretical expected frequencies. Chi-square statistics use nominal data, thus instead of using means and variances, this test uses frequencies. CHAID's advantages are that its output is highly visual and easy to interpret. Because it uses multiway splits by default, it needs rather large sample sizes to work effectively, since with small sample sizes the respondent groups can quickly become too small for reliable analysis.

The representation of the data as Tree has the advantage compared with other approaches of being meaningful and easy to interpret. The goal is to create a classification model that predicts the value of the label based on several input attributes of the ExampleSet. Each interior node of the tree corresponds to one of the input attributes. The number of edges of an interior node is equal to the number of possible values of the corresponding input attribute. Each leaf node represents a value of the label given the values of the input attributes represented by the path from the root to the leaf. This description can be easily understood by studying the Example Process of the Decision Tree operator.

Pruning is a technique in which leaf nodes that do not add to the discriminative power of the decision tree are removed. This is done to convert an over-specific or over-fitted tree to a more general form in order to enhance its predictive power on unseen datasets. Pre-pruning is a type of pruning performed parallel to the tree creation process. Post-pruning, on the other hand, is done after the tree creation process is complete.

## Differentiation

- **CHAID** If the Weight by Chi Squared Statistic operator is applied for attribute weighting in the subprocess of the Decision Tree (Weight-Based) operator, it works exactly like the CHAID operator. See page 421 for details.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Generate Nominal Data operator in the attached Example Process. The output of other operators can also be used as input. This operator cannot handle numerical data, therefore the ExampleSet should not have numerical attributes.

## Output Ports

**model *(mod)*** The Decision Tree is delivered from this output port. This classification model can now be applied on unseen data sets for the prediction of the *label* attribute.

## Parameters

**minimal size for split *(integer)*** The size of a node in a Tree is the number of examples in its subset. The size of the root node is equal to the total number of examples in the ExampleSet. Only those nodes are split whose size is greater than or equal to the *minimal size for split* parameter.

**minimal leaf size *(integer)*** The size of a leaf node in a Tree is the number of examples in its subset. The tree is generated in such a way that every leaf node subset has at least the *minimal leaf size* number of instances.

**maximal depth *(integer)*** The depth of a tree varies depending upon size and nature of the ExampleSet. This parameter is used to restrict the size of the Decision Tree. The tree generation process is not continued when the tree depth is equal to the *maximal depth*. If its value is set to '-1', the *maximal depth* parameter puts no bound on the depth of the tree, a tree of maximum depth is generated. If its value is set to '1', a Tree with a single node is generated.

**confidence *(real)*** This parameter specifies the confidence level used for the pessimistic error calculation of pruning.

**no pruning *(boolean)*** By default the Decision Tree is generated with pruning. Setting this parameter to true disables the pruning and delivers an unpruned Tree.

**number of prepruning alternatives *(integer)*** As prepruning runs parallel to the tree generation process, it may prevent splitting at certain nodes when splitting at that node does not add to the discriminative power of the entire tree. In such a case alternative nodes are tried for splitting. This parameter adjusts the number of alternative nodes tried for splitting when the split is prevented by prepruning at a certain node.

## Related Documents

- **CHAID** (page 421)

## Tutorial Processes

### Introduction to the Decision Tree (Weight-Based) operator

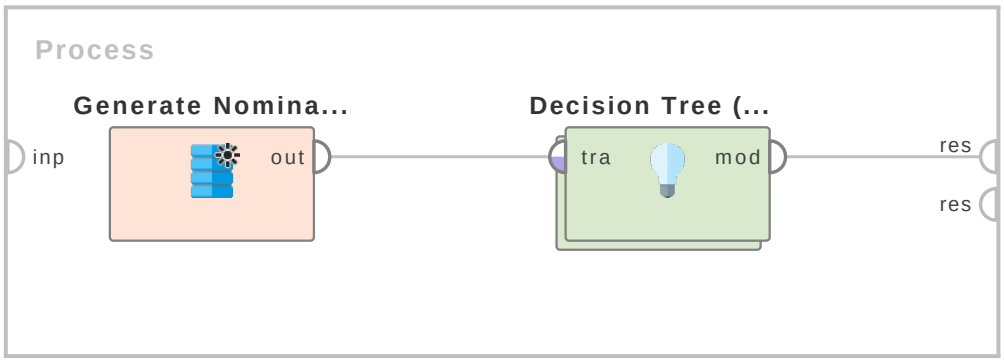
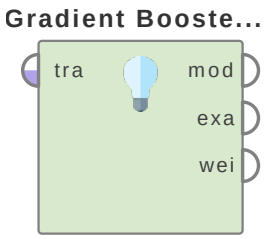



Figure 4.15: Tutorial process 'Introduction to the Decision Tree (Weight-Based) operator'.

The Generate Nominal Data operator is used for generating an ExampleSet with 100 examples. There are three nominal attributes in the ExampleSet and every attribute has three possible values. A breakpoint is inserted here so that you can have a look at the ExampleSet. The Decision Tree (Weight-Based) operator is applied on this ExampleSet with default values of all parameters. The resultant model is connected to the result port of the process and it can be seen in the Results Workspace.

# Gradient Boosted Trees




Executes GBT algorithm using H2O 3.8.2.6.

## Description

*Please note that the result of this algorithm may depend on the number of threads used. Different settings may lead to slightly different outputs.*

A gradient boosted model is an ensemble of either regression or classification tree models. Both are forward-learning ensemble methods that obtain predictive results through gradually improved estimations. Boosting is a flexible nonlinear regression procedure that helps improving the accuracy of trees. By sequentially applying weak classification algorithms to the incrementally changed data, a series of decision trees are created that produce an ensemble of weak prediction models. While boosting trees increases their accuracy, it also decreases speed and human interpretability. The gradient boosting method generalizes tree boosting to minimize these issues.

The operator starts a 1-node local H2O cluster and runs the algorithm on it. Although it uses one node, the execution is parallel. You can set the level of parallelism by changing the Settings/Preferences/General/Number of threads setting. By default it uses the recommended number of threads for the system. Only one instance of the cluster is started and it remains running until you close RapidMiner Studio.

## Input Ports

**training set *(tra)*** The input port expects a labeled ExampleSet.

## Output Ports

**model *(mod)*** The Gradient Boosted classification or regression model is delivered from this output port. This classification or regression model can be applied on unseen data sets for prediction of the label attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute.

## Parameters

**number of trees *(integer)*** A non-negative integer that defines the number of trees. The default is 20.

**reproducible** ***(boolean)*** Makes model building reproducible. If set then maximum_number_of_threads parameter controls parallelism level of model building. If this is not set then parallelism level is defined by number of threads in General Preferences.

**maximum number of threads** ***(integer)*** Controls parallelism level of model building.

**use local random seed** ***(boolean)*** Available only if reproducible is set to true. Indicates if a local random seed should be used for randomization.

**local random seed** ***(integer)*** This parameter specifies the local random seed. This parameter is only available if the use local random seed parameter is set to true.

**maximal depth** ***(integer)*** The user-defined tree depth. The default is 5.

**min rows** ***(real)*** The minimum number of rows to assign to the terminal nodes. The default is 10.0. If a weight column is specified, the number of rows are also weighted. E.g. if a terminal node contains two rows with the weights 0.3 and 0.4, it is counted as 0.7 in the minimum number of rows.

**min split improvement** ***(real)*** Minimum relative improvement in squared error reduction for a split to happen.

**number of bins** ***(integer)*** For numerical columns (real/integer), build a histogram of at least the specified number of bins, then split at the best point The default is 20.

**learning rate** ***(real)*** The learning rate. Smaller learning rates lead to better models, however, it comes at the price of increasing computational time both during training and scoring: lower learning rate requires more iterations. The default is 0.1 and the range is 0.0 to 1.0.

**sample rate** ***(real)*** Row sample rate per tree (from 0.0 to 1.0).

**distribution** ***(selection)*** The distribution function for the training data. For some function (e.g. tweedie) further tuning can be achieved via the expert parameters

- **AUTO** Automatic selection. Uses multinomial for nominal and gaussian for numeric labels.
- **bernoulli** Bernoulli distribution. Can be used for binominal or 2-class polynominal labels.
- **gaussian, possion, gamma, tweedie, quantile** Distribution functions for regression.

**early stopping** ***(boolean)*** If true, parameters for early stopping needs to be specified.

**stopping rounds** ***(integer)*** Early stopping based on convergence of stopping_metric. Stop if simple moving average of length k of the stopping_metric does not improve for k:=stopping_rounds scoring events. This parameter is visible only if early_stopping is set.

**stopping metric** ***(selection)*** Metric to use for early stopping. Set stopping_tolerance to tune it. This parameter is visible only if early_stopping is set.

- **AUTO** Automatic selection. Uses logloss for classification, deviance for regression.
- **deviance, logloss, MSE, AUC, lift_top_group, r2, misclassification** The metric to use to decide if the algorithm should be stopped.

**stopping tolerance** ***(real)*** Relative tolerance for metric-based stopping criterion (stop if relative improvement is not at least this much). This parameter is visible only if early_stopping is set.

**max runtime seconds** ***(integer)*** Maximum allowed runtime in seconds for model training. Use 0 to disable.

**expert parameters** ***(enumeration)*** These parameters are for fine tuning the algorithm. Usually the default values provide a decent model, but in some cases it may be useful to change them. Please use true/false values for boolean parameters and the exact attribute name for columns. Arrays can be provided by splitting the values with the comma (,) character. More information on the parameters can be found in the H2O documentation.

- **score_each_iteration** Whether to score during each iteration of model training. Type: boolean, Default: false
- **score_tree_interval** Score the model after every so many trees. Disabled if set to 0. Type: integer, Default: 0
- **fold_assignment** Cross-validation fold assignment scheme, if fold_column is not specified. Options: AUTO, Random, Modulo, Stratified. Type: enumeration, Default: AUTO
- **fold_column** Column name with cross-validation fold index assignment per observation. Type: column, Default: no fold column
- **offset_column** Offset column name. Type: Column, Default: no offset column
- **balance_classes** Balance training data class counts via over/under-sampling (for imbalanced data). Type: boolean, Default: false
- **max_after_balance_size** Maximum relative size of the training data after balancing class counts (can be less than 1.0). Requires balance_classes. Type: real, Default: 5.0
- **max_confusion_matrix_size** Maximum size (# classes) for confusion matrices to be printed in the Logs. Type: integer, Default: 20
- **nbins_top_level** For numerical columns (real/int), build a histogram of (at most) this many bins at the root level, then decrease by factor of two per level. Type: integer, Default: 1024
- **nbins_cats** For categorical columns (factors), build a histogram of this many bins, then split at the best point. Higher values can lead to more overfitting. Type: integer, Default: 1024
- **r2_stopping** Stop making trees when the R^2 metric equals or exceeds this. type: double, Default: 0.999999
- **quantile_alpha** Desired quantile for quantile regression (from 0.0 to 1.0). Type: double, Default: 0.5
- **tweedie_power** Tweedie Power (between 1 and 2). Type: double, Default: 1.5
- **col_sample_rate** Column sample rate (from 0.0 to 1.0). Type: double, Default: 1.0
- **col_sample_rate_per_tree** Column sample rate per tree (from 0.0 to 1.0). Type: double, Default: 1.0
- **keep_cross_validation_predictions** Keep cross-validation model predictions. Type: boolean, Default: false
- **keep_cross_validation_fold_assignment** Keep cross-validation fold assignment. Type: boolean, Default: false
- **class_sampling_factors** Desired over/under-sampling ratios per class (in lexicographic order). If not specified, sampling factors will be automatically computed to obtain class balance during training. Requires balance_classes=true. Type: float array, Default: empty

- **learn_rate_annealing** Scale down the learning rate by this factor after each tree. Type: double, Default: 1.0
- **sample_rate_per_class** Row sample rate per tree per class (from 0.0 to 1.0) Type: double arary, Default: empty
- **col_sample_rate_change_per_level** Relative change of the column sampling rate for every level (from 0.0 to 2.0). Type: double, Default: 1.0
- **max_abs_leafnode_pred** Maximum absolute value of a leaf node prediction. Type: double, Default: Infinity
- **nfolds** Number of folds for cross-validation. Use 0 to turn off cross-validation. Type: integer, Default: 0

## Tutorial Processes

### Classification using GBT

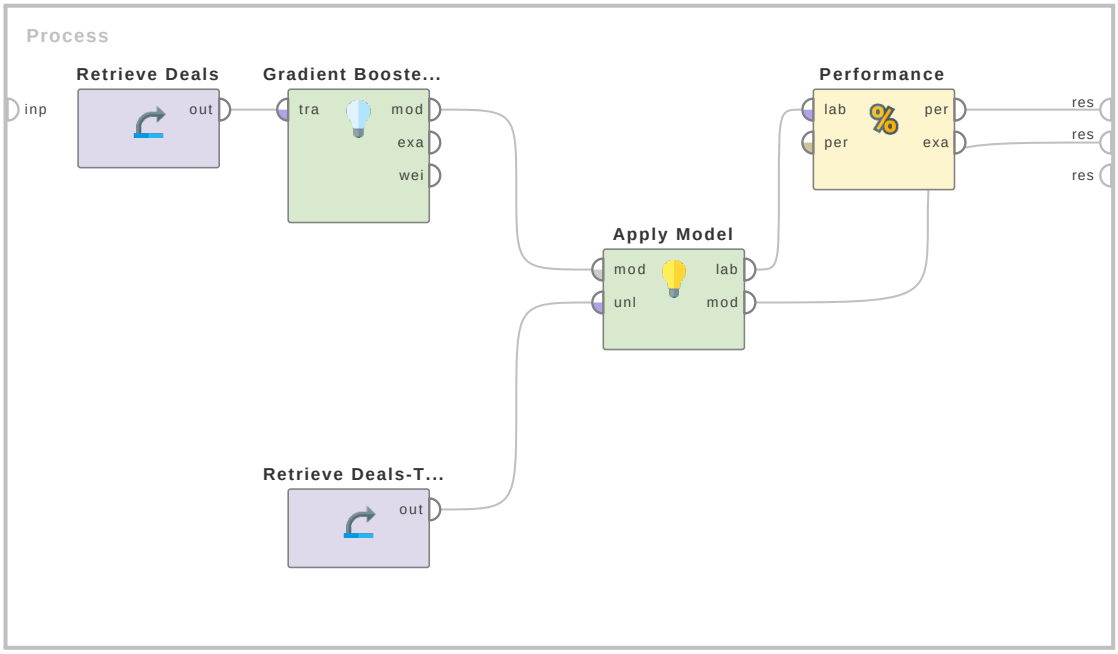



Figure 4.16: Tutorial process 'Classification using GBT'.

The H2O GBT operator is used to predict the future_customer attribute of the Deals sample dataset. Since the label is nominal, classification will be performed. The GBT parameters are slightly changed. The number of trees is decreased to 10 to lower the execution time and to prevent overfitting. The learning rate is increased to 0.3 for similar reasons. The resulting model is connected to an Apply Model operator that applies the GBT model on the Deals_Testset sample data. The labeled ExampleSet is connected to a Performance (Binominal Classification) operator, that calculates the Accuracy metric. On the process output the Performance Vector and the Gradient Boosted Model is shown. The trees of the Gradient Boosted model can be checked on the Results view.

### Classification with Split Validation using GBT

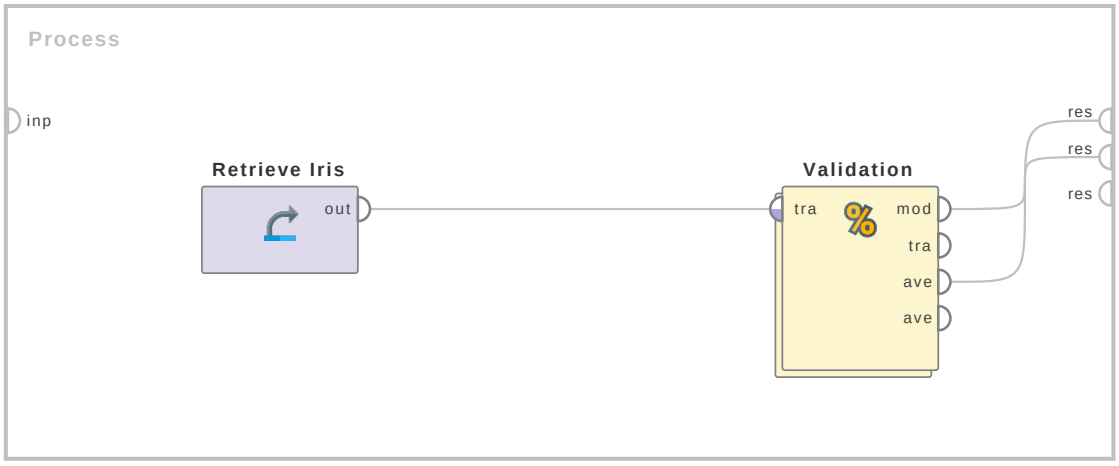



Figure 4.17: Tutorial process ‘Classification with Split Validation using GBT’.

The H2O GBT operator is used to predict the label attribute of the Iris sample dataset. Since the label is polynominal, classification will be performed. The learner operator is inside a Split Validation for being able to check the performance of the classification. The number of trees is set to 10, all other parameters are kept at the default value. The Performance (Classification) operator delivers the accuracy and the classification error. The model contains 30 trees, because H2O creates 10 trees for every unique label value.

### Regression using GBT

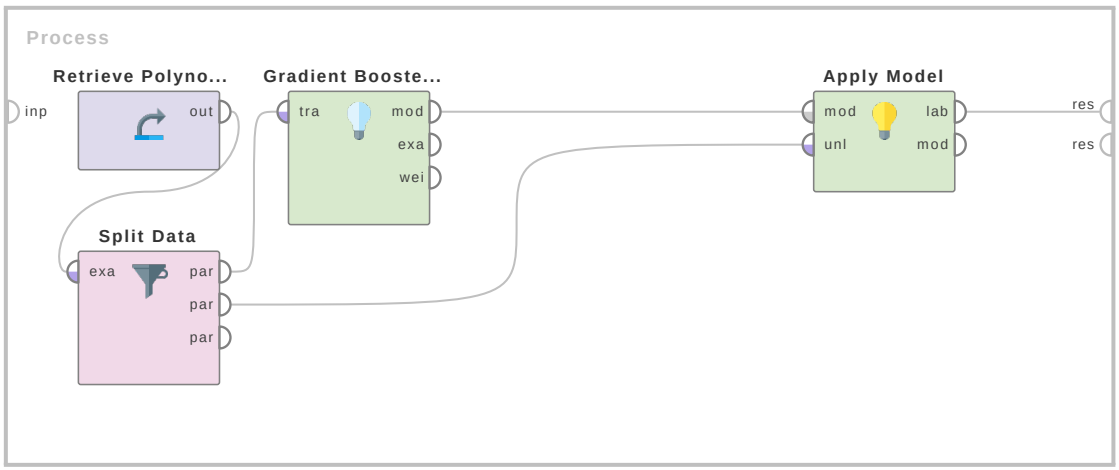
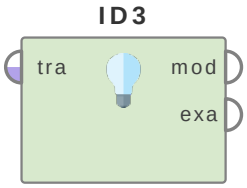



Figure 4.18: Tutorial process ‘Regression using GBT’.

The H2O GBT operator is used to predict the label attribute of the Polynomial sample dataset. Since the label is real, regression is performed. The sample data is retrieved, then splitted into

two parts with the Split Data operator. The first output is used as the training, the second as the scoring data set. The GBT operator's distribution parameter is changed to "gamma". After applying on the scoring ExampleSet, the output contains the GradientBoostedModel and the labeled data. If you select Charts/Series Chart style for the labeled data and choose label and prediction label in the Plot Series field, you can check the accuracy of the prediction visually.

# ID3




This operator learns an unpruned Decision Tree from nominal data for classification. This decision tree learner works similar to Quinlan's ID3.

## Description

ID3 (Iterative Dichotomiser 3) is an algorithm used to generate a decision tree invented by Ross Quinlan. ID3 is the precursor to the C4.5 algorithm. Very simply, ID3 builds a decision tree from a fixed set of examples. The resulting tree is used to classify future samples. The examples of the given ExampleSet have several attributes and every example belongs to a class (like yes or no). The leaf nodes of the decision tree contain the class name whereas a non-leaf node is a decision node. The decision node is an attribute test with each branch (to another decision tree) being a possible value of the attribute. ID3 uses feature selection heuristic to help it decide which attribute goes into a decision node. The required heuristic can be selected by the *criterion* parameter.

The ID3 algorithm can be summarized as follows:

1. Take all unused attributes and calculate their selection criterion (e.g. information gain)
2. Choose the attribute for which the selection criterion has the best value (e.g. minimum entropy or maximum information gain)
3. Make node containing that attribute

ID3 searches through the attributes of the training instances and extracts the attribute that best separates the given examples. If the attribute perfectly classifies the training sets then ID3 stops; otherwise it recursively operates on the n (where n = number of possible values of an attribute) partitioned subsets to get their best attribute. The algorithm uses a greedy search, meaning it picks the best attribute and never looks back to reconsider earlier choices.

Some major benefits of ID3 are:

- Understandable prediction rules are created from the training data.
- Builds a short tree in relatively small time.
- It only needs to test enough attributes until all data is classified.
- Finding leaf nodes enables test data to be pruned, reducing the number of tests.

ID3 may have some disadvantages in some cases e.g.

- Data may be over-fitted or over-classified, if a small sample is tested.
- Only one attribute at a time is tested for making a decision.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Generate Nominal Data operator in the attached Example Process. This operator cannot handle numerical attributes. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The Decision Tree is delivered from this output port. This classification model can now be applied on unseen data sets for the prediction of the *label* attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**criterion *(selection)*** This parameter specifies the criterion on which attributes will be selected for splitting. It can have one of the following values:

- **information_gain** The entropy of all the attributes is calculated. The attribute with minimum entropy is selected for split. This method has a bias towards selecting attributes with a large number of values.
- **gain_ratio** It is a variant of information gain. It adjusts the information gain for each attribute to allow the breadth and uniformity of the attribute values.
- **gini_index** This is a measure of impurity of an ExampleSet. Splitting on a chosen attribute gives a reduction in the average gini index of the resulting subsets.
- **accuracy** Such an attribute is selected for a split that maximizes the accuracy of the whole Tree.

**minimal size for split *(integer)*** The size of a node is the number of examples in its subset. The size of the root node is equal to the total number of examples in the ExampleSet. Only those nodes are split whose size is greater than or equal to the *minimal size for split* parameter.

**minimal leaf size *(integer)*** The size of a leaf node is the number of examples in its subset. The tree is generated in such a way that every leaf node subset has at least the *minimal leaf size* number of instances.

**minimal gain *(real)*** The gain of a node is calculated before splitting it. The node is split if its Gain is greater than the *minimal gain*. Higher value of minimal gain results in fewer splits and thus a smaller tree. A too high value will completely prevent splitting and a tree with a single node is generated.

## Tutorial Processes

### Getting started with ID3

To understand the basic terminology of trees, please study the Example Process of the Decision Tree operator.

The Generate Nominal Data operator is used for generating an ExampleSet with nominal attributes. It should be kept in mind that the ID3 operator cannot handle numerical attributes. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has three attributes and each attribute has three possible values. The ID3 operator is applied on this ExampleSet with default values of all parameters. The resultant Decision Tree model is delivered to the result port of the process and it can be seen in the Results Workspace.

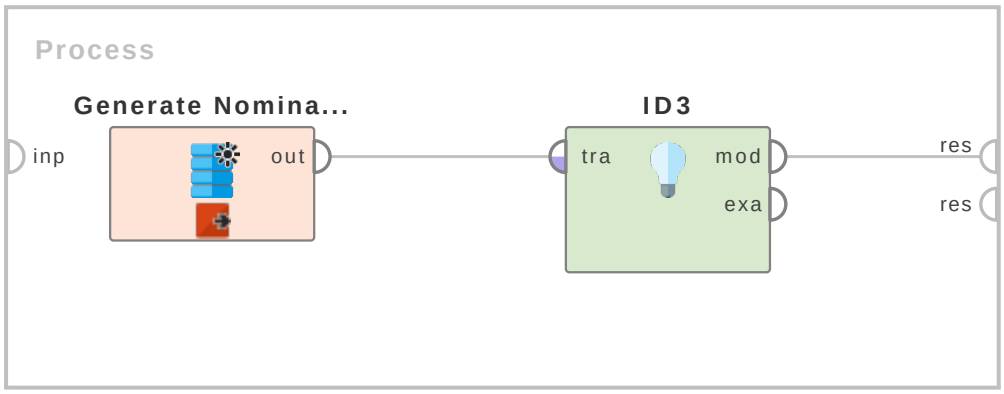
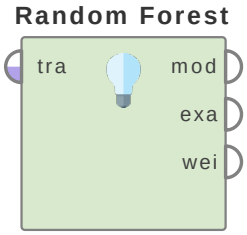



Figure 4.19: Tutorial process 'Getting started with ID3'.

## Random Forest




This Operator generates a random forest model, which can be used for classification and regression.

### Description

A random forest is an ensemble of a certain number of random trees, specified by the *number of trees* parameter. These trees are created/trained on bootstrapped sub-sets of the ExampleSet provided at the Input Port. Each node of a tree represents a splitting rule for one specific Attribute. Only a sub-set of Attributes, specified with the *subset ratio* criterion, is considered for the splitting rule selection. This rule separates values in an optimal way for the selected parameter *criterion*. For classification the rule is separating values belonging to different classes, while for regression it separates them in order to reduce the error made by the estimation. The building of new nodes is repeated until the stopping criteria are met.

After generation, the random forest model can be applied to new Examples using the Apply Model Operator. Each random tree generates a prediction for each Example by following the branches of the tree in accordance to the splitting rules and evaluating the leaf. Class predictions are based on the majority of Examples, while estimations are obtained through the average of values reaching a leaf. The resulting model is a voting model of all created random trees. Since all single predictions are considered equally important, and are based on sub-sets of Examples the resulting prediction tends to vary less than the single predictions.

A concept called pruning can be leveraged to reduce complexity of the model by replacing subtrees, that only provide little predictive power with leaves. For different types of pruning refer to the parameter descriptions.

Extremely randomized trees are a method similar to random forest, which can be obtained by checking the *split random* parameter and disabling *pruning*. Important parameters to tune for this method are the *minimal leaf size* and *split ratio*, which can be changed after disabling *guess split ratio*. Good default choices for the *minimal leaf size* are 2 for classification and 5 for

regression problems.

## Differentiation

- **Decision Tree**

  The Decision Tree Operator creates one tree, where all Attributes are available at each node for selecting the optimal one with regards to the chosen *criterion*. Since only one tree is generated the prediction is more comprehensible for humans, but might lead to overtraining.

  See page 427 for details.

- **Bagging**

  Bootstrap aggregating (bagging) is a machine learning ensemble meta-algorithm to improve classification and regression models in terms of stability and classification accuracy. It also reduces variance and helps to avoid 'overfitting'. Although it is usually applied to decision tree models, it can be used with any type of model. The random forest uses bagging with random trees.

  See page 537 for details.

- **Gradient Boosted Trees**

  The Gradient Boosted Trees Operator trains a model by iteratively improving a single tree model. After each iteration step the Examples are reweighted based on their previous prediction. The final model is a weighted sum of all created models. Training parameters are optimized based on the gradient of the function described by the errors made.

  See page 436 for details.

## Input Ports

**training set *(tra)*** The input data which is used to generate the random forest model.

## Output Ports

**model *(mod)*** The random forest model is delivered from this output port.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port.

**weights *(wei)*** An ExampleSet containing Attributes and weight values, where each weight represents the feature importance for the given Attribute. A weight is given by the sum of improvements the selection of a given Attribute provided at a node. The amount of improvement is dependent on the chosen *criterion*.

## Parameters

**number of trees** This parameter specifies the number of random trees to generate. For each tree a sub-set of Examples is selected via bootstrapping. If the parameter *enable parallel execution* is checked, the trees are trained in parallel across available processor threads.

**criterion** Selects the criterion on which Attributes will be selected for splitting. For each of these criteria the split value is optimized with regards to the chosen criterion. It can have one of the following values:

- **information_gain** The entropies of all the Attributes are calculated and the one with least entropy is selected for split. This method has a bias towards selecting Attributes with a large number of values.
- **gain_ratio** A variant of information gain that adjusts the information gain for each Attribute to allow the breadth and uniformity of the Attribute values.
- **gini_index** A measure of inequality between the distributions of label characteristics. Splitting on a chosen Attribute results in a reduction in the average gini index of the resulting subsets.
- **accuracy** An Attribute is selected for splitting, which maximizes the accuracy of the whole tree.
- **least_square** An Attribute is selected for splitting, that minimizes the squared distance between the average of values in the node with regards to the true value.

**maximal depth** The depth of a tree varies depending upon the size and characteristics of the ExampleSet. This parameter is used to restrict the depth for each random tree. If its value is set to '-1', the *maximal depth* parameter puts no bound on the depth of the trees. In this case all trees are built until other stopping criteria are met. If its value is set to '1', only trees with a single node are generated.

**apply prepruning** This parameter specifies if more stopping criteria than the *maximal depth* should be used during generation of the decision trees. If checked, the parameters *minimal gain*, *minimal leaf size*, *minimal size for split* and *number of prepruning alternatives* are used as stopping criteria.

**minimal gain** The gain of a node is calculated before splitting it. The node is split if its gain is greater than the minimal gain. A higher value of *minimal gain* results in fewer splits and thus smaller trees. A value that is too high will completely prevent splitting and trees with single nodes are generated.

**minimal leaf size** The size of a leaf is the number of Examples in its subset. The trees of the random forest are generated in such a way that every leaf has at least the *minimal leaf size* number of Examples.

**minimal size for split** The size of a node is the number of Examples in its subset. Only those nodes are split whose size is greater than or equal to the *minimal size for split* parameter.

**number of prepruning alternatives** When split is prevented by prepruning at a certain node this parameter will adjust the number of alternative nodes tested for splitting. Occurs as prepruning runs parallel to the tree generation process. This may prevent splitting at certain nodes, when splitting at that node does not add to the discriminative power of the entire tree. In such a case, alternative nodes are tried for splitting.

**apply pruning** The random trees of the random forest model can be pruned after generation. If checked, some branches are replaced by leaves according to the *confidence* parameter. This parameter is not available for the 'least_square' criterion.

**confidence** This parameter specifies the confidence level used for the pessimistic error calculation of pruning.

**random splits** If checked, this parameter causes the splits of numerical Attributes to be chosen randomly instead of being optimized. For the random selection a uniform sampling

between the minimal and maximal value for the current Attribute is performed. Activating this parameter while disabling *pruning* configures the random forest to become an extremely randomized tree (also known as Extra-Tree). This also speeds up the model building process.

**guess subset ratio** If this parameter is set to true then *int(log(m) + 1)*Attributes are used, otherwise a ratio should be specified by the* subset ratio* parameter.

**subset ratio** This parameter specifies the ratio of randomly chosen Attributes to test.

**voting strategy** Specifies the prediction strategy in case of dissenting tree model predictions:

This parameter is not available for the 'least_square' criterion.

- **confidence_vote** Selects the class that has the highest accumulated confidence.
- **majority_vote** Selects the class that was predicted by the majority of tree models.

**use local random seed** This parameter indicates if a *local random seed* should be used for randomization.

**local random seed** If the *use local random seed* parameter is checked this parameter determines the local random seed.

**enable parallel execution** This parameter enables the parallel execution of the model building process by distributing the Random Tree generation between all available CPU threads. Please disable the parallel execution if you run into memory problems.

## Tutorial Processes

### Generating a set of random trees using the random forest Operator

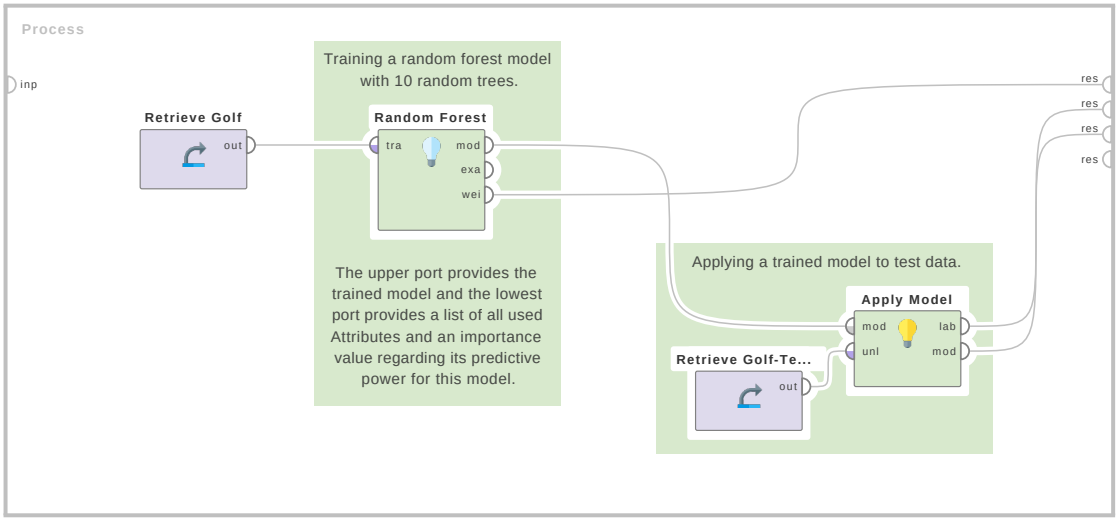



Figure 4.20: Tutorial process 'Generating a set of random trees using the random forest Operator'.

In this tutorial process the ‘Golf’ data set is retrieved and used to train a random forest for classification with 10 random trees. The generated model is afterwards applied to a test data set. Resulting predictions, the generated model and feature importance values provided by the Operators are viewed.

Checking the output of the Apply Model Operators ‘lab’ port reveals the labeled data set with predictions obtained from applying the model to an unseen data set. Inspecting the model shows a Collection of 10 random trees that build up the random forest and contribute to the predictive process. Looking at the output of the ‘wei’ port from the Random Forest Operator provides information about the Attribute weights. These weights contain importance values regarding the predictive power of an Attribute to the overall decision of the random forest.

### Random forest for regression

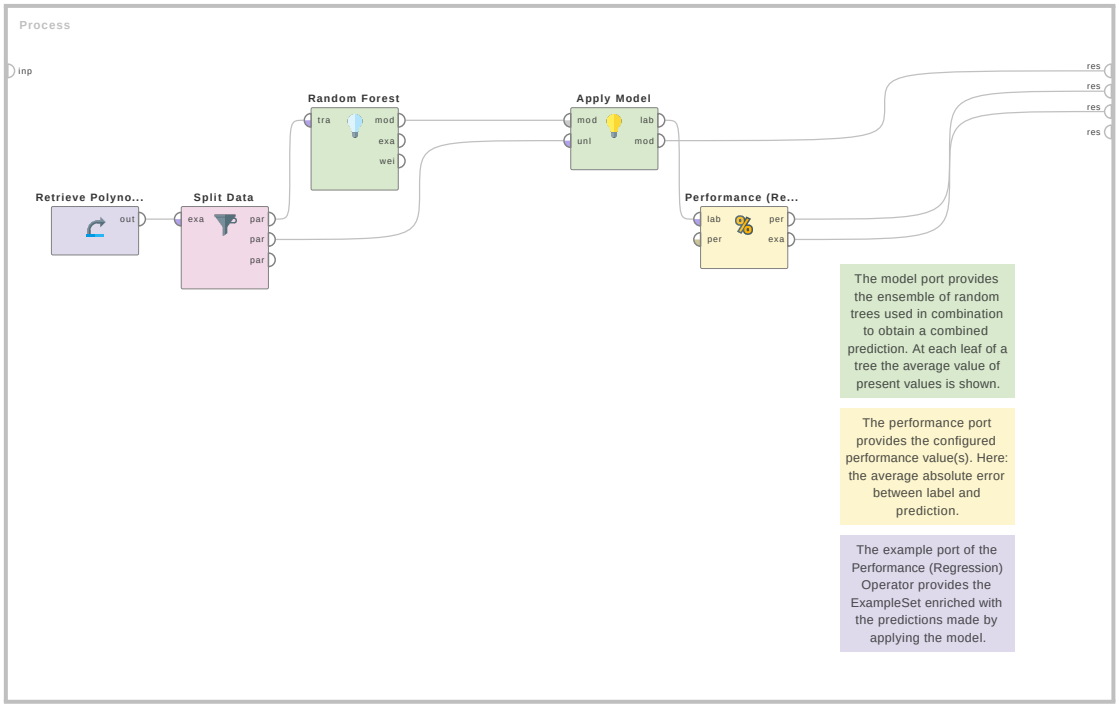



Figure 4.21: Tutorial process ‘Random forest for regression’.

ln this tutorial process a random forest is used for regression. The ‘Polynominal’ data set with a numerical target Attribute is used as a label. Before training the model the data set is split into a training and a test set. Afterwards the regressed values are compared with the label values to obtain a performance measure using the Performance (Regression) Operator.

### Comparison between decision tree and random forest

In this tutorial process a comparison highlighting the difference between decision trees and random forest is shown. The ‘Polynominal’ sample data set is split into a training and a test set. Afterwards each training data set is used to generate a decision tree and a random forest model for regression. Applying the models to the test data sets and evaluating the performance shows

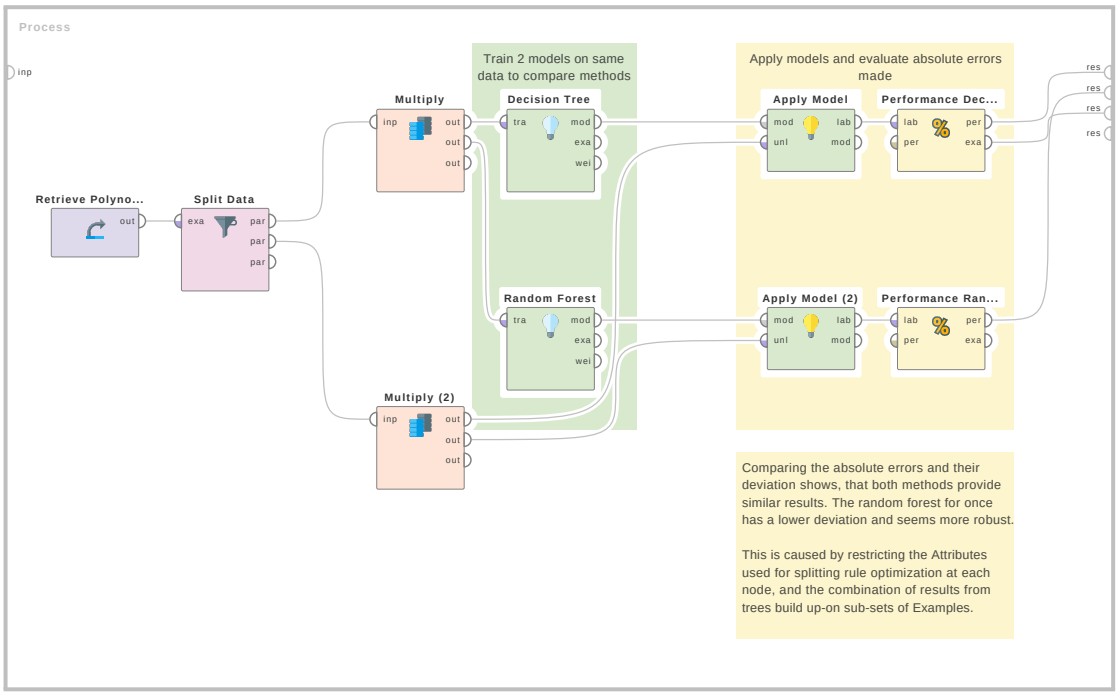
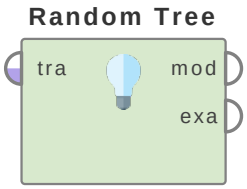



Figure 4.22: Tutorial process ‘Comparison between decision tree and random forest’.

that both methods provide similar results with a difference in deviation of the result when applied to test data.

# Random Tree




This operator learns a decision tree. This operator uses only a random subset of attributes for each split.

## Description

The Random Tree operator works exactly like the Decision Tree operator with one exception: for each split only a random subset of attributes is available. It is recommended that you study the documentation of the Decision Tree operator for basic understanding of decision trees.

This operator learns decision trees from both nominal and numerical data. Decision trees are powerful classification methods which can be easily understood. The Random Tree operator works similar to Quinlan's C4.5 or CART but it selects a random subset of attributes before it is applied. The size of the subset is specified by the *subset ratio* parameter.

Representation of the data as Tree has the advantage compared with other approaches of being meaningful and easy to interpret. The goal is to create a classification model that predicts the value of the label based on several input attributes of the ExampleSet. Each interior node of tree corresponds to one of the input attributes. The number of edges of an interior node is equal to the number of possible values of the corresponding input attribute. Each leaf node represents a value of the label given the values of the input attributes represented by the path from the root to the leaf. This description can be easily understood by studying the Example Process of the Decision Tree operator.

Pruning is a technique in which leaf nodes that do not add to the discriminative power of the decision tree are removed. This is done to convert an over-specific or over-fitted tree to a more general form in order to enhance its predictive power on unseen datasets. Pre-pruning is a type of pruning performed parallel to the tree creation process. Post-pruning, on the other hand, is done after the tree creation process is complete.

## Differentiation

- The Random Tree operator works exactly like the Decision Tree operator with one exception: for each split only a random subset of attributes is available. See page **??** for details.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The Random Tree is delivered from this output port. This classification model can now be applied on unseen data sets for the prediction of the *label* attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**criterion** ***(selection)*** This parameter selects the criterion on which attributes will be selected for splitting. It can have one of the following values:

- **information_gain** The entropy of all the attributes is calculated. The attribute with minimum entropy is selected for split. This method has a bias towards selecting attributes with a large number of values.
- **gain_ratio** It is a variant of information gain. It adjusts the information gain for each attribute to allow the breadth and uniformity of the attribute values.
- **gini_index** This is a measure of impurity of an ExampleSet. Splitting on a chosen attribute gives a reduction in the average gini index of the resulting subsets.
- **accuracy** Such an attribute is selected for split that maximizes the accuracy of the whole Tree.

**minimal size for split** ***(integer)*** The size of a node in a Tree is the number of examples in its subset. The size of the root node is equal to the total number of examples in the ExampleSet. Only those nodes are split whose size is greater than or equal to the *minimal size for split* parameter.

**minimal leaf size** ***(integer)*** The size of a leaf node in a Tree is the number of examples in its subset. The tree is generated in such a way that every leaf node subset has at least the *minimal leaf size* number of instances.

**minimal gain** ***(real)*** The gain of a node is calculated before splitting it. The node is split if its Gain is greater than the *minimal gain*. Higher value of minimal gain results in fewer splits and thus a smaller tree. A too high value will completely prevent splitting and a tree with a single node is generated.

**maximal depth** ***(integer)*** The depth of a tree varies depending upon size and nature of the ExampleSet. This parameter is used to restrict the size of the Tree. The tree generation process is not continued when the tree depth is equal to the *maximal depth*. If its value is set to '-1', the *maximal depth* parameter puts no bound on the depth of the tree, a tree of maximum depth is generated. If its value is set to '1', a Tree with a single node is generated.

**confidence** ***(real)*** This parameter specifies the confidence level used for the pessimistic error calculation of pruning.

**number of prepruning alternatives** ***(integer)*** As prepruning runs parallel to the tree generation process, it may prevent splitting at certain nodes when splitting at that node does not add to the discriminative power of the entire tree. In such a case alternative nodes are tried for splitting. This parameter adjusts the number of alternative nodes tried for splitting when split is prevented by prepruning at a certain node.

**no prepruning** ***(boolean)*** By default the Tree is generated with prepruning. Setting this parameter to true disables the prepruning and delivers a tree without any prepruning.

**no pruning** ***(boolean)*** By default the Tree is generated with pruning. Setting this parameter to true disables the pruning and delivers an unpruned Tree.

**guess subset ratio** ***(boolean)*** This parameter specifies if the subset ratio should be guessed or not. If set to true, *log(m) + 1* features are used as subset, otherwise a ratio has to be specified through the *subset ratio* parameter.

**subset ratio** ***(real)*** This parameter specifies the subset ratio of randomly chosen attributes.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of the *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Related Documents

- (page **??**)

## Tutorial Processes

### Introduction to the Random Tree operator

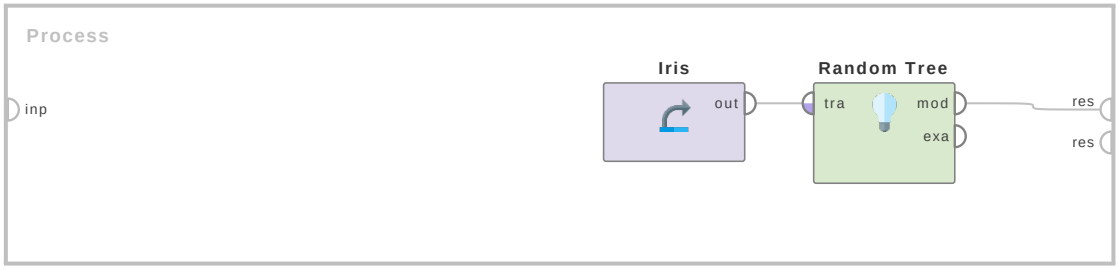
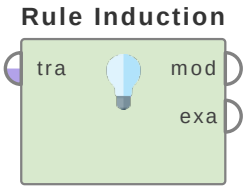



Figure 4.23: Tutorial process 'Introduction to the Random Tree operator'.

The 'Iris' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. The Random Tree operator is applied on this ExampleSet with default values of all parameters. The resultant tree is connected to the result port of the process and it can be seen in the Results Workspace.

# 4.1.4 Rules

## Rule Induction




This operator learns a pruned set of rules with respect to the information gain from the given ExampleSet.

### Description

The Rule Induction operator works similar to the propositional rule learner named 'Repeated Incremental Pruning to Produce Error Reduction' (RIPPER, Cohen 1995). Starting with the less prevalent classes, the algorithm iteratively grows and prunes rules until there are no positive examples left or the error rate is greater than 50%.

In the growing phase, for each rule greedily conditions are added to the rule until it is perfect (i.e. 100% accurate). The procedure tries every possible value of each attribute and selects the condition with highest information gain.

In the prune phase, for each rule any final sequences of the antecedents is pruned with the pruning metric *p/(p+n)*.

Rule Set learners are often compared to Decision Tree learners. Rule Sets have the advantage that they are easy to understand, representable in first order logic (easy to implement in languages like Prolog) and prior knowledge can be added to them easily. The major disadvantages of Rule Sets were that they scaled poorly with training set size and had problems with noisy data. The RIPPER algorithm (which this operator implements) pretty much overcomes these disadvantages. The major problem with Decision Trees is overfitting i.e. the model works very well on the training set but does not perform well on the validation set. Reduced Error Pruning (REP) is a technique that tries to overcome overfitting. After various improvements and enhancements over the period of time REP changed to IREP, IREP* and RIPPER.

Pruning in decision trees is a technique in which leaf nodes that do not add to the discriminative power of the decision tree are removed. This is done to convert an over-specific or overfitted tree to a more general form in order to enhance its predictive power on unseen datasets. A similar concept of pruning implies on Rule Sets.

### Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Discretize by Frequency operator in the attached Example Process. The output of other operators can also be used as input.

### Output Ports

**model *(mod)*** The Rule Model is delivered from this output port. This model can now be applied on unseen data sets.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**criterion** ***(selection)*** This parameter specifies the criterion for selecting attributes and numerical splits. It can have one of the following values:

- **information_gain** The entropy of all the attributes is calculated. The attribute with minimum entropy is selected for split. This method has a bias towards selecting attributes with a large number of values.
- **accuracy** Such an attribute is selected for a split that maximizes the accuracy of the Rule Set.

**sample ratio** ***(real)*** This parameter specifies the sample ratio of training data used for growing and pruning.

**pureness** ***(real)*** This parameter specifies the desired pureness, i.e. the minimum ratio of the major class in a covered subset in order to consider the subset pure.

**minimal prune benefit** ***(real)*** This parameter specifies the minimum amount of benefit which must be exceeded over unpruned benefit in order to be pruned.

**use local random seed** ***(boolean)*** Indicates if a *local random seed* should be used for randomization.

**local random seed** ***(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Introduction to the Rule Induction operator

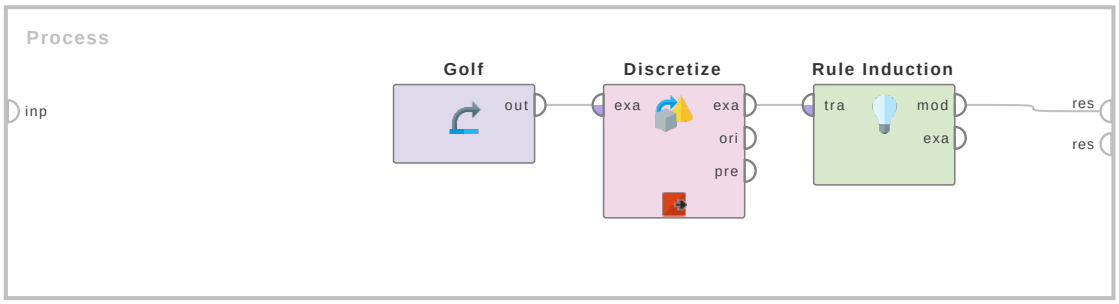
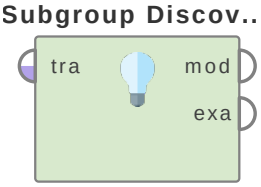



Figure 4.24: Tutorial process 'Introduction to the Rule Induction operator'.

The 'Golf' data set is loaded using the Retrieve operator. The Discretize by Frequency operator is applied on it to convert the numerical attributes to nominal attributes. This is done because the Rule Learners usually perform well on nominal attributes. The number of bins parameter of the Discretize by Frequency operator is set to 3. All other parameters are used with default values. A breakpoint is inserted here so that you can have a look at the ExampleSet before application of the Rule Induction operator. The Rule Induction operator is applied next. All parameters are used with default values. The resulting model is connected to the result port of the process. The Rule Set (RuleModel) can be seen in the Results Workspace after execution of the process.

# Subgroup Discovery




This operator performs an exhaustive subgroup discovery. The goal of subgroup discovery is to find rules describing subsets of the population that are sufficiently large and statistically unusual.

## Description

This operator discovers subgroups (or induces a rule set) by generating hypotheses exhaustively. Generation is done by stepwise refining the empty hypothesis (which contains no literals). The loop for this task hence iterates over the depth of the search space, i.e. the number of literals of the generated hypotheses. The maximum depth of the search can be specified by the *max depth* parameter. Furthermore the search space can be pruned by specifying a minimum coverage (by the *min coverage* parameter) of the hypothesis or by using only a given amount of hypotheses which have the highest coverage. From the hypotheses, rules are derived according to the user's preference. This operator allows the derivation of positive rules and negative rules separately or the combination by deriving both rules or only the one which is the most probable due to the examples covered by the hypothesis (hence: the actual prediction for that subset). This behavior can be controlled by the *rule generation* parameter. All generated rules are evaluated on the ExampleSet by a user specified utility function (which is specified by the *utility function* parameter) and stored in the final rule set if:

- They exceed a minimum utility threshold (which is specified by the *min utility* parameter) or
- They are among the k best rules (where k is specified by the *k best rules* parameter).

The desired behavior can be specified by the *mode* parameter.

The problem of subgroup discovery has been defined as follows: Given a population of individuals and a property of those individuals we are interested in finding population subgroups that are statistically most interesting, e.g. are as large as possible and have the most unusual statistical (distributional) characteristics with respect to the property of interest. In subgroup discovery, rules have the form *Class* >- *Cond*, where the property of interest for subgroup discovery is the class value *Class* which appears in the rule consequent, and the rule antecedent *Cond* is a conjunction of features (attribute-value pairs) selected from the features describing the training instances. As rules are induced from labeled training instances (labeled positive if the property of interest holds, and negative otherwise), the process of subgroup discovery is targeted at uncovering properties of a selected target population of individuals with the given property of interest. In this sense, subgroup discovery is a form of supervised learning. However, in many respects subgroup discovery is a form of descriptive induction as the task is to uncover individual interesting patterns in data.

Rule learning is most frequently used in the context of classification rule learning and association rule learning. While classification rule learning is an approach to predictive induction (or supervised learning), aimed at constructing a set of rules to be used for classification and/or prediction, association rule learning is a form of descriptive induction (non- classification induction or unsupervised learning), aimed at the discovery of individual rules which define interesting patterns in data.

Let us emphasize the difference between subgroup discovery (as a task at the intersection of predictive and descriptive induction) and classification rule learning (as a form of predictive induction). The goal of standard rule learning is to generate models, one for each class, consisting of rule sets describing class characteristics in terms of properties occurring in the descriptions of training examples. In contrast, subgroup discovery aims at discovering individual rules or 'patterns' of interest, which must be represented in explicit symbolic form and which must be relatively simple in order to be recognized as actionable by potential users. Moreover, standard classification rule learning algorithms cannot appropriately address the task of subgroup discovery as they use the covering algorithm for rule set construction which hinders the applicability of classification rule induction approaches in subgroup discovery. Subgroup discovery is usually seen as different from classification, as it addresses different goals (discovery of interesting population subgroups instead of maximizing classification accuracy of the induced rule set).

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Generate Nominal Data operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The Rule Set is delivered from this output port.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**mode *(selection)*** This parameter specifies the discovery mode.

- **minimum_utility** If this option is selected the rules are stored in the final rule set if they exceed the minimum utility threshold specified by the *min utility* parameter
- **k_best_rules** If this option is selected the rules are stored in the final rule set if they are among the k best rules (where k is specified by the *k best rules* parameter).

**utility function *(selection)*** This parameter specifies the desired utility function.

**min utility *(real)*** This parameter specifies the minimum utility. This parameter is useful when the *mode* parameter is set to 'minimum utility'. The rules are stored in the final rule set if they exceed the minimum utility threshold specified by this parameter.

**k best rules *(integer)*** This parameter specifies the number of required best rules. This parameter is useful when the *mode* parameter is set to 'k best rules'. The rules are stored in the final rule set if they are among the *k* best rules where *k* is specified by this parameter.

**rule generation *(selection)*** This parameter determines which rules should be generated. This operator allows the derivation of positive rules and negative rules separately or the combination by deriving both rules or only the one which is the most probable due to the examples covered by the hypothesis (hence: the actual prediction for that subset).

**max depth** ***(integer)*** This parameter specifies the maximum depth of breadth-first search. The loop for this task iterates over the depth of the search space, i.e. the number of literals of the generated hypotheses. The maximum depth of the search can be specified by this parameter

**min coverage** ***(real)*** This parameter specifies the minimum coverage. Only the rules which exceed this coverage threshold are considered.

**max cache** ***(integer)*** This parameter bounds the number of rules which are evaluated (only the most supported rules are used).

## Tutorial Processes

### Introduction to the Subgroup Discovery operator

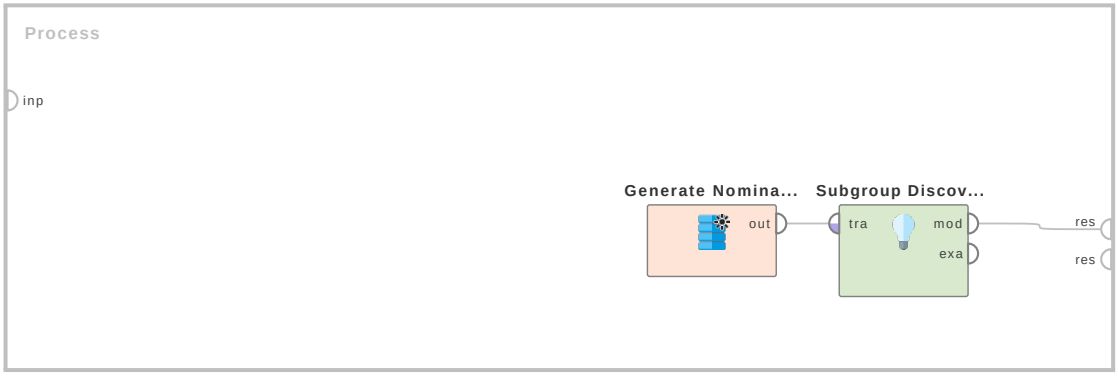
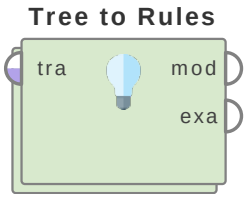



Figure 4.25: Tutorial process 'Introduction to the Subgroup Discovery operator'.

The Generate Nominal Data operator is used for generating an ExampleSet. The ExampleSet has two binominal attributes with 100 examples. The Subgroup Discovery operator is applied on this ExampleSet with default values of all parameters. The mode parameter is set to 'k best rules' and the k best rules parameter is set to 10. Moreover the utility function parameter is set to 'WRAcc'. Thus the Rule Set will be composed of 10 best rules where rules are evaluated by the WRAcc function. The resultant Rule Set can be seen in the Results Workspace. You can see that there are 10 rules and they are sorted in order of their WRAcc values.

# Tree to Rules




This operator is a meta learner. It uses an inner tree learner for creating a rule model.

## Description

The Tree to Rules operator determines a set of rules from the given decision tree model. This operator is a nested operator i.e. it has a subprocess. The subprocess must have a tree learner i.e. an operator that expects an ExampleSet and generates a tree model. This operator builds a rule model using the tree learner provided in its subprocess. You need to have basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

Decision tree is a predictive model which maps observations about an item to conclusions about the item's target value. In these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. In decision analysis, a decision tree can be used to visually and explicitly represent decisions and decision making.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The rule model is delivered from this output port which can now be applied on unseen data sets for prediction of the *label* attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Tutorial Processes

### Generating rules from a Decision Tree

The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at this ExampleSet. The Tree to Rules operator is applied on this ExampleSet. The Decision Tree operator is applied in the subprocess of the Tree to Rules operator. A breakpoint is inserted after the Decision Tree operator so that you can have a look at the Decision Tree. The Tree to Rules operator generates a rule model from this Tree. The resultant rule model can be seen in the Results Workspace.

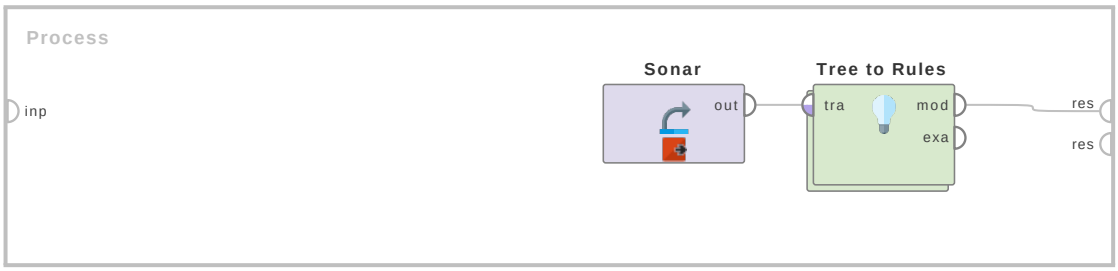
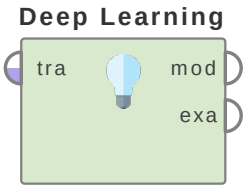



Figure 4.26: Tutorial process 'Generating rules from a Decision Tree'.

## 4.1.5 Neural Nets

### Deep Learning




Executes Deep Learning algorithm using H2O 3.8.2.6.

#### Description

*Please note that this algorithm is deterministic only if the* reproducible*parameter is set to true. In this case the algorithm uses only 1 thread.*

Deep Learning is based on a multi-layer feed-forward artificial neural network that is trained with stochastic gradient descent using back-propagation. The network can contain a large number of hidden layers consisting of neurons with tanh, rectifier and maxout activation functions. Advanced features such as adaptive learning rate, rate annealing, momentum training, dropout and L1 or L2 regularization enable high predictive accuracy. Each compute node trains a copy of the global model parameters on its local data with multi-threading (asynchronously), and contributes periodically to the global model via model averaging across the network.

The operator starts a 1-node local H2O cluster and runs the algorithm on it. Although it uses one node, the execution is parallel. You can set the level of parallelism by changing the Settings/Preferences/General/Number of threads setting. By default it uses the recommended number of threads for the system. Only one instance of the cluster is started and it remains running until you close RapidMiner Studio.

#### Input Ports

**training set *(tra)*** The input port expects a labeled ExampleSet.

#### Output Ports

**model *(mod)*** The Deep Learning classification or regression model is delivered from this output port. This classification or regression model can be applied on unseen data sets for prediction of the label attribute.

**example set** ***(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**activation** ***(selection)*** The activation function (non-linearity) to be used by the neurons in the hidden layers.

- **Tanh** Hyperbolic tangent function (same as scaled and shifted sigmoid).
- **Rectifier** Rectifier Linear Unit: Chooses the maximum of (0, x) where x is the input value.
- **Maxout** Choose the maximum coordinate of the input vector.
- **ExpRectifier** Exponential Rectifier Linear Unit function. With Dropout options: Zero out a random user-given fraction of the incoming weights to each hidden layer during training, for each training row. This effectively trains exponentially many models at once, and can improve generalization. In this case the hidden_dropout_ratios parameter is used.

**hidden layer sizes** ***(enumeration)*** The number and size of each hidden layer in the model. For example, if a user specifies "100,200,100" a model with 3 hidden layers will be produced, and the middle hidden layer will have 200 neurons.

**hidden dropout ratios** ***(enumeration)*** A fraction of the inputs for each hidden layer to be omitted from training in order to improve generalization. Defaults to 0.5 for each hidden layer if omitted. Visible only if an activation function with dropout is selected.

**reproducible** ***(boolean)*** Force reproducibility on small data (will be slow - only uses 1 thread).

**use local random seed** ***(boolean)*** Indicates if a local random seed should be used for randomization. Available only if reproducible is set to true.

**local random seed** ***(integer)*** Local random seed for random generation. This parameter is only available if the use local random seed parameter is set to true.

**epochs** ***(real)*** How many times the dataset should be iterated (streamed), can be fractional.

**compute variable importances** ***(boolean)*** Whether to compute variable importances for input features. The implemented method considers the weights connecting the input features to the first two hidden layers.

**train samples per iteration** ***(long)*** The number of training data rows to be processed per iteration. Note that independent of this parameter, each row is used immediately to update the model with (online) stochastic gradient descent. This parameter controls the frequency at which scoring and model cancellation can happen. Special values are 0 for one epoch per iteration, -1 for processing the maximum amount of data per iteration. Special value of -2 turns on automatic mode (auto-tuning).

**adaptive rate** ***(boolean)*** The implemented adaptive learning rate algorithm (ADADELTA) automatically combines the benefits of learning rate annealing and momentum training to avoid slow convergence. Specification of only two parameters (rho and epsilon) simplifies hyper parameter search. In some cases, manually controlled (non-adaptive) learning rate and momentum specifications can lead to better results, but require the specification

(and hyper parameter search) of up to 7 parameters. If the model is built on a topology with many local minima or long plateaus, it is possible for a constant learning rate to produce sub-optimal results. Learning rate annealing allows digging deeper into local minima, while rate decay allows specification of different learning rates per layer. When the gradient is being estimated in a long valley in the optimization landscape, a large learning rate can cause the gradient to oscillate and move in the wrong direction. When the gradient is computed on a relatively flat surface with small learning rates, the model can converge far slower than necessary.

**epsilon** ***(real)*** Similar to learning rate annealing during initial training and momentum at later stages where it allows forward progress. Typical values are between 1e-10 and 1e-4. This parameter is only active if adaptive learning rate is enabled.

**rho** ***(real)*** Similar to momentum and relates to the memory to prior weight updates. Typical values are between 0.9 and 0.999. This parameter is only active if adaptive learning rate is enabled.

**standardize** ***(boolean)*** If enabled, automatically standardize the data. If disabled, the user must provide properly scaled input data.

**learning rate** ***(real)*** When adaptive learning rate is disabled, the magnitude of the weight updates are determined by the user specified learning rate (potentially annealed), and are a function of the difference between the predicted value and the target value. That difference, generally called delta, is only available at the output layer. To correct the output at each hidden layer, back propagation is used. Momentum modifies back propagation by allowing prior iterations to influence the current update. Using the momentum parameter can aid in avoiding local minima and the associated instability. Too much momentum can lead to instabilities, that's why the momentum is best ramped up slowly. This parameter is only active if adaptive learning rate is disabled.

**rate annealing** ***(real)*** Learning rate annealing reduces the learning rate to "freeze" into local minima in the optimization landscape. The annealing rate is the inverse of the number of training samples it takes to cut the learning rate in half (e.g., 1e-6 means that it takes 1e6 training samples to halve the learning rate). This parameter is only active if adaptive learning rate is disabled.

**rate decay** ***(real)*** The learning rate decay parameter controls the change of learning rate across layers. For example, assume the rate parameter is set to 0.01, and the rate_decay parameter is set to 0.5. Then the learning rate for the weights connecting the input and first hidden layer will be 0.01, the learning rate for the weights connecting the first and the second hidden layer will be 0.005, and the learning rate for the weights connecting the second and third hidden layer will be 0.0025, etc. This parameter is only active if adaptive learning rate is disabled.

**momentum start** ***(real)*** The momentum_start parameter controls the amount of momentum at the beginning of training. This parameter is only active if adaptive learning rate is disabled.

**momentum ramp** ***(real)*** The momentum_ramp parameter controls the amount of learning for which momentum increases (assuming momentum_stable is larger than momentum_start). The ramp is measured in the number of training samples. This parameter is only active if adaptive learning rate is disabled.

**momentum stable** ***(real)*** The momentum_stable parameter controls the final momentum value reached after momentum_ramp training samples. The momentum used for training will remain the same for training beyond reaching that point. This parameter is only active if adaptive learning rate is disabled.

**nesterov accelerated gradient** ***(boolean)*** The Nesterov accelerated gradient descent method is a modification to traditional gradient descent for convex functions. The method relies on gradient information at various points to build a polynomial approximation that minimizes the residuals in fewer iterations of the descent. This parameter is only active if adaptive learning rate is disabled.

**L1** ***(real)*** A regularization method that constrains the absolute value of the weights and has the net effect of dropping some weights (setting them to zero) from a model to reduce complexity and avoid overfitting.

**L2** ***(real)*** A regularization method that constrains the sum of the squared weights. This method introduces bias into parameter estimates, but frequently produces substantial gains in modeling as estimate variance is reduced.

**max w2** ***(real)*** A maximum on the sum of the squared incoming weights into any one neuron. This tuning parameter is especially useful for unbound activation functions such as Maxout or Rectifier. A special value of 0 means infinity.

**loss function** ***(selection)*** The loss (error) function to be minimized by the model. Absolute, Quadratic, and Huber are applicable for regression or classification, while CrossEntropy is only applicable for classification. Huber can improve for regression problems with outliers. CrossEntropy loss is used when the model output consists of independent hypotheses, and the outputs can be interpreted as the probability that each hypothesis is true. Cross entropy is the recommended loss function when the target values are class labels, and especially for imbalanced data. It strongly penalizes error in the prediction of the actual class label. Quadratic loss is used when the model output are continuous real values, but can be used for classification as well (where it emphasizes the error on all output classes, not just for the actual class).

**Automatic**

**Quadratic**

**Cross Entropy**

**Huber**

**Absolute**

**Quantile**

**distribution function** ***(selection)*** The distribution function for the training data. For some function (e.g. tweedie) further tuning can be achieved via the expert parameters

- **AUTO** Automatic selection. Uses multinomial for nominal and gaussian for numeric labels.
- **bernoulli** Bernoulli distribution. Can be used for binominal or 2-class polynominal labels.
- **gaussian, possion, gamma, tweedie, quantile, laplace** Distribution functions for regression.

**early stopping** ***(boolean)*** If true, parameters for early stopping needs to be specified.

**stopping rounds** ***(integer)*** Early stopping based on convergence of stopping_metric. Stop if simple moving average of length k of the stopping_metric does not improve for k:=stopping_rounds scoring events (0 to disable). This parameter is visible only if early_stopping is set.

**stopping metric** ***(selection)*** Metric to use for early stopping. Set stopping_tolerance to tune it. This parameter is visible only if early_stopping is set.

- **AUTO** Automatic selection. Uses logloss for classification, deviance for regression.
- **deviance, logloss, MSE, AUC, lift_top_group, r2, misclassification, mean_per_class_error** The metric to use to decide if the algorithm should be stopped.

**stopping tolerance** ***(real)*** Relative tolerance for metric-based stopping criterion (stop if relative improvement is not at least this much). This parameter is visible only if early_stopping is set.

**missing values handling** ***(selection)*** Handling of missing values. Either Skip or MeanImputation.

- **Skip** Missing values are skipped.
- **MeanImputation** Missing values are replaced with the mean value.

**max runtime seconds** ***(integer)*** Maximum allowed runtime in seconds for model training. Use 0 to disable.

**expert parameters** ***(enumeration)*** These parameters are for fine tuning the algorithm. Usually the default values provide a decent model, but in some cases it may be useful to change them. Please use true/false values for boolean parameters and the exact attribute name for columns. Arrays can be provided by splitting the values with the comma (,) character. More information on the parameters can be found in the H2O documentation.

- **score_each_iteration** Whether to score during each iteration of model training. Type: boolean, Default: false
- **fold_assignment** Cross-validation fold assignment scheme, if fold_column is not specified. Options: AUTO, Random, Modulo, Stratified. Type: enumeration, Default: AUTO
- **fold_column** Column name with cross-validation fold index assignment per observation. Type: column, Default: no fold column
- **offset_column** Offset column name. Type: Column, Default: no offset column
- **balance_classes** Balance training data class counts via over/under-sampling (for imbalanced data). Type: boolean, Default: false
- **keep_cross_validation_predictions** Keep cross-validation model predictions. Type: boolean, Default: false
- **keep_cross_validation_fold_assignment** Keep cross-validation fold assignment. Type: boolean, Default: false
- **max_after_balance_size** Maximum relative size of the training data after balancing class counts (can be less than 1.0). Requires balance_classes. Type: real, Default: 5.0
- **max_confusion_matrix_size** Maximum size (# classes) for confusion matrices to be printed in the Logs. Type: integer, Default: 20
- **quantile_alpha** Desired quantile for quantile regression (from 0.0 to 1.0). Type: real, Default: 0.5

- **tweedie_power** Tweedie Power (between 1 and 2). Type: real, Default: 1.5
- **class_sampling_factors** Desired over/under-sampling ratios per class (in lexicographic order). If not specified, sampling factors will be automatically computed to obtain class balance during training. Requires balance_classes=true. Type: float array, Default: empty
- **input_dropout_ratio** A fraction of the features for each training row to be omitted from training in order to improve generalization (dimension sampling). Type: real, Default: 0
- **score_interval** The minimum time (in seconds) to elapse between model scoring. The actual interval is determined by the number of training samples per iteration and the scoring duty cycle. Type: integer, Default: 5
- **score_training_samples** The number of training dataset points to be used for scoring. Will be randomly sampled. Use 0 for selecting the entire training dataset. Type: integer, Default: 10000
- **score_validation_samples** The number of validation dataset points to be used for scoring. Can be randomly sampled or stratified (if "balance classes" is set and "score validation sampling" is set to stratify). Use 0 for selecting the entire training dataset. Type: integer, Default: 0
- **score_duty_cycle** Maximum fraction of wall clock time spent on model scoring on training and validation samples, and on diagnostics such as computation of feature importances (i.e., not on training). Lower: more training, higher: more scoring. Type: real, Default: 0.1
- **overwrite_with_best_model** If enabled, store the best model under the destination key of this model at the end of training. Type: boolean, Default: true.
- **initial_weight_distribution** The distribution from which initial weights are to be drawn. The default option is an optimized initialization that considers the size of the network. The "uniform" option uses a uniform distribution with a mean of 0 and a given interval. The "normal" option draws weights from the standard normal distribution with a mean of 0 and given standard deviation. Type: enumeration, Options: UniformAdaptive, Uniform, Normal. Default: UniformAdaptive
- **initial_weight_scale** The scale of the distribution function for Uniform or Normal distributions. For Uniform, the values are drawn uniformly from -initial_weight_scale...initial_weight_scale. For Normal, the values are drawn from a Normal distribution with a standard deviation of initial_weight_scale. Type: real, Default: 1
- **classification_stop** The stopping criteria in terms of classification error (1-accuracy) on the training data scoring dataset. When the error is at or below this threshold, training stops. Type: real, Default: 0
- **regression_stop** The stopping criteria in terms of regression error (MSE) on the training data scoring dataset. When the error is at or below this threshold, training stops. Type: real, Default: 1e-6.
- **score_validation_sampling** Method used to sample the validation dataset for scoring. Type: enumeration, Options: Uniform, Stratified. Default: Uniform.
- **fast_mode** Enable fast mode (minor approximation in back-propagation), should not affect results significantly. Type: boolean, Default: true.
- **force_load_balance** Increase training speed on small datasets by splitting it into many chunks to allow utilization of all cores. Type: boolean, Default: true.

- **shuffle_training_data** Enable shuffling of training data (on each node). This option is recommended if training data is replicated on N nodes, and the number of training samples per iteration is close to N times the dataset size, where all nodes train will (almost) all the data. It is automatically enabled if the number of training samples per iteration is set to -1 (or to N times the dataset size or larger). This parameter usually doesn't need to be set, because RapidMiner runs H2O always on 1 node. Type: boolean, Default: false.
- **quiet_mode** Enable quiet mode for less output to standard output. Type: boolean, Default: false.
- **sparse** Sparse data handling (more efficient for data with lots of 0 values). Type: boolean, Default: false.
- **average_activation** Average activation for sparse auto-encoder (Experimental) Type: double, Default: 0.0.
- **sparsity_beta** Sparsity regularization. (Experimental) Type: double, Default: 0.0.
- **max_categorical_features** Max. number of categorical features, enforced via hashing (Experimental) Type: integer, Default: 2147483647.
- **export_weights_and_biases** Whether to export Neural Network weights and biases to H2O Frames. Type: boolean, Default: false.
- **mini_batch_size** Mini-batch size (smaller leads to better fit, larger can speed up and generalize better). Type: integer, Default: 1
- **elastic_averaging** Elastic averaging between compute nodes can improve distributed model convergence. (Experimental) Type: boolean, Default: false.
- **use_all_factor_levels** Use all factor levels of categorical variables. Otherwise, the first factor level is omitted (without loss of accuracy). Useful for variable importances and auto-enabled for autoencoder. Type: boolean, Default: true.
- **nfolds** Number of folds for cross-validation. Use 0 to turn off cross-validation. Type: integer, Default: 0

## Tutorial Processes

### Classification with Split Validation using Deep Learning

The H2O Deep Learning operator is used to predict the Survived attribute of the Titanic sample dataset. Since the label is binominal, classification will be performed. To check the quality of the model, the Split Validation operator is used to generate the training and testing datasets. The Deep Learning operator's parameters are the default values. This means that 2 hidden layers, each with 50 neurons will be constructed. The labeled ExampleSet is connected to a Performance (Binominal Classification) operator, that calculates the Accuracy metric. On the process output the Deep Learning Model, the labeled data and the Performance Vector is shown.

### Regression using Deep Learning

The H2O Deep Learning operator is used to predict the numerical label attribute of a generated dataset. Since the label is real, regression is performed. The data is generated, then splitted into two parts with the Split Data operator. The first output is used as the training, the second as the scoring data set. The Deep Learning operator uses the adaptive learning rate option (default). The algorithm automatically determines the learning rate based on the epsilon and rho

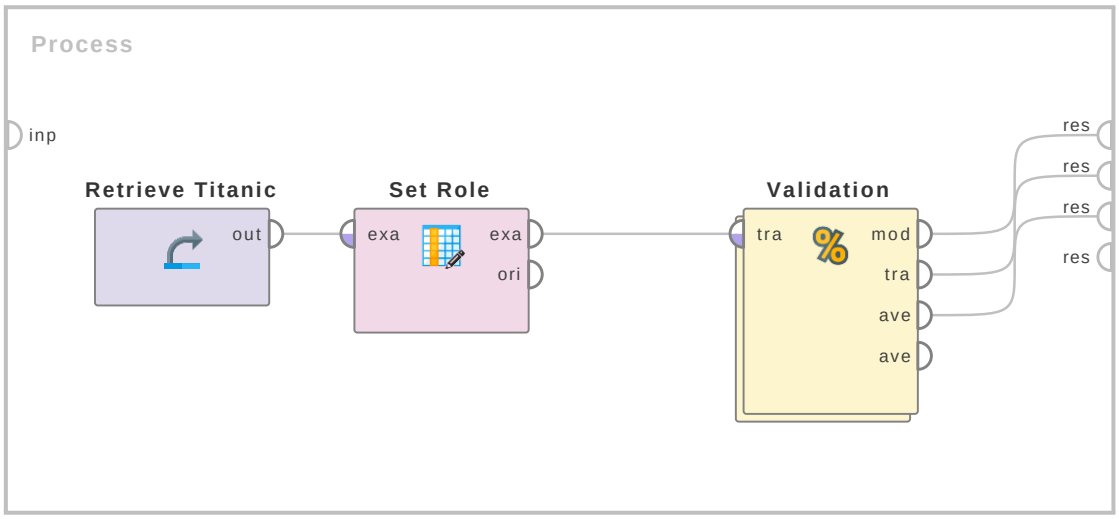



Figure 4.27: Tutorial process 'Classification with Split Validation using Deep Learning'.

parameters. The only non-default parameter is the hidden layer sizes, where 3 layers are used, each with 50 neurons. After applying on the testing ExampleSet, the labelled data is connected to the process output.

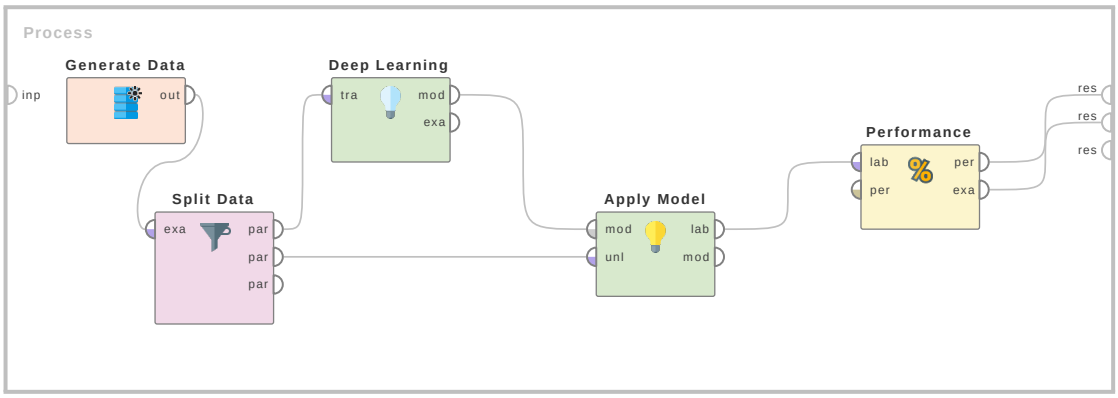
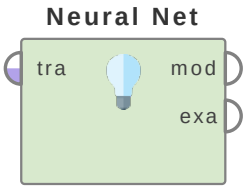



Figure 4.28: Tutorial process 'Regression using Deep Learning'.

# Neural Net




This operator learns a model by means of a feed-forward neural network trained by a back propagation algorithm (multi-layer perceptron). This operator cannot handle polynominal attributes.

## Description

This operator learns a model by means of a feed-forward neural network trained by a back propagation algorithm (multi-layer perceptron). The coming paragraphs explain the basic ideas about neural networks, need-forward neural networks, back-propagation and multi-layer perceptron.

An artificial neural network (ANN), usually called neural network (NN), is a mathematical model or computational model that is inspired by the structure and functional aspects of biological neural networks. A neural network consists of an interconnected group of artificial neurons, and it processes information using a connectionist approach to computation (the central connectionist principle is that mental phenomena can be described by interconnected networks of simple and often uniform units). In most cases an ANN is an adaptive system that changes its structure based on external or internal information that flows through the network during the learning phase. Modern neural networks are usually used to model complex relationships between inputs and outputs or to find patterns in data.

A feed-forward neural network is an artificial neural network where connections between the units do not form a directed cycle. In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) to the output nodes. There are no cycles or loops in the network.

Back propagation algorithm is a supervised learning method which can be divided into two phases: propagation and weight update. The two phases are repeated until the performance of the network is good enough. In back propagation algorithms, the output values are compared with the correct answer to compute the value of some predefined error-function. By various techniques, the error is then fed back through the network. Using this information, the algorithm adjusts the weights of each connection in order to reduce the value of the error function by some small amount. After repeating this process for a sufficiently large number of training

cycles, the network will usually converge to some state where the error of the calculations is small. In this case, one would say that the network has learned a certain target function.

A multilayer perceptron (MLP) is a feed-forward artificial neural network model that maps sets of input data onto a set of appropriate output. An MLP consists of multiple layers of nodes in a directed graph, with each layer fully connected to the next one. Except for the input nodes, each node is a neuron (or processing element) with a nonlinear activation function. MLP utilizes back propagation for training the network. This class of networks consists of multiple layers of computational units, usually interconnected in a feed-forward way. In many applications the units of these networks apply a sigmoid function as an activation function.

In this operator usual sigmoid function is used as the activation function. Therefore, the values ranges of the attributes should be scaled to -1 and +1. This can be done through the *normalize* parameter. The type of the output node is sigmoid if the learning data describes a classification task and linear if the learning data describes a numerical regression task.

## Input Ports

**training set *(tra)*** The input port expects an ExampleSet. It is output of the Retrieve operator in our example process. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The Neural Net model is delivered from this output port. This model can now be applied on unseen data sets for prediction of the *label* attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**hidden layers** This parameter describes the name and the size of all hidden layers. The user can define the structure of the neural network with this parameter. Each list entry describes a new hidden layer. Each entry requires name and size of the hidden layer. The layer name can be chosen arbitrarily. It is only used for displaying the model. Note that the actual number of nodes will be one more than the value specified as hidden layer size because an additional constant node will be added to each layer. This node will not be connected to the preceding layer. If the hidden layer size value is set to -1 the layer size would be calculated from the number of attributes of the input example set. In this case, the layer size will be set to *(number of attributes + number of classes) / 2 + 1*. If the user does not specify any hidden layers, a default hidden layer with sigmoid type and size equal to *(number of attributes + number of classes) / 2 + 1* will be created and added to the net. If only a single layer without nodes is specified, the input nodes are directly connected to the output nodes and no hidden layer will be used.

**training cycles *(integer)*** This parameter specifies the number of training cycles used for the neural network training. In back-propagation the output values are compared with the correct answer to compute the value of some predefined error-function. The error is then fed back through the network. Using this information, the algorithm adjusts the weights of each connection in order to reduce the value of the error function by some small amount. This process is repeated *n* number of times. *n* can be specified using this parameter.

**learning rate** ***(real)*** This parameter determines how much we change the weights at each step. It should not be 0.

**momentum** ***(real)*** The momentum simply adds a fraction of the previous weight update to the current one. This prevents local maxima and smoothes optimization directions.

**decay** ***(boolean)*** This is an expert parameter. It indicates if the learning rate should be decreased during learning.

**shuffle** ***(boolean)*** This is an expert parameter. It indicates if the input data should be shuffled before learning. Although it increases memory usage but it is recommended if data is sorted before.

**normalize** ***(boolean)*** This is an expert parameter. The Neural Net operator uses an usual sigmoid function as the activation function. Therefore, the value range of the attributes should be scaled to -1 and +1. This can be done through the normalize parameter. Normalization is performed before learning. Although it increases runtime but it is necessary in most cases.

**error epsilon** ***(real)*** The optimization is stopped if the training error gets below this epsilon value.

**use local random seed** ***(boolean)*** Indicates if a *local random seed* should be used for randomization.

**local random seed** ***(integer)*** This parameter specifies the *local random seed*. It is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Introduction to Neural Net

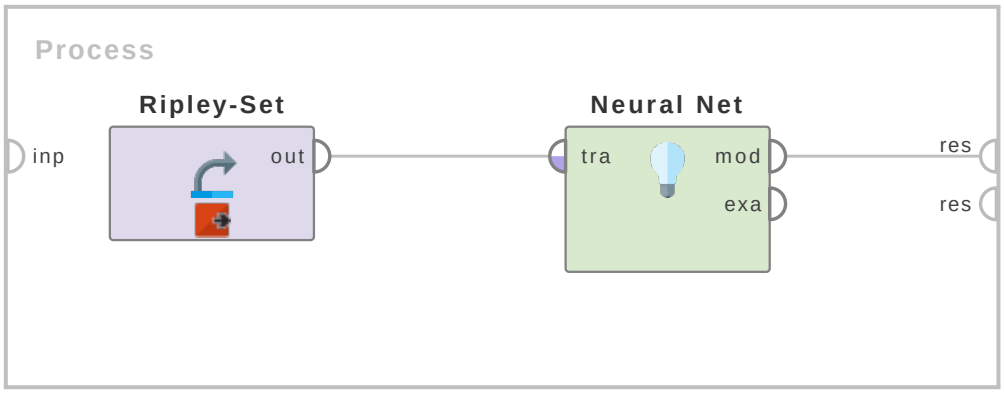


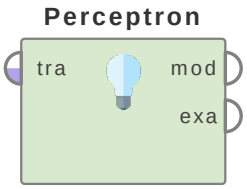

Figure 4.29: Tutorial process 'Introduction to Neural Net'.

The 'Ripley' data set is loaded using the Retrieve operator. A breakpoint is inserted here so you can see the data set before the application of the Neural Net operator. You can see that this data set has two regular attributes i.e. att1 and att2. The label attribute has two possible values i.e. 1 or 0. Then the Neural Net operator is applied on it. All parameters are used with default values. When you run the process, you can see the neural net in the Results Workspace. There are x+1

number of nodes in the input, where x is the number of attributes in the input ExampleSet (other than label attribute). The last node is the threshold node. There are y number of nodes in the output, where y is the number of classes in the input ExampleSet (i.e. number of possible values of label attribute). As no value was specified in the hidden layers parameter, the default value is used. Therefore, the number of nodes are created in hidden layer are = size of hidden layer = (number of attributes + number of classes) / 2 + 1 = (2+2)/2+1= 3. The last node (4th node) is a threshold node. The connections between nodes are colored darker if the connection weight is high. You can click on a node in this visualization in order to see the actual weights.

This simple process just provides basic working of this operator. In real scenarios all parameters should be carefully chosen.

# Perceptron




This operator learns a linear classifier called Single Perceptron which finds separating hyperplane (if existent). This operator cannot handle polynominal attributes.

## Description

The perceptron is a type of artificial neural network invented in 1957 by Frank Rosenblatt. It can be seen as the simplest kind of feed-forward neural network: a linear classifier. Beside all biological analogies, the single layer perceptron is simply a linear classifier which is efficiently trained by a simple update rule: for all wrongly classified data points, the weight vector is either increased or decreased by the corresponding example values. The coming paragraphs explain the basic ideas about neural networks and feed-forward neural networks.

An artificial neural network (ANN), usually called neural network (NN), is a mathematical model or computational model that is inspired by the structure and functional aspects of biological neural networks. A neural network consists of an interconnected group of artificial neurons, and it processes information using a connectionist approach to computation (the central connectionist principle is that mental phenomena can be described by interconnected networks of simple and often uniform units). In most cases an ANN is an adaptive system that changes its structure based on external or internal information that flows through the network during the learning phase. Modern neural networks are usually used to model complex relationships between inputs and outputs or to find patterns in data.

A feed-forward neural network is an artificial neural network where connections between the units do not form a directed cycle. In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) to the output nodes. There are no cycles or loops in the network. If you want to use a more sophisticated neural net, please use the Neural Net operator.

## Input Ports

**training set *(tra)*** The input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The Hyperplane model is delivered from this output port. This model can now be applied on unseen data sets for the prediction of the *label* attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**rounds *(integer)*** This parameter specifies the number of datascans to use to adapt the hyperplane.

**learning rate *(real)*** This parameter determines how much the weights should be changed at each step. It should not be 0. The hyperplane will adapt to each example with this rate.

## Tutorial Processes

### Introduction to Perceptron operator

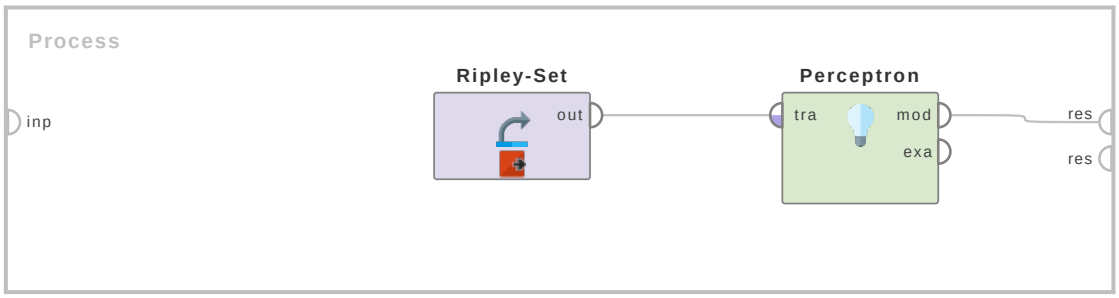
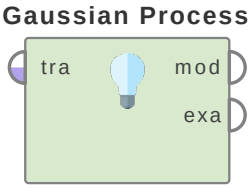



Figure 4.30: Tutorial process 'Introduction to Perceptron operator'.

The 'Ripley' data set is loaded using the Retrieve operator. A breakpoint is inserted here so you can see the data set before the application of the Perceptron operator. You can see that this data set has two regular attributes: att1 and att2. The label attribute has two possible values: 1 and 0. The Perceptron operator is applied on this ExampleSet. All parameters are used with default values. The rounds parameter is set to 3 and the learning rate parameter is set to 0.05. After running the process, you can see the resultant hyperplane model in the Results Workspace.

# 4.1.6 Functions

## Gaussian Process




This operator is an implementation of Gaussian Process (GP) which is a probabilistic method both for classification and regression.

### Description

A Gaussian process is a stochastic process whose realizations consist of random values associated with every point in a range of times (or of space) such that each such random variable has a normal distribution. Moreover, every finite collection of those random variables has a multivariate normal distribution. Gaussian processes are important in statistical modeling because of properties inherited from the normal. For example, if a random process is modeled as a Gaussian process, the distributions of various derived quantities can be obtained explicitly. Such quantities include: the average value of the process over a range of times; the error in estimating the average using sample values at a small set of times.

Gaussian processes (GPs) extend multivariate Gaussian distributions to infinite dimensionality. Formally, a Gaussian process generates data located throughout some domain such that any finite subset of the range follows a multivariate Gaussian distribution. Gaussian Process is a powerful non-parametric machine learning technique for constructing comprehensive probabilistic models of real world problems. They can be applied to geostatistics, supervised, unsupervised and reinforcement learning, principal component analysis, system identification and control, rendering music performance, optimization and many other tasks.

### Input Ports

**training set *(tra)*** This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before the application of this operator.

### Output Ports

**model *(mod)*** The Gaussian Process is applied and the resultant model is delivered from this output port. This model can now be applied on unseen data sets.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

### Parameters

**kernel type *(selection)*** The type of the kernel function is selected through this parameter. Following kernel types are supported: *rbf, cauchy, laplace, poly, sigmoid, Epanechnikov, gaussian combination, multiquadric.*

**kernel lengthscale *(real)*** This parameter specifies the lengthscale to be used in all kernels.

**kernel degree *(real)*** This is the kernel parameter degree. This is only available when the *kernel type* parameter is set to *polynomial* or *epachnenikov*.

**kernel bias *(real)*** This parameter specifies the bias to be used in the poly kernel.

**kernel sigma1 *(real)*** This is the kernel parameter sigma1. This is only available when the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2 *(real)*** This is the kernel parameter sigma2. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3 *(real)*** This is the kernel parameter sigma3. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel shift *(real)*** This is the kernel parameter shift. This is only available when the *kernel type* parameter is set to *multiquadric*.

**kernel a *(real)*** This is the kernel parameter a. This is only available when the *kernel type* parameter is set to *sigmoid*

**kernel b *(real)*** This is the kernel parameter b. This is only available when the *kernel type* parameter is set to *sigmoid*

**max basis vectors *(integer)*** This parameter specifies the maximum number of basis vectors to be used.

**epsilon tol *(real)*** This parameter specifies the tolerance for gamma induced projections.

**geometrical tol *(real)*** This parameter specifies the tolerance for geometry induced projections.

## Tutorial Processes

### Introduction to the Gaussian Process operator

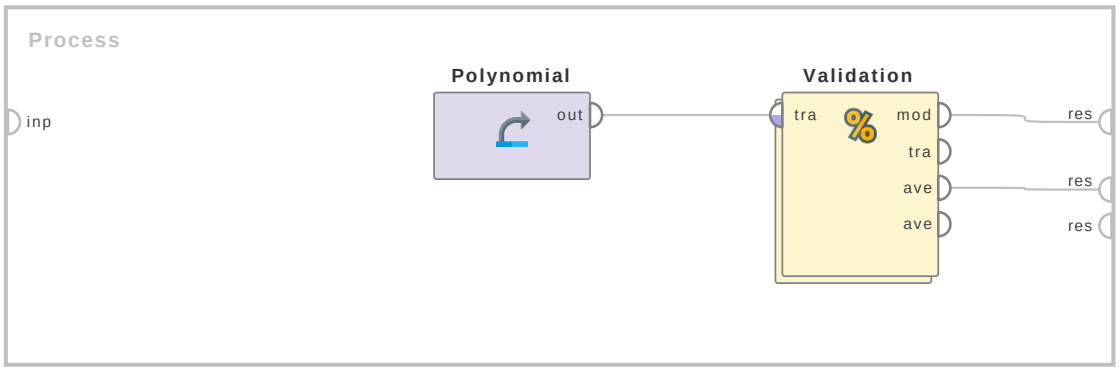
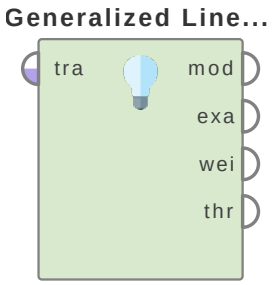



Figure 4.31: Tutorial process ‘Introduction to the Gaussian Process operator’.

The ‘Polynomial’ data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a regression model. The Gaussian Process operator is applied in the training subprocess of the Split Validation operator. All parameters are used with

default values. The Gaussian Process operator generates a regression model. The Apply Model operator is used in the testing subprocess to apply this model on the testing data set. The resultant labeled ExampleSet is used by the Performance operator for measuring the performance of the model. The regression model and its performance vector are connected to the output and it can be seen in the Results Workspace.

# Generalized Linear Model




Executes GLM algorithm using H2O 3.8.2.6.

## Description

*Please note that the result of this algorithm may depend on the number of threads used. Different settings may lead to slightly different outputs.*

Generalized linear models (GLMs) are an extension of traditional linear models. This algorithm fits generalized linear models to the data by maximizing the log-likelihood. The elastic net penalty can be used for parameter regularization. The model fitting computation is parallel, extremely fast, and scales extremely well for models with a limited number of predictors with non-zero coefficients.

The operator starts a 1-node local H2O cluster and runs the algorithm on it. Although it uses one node, the execution is parallel. You can set the level of parallelism by changing the Settings/Preferences/General/Number of threads setting. By default it uses the recommended number of threads for the system. Only one instance of the cluster is started and it remains running until you close RapidMiner Studio.

Please note that below version 7.6, a threshold value optimized for maximal F-measure is used for prediction by default.

## Input Ports

**training set *(tra)*** The input port expects a labeled ExampleSet.

## Output Ports

**model *(mod)*** The Generalized Linear classification or regression model is delivered from this output port. This classification or regression model can be applied on unseen data sets for prediction of the label attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute.

**threshold *(thr)*** This port is used only for binominal classification tasks. It provides a threshold value optimized for maximal F-measure. If you wish to use this threshold value calculated by H2O, connect this output to an *Apply Threshold* operator, along with the scored ExampleSet. (By default, RapidMiner uses 0.5 threshold value when applying models.)

## Parameters

**family** ***(selection)*** Family. Use binomial for classification with logistic regression, others are for regression problems.

- **AUTO** Automatic selection. Uses multinomial for polynominal, binomial for binominal and gaussian for numeric labels.
- **gaussian** The data must be numeric (real or integer).
- **binomial** The data must be binominal or polynominal with 2 levels/classes.
- **multinomial** The data must be polynominal with more than two levels/classes.
- **poisson** The data must be numeric and non-negative (integer).
- **gamma** The data must be numeric and continuous and positive (real or integer).
- **tweedie** The data must be numeric and continuous (real) and non-negative.

**solver** ***(selection)*** Select the solver to use. IRLSM is fast on problems with a small number of predictors and for lambda-search with L1 penalty, while L_BFGS scales better for datasets with many columns. COORDINATE_DESCENT is IRLSM with the covariance updates version of cyclical coordinate descent in the innermost loop. COORDINATE_DESCENT_NAIVE is IRLSM with the naive updates version of cyclical coordinate descent in the innermost loop. COORDINATE_DESCENT_NAIVE and COORDINATE_DESCENT are currently experimental. Values:

- AUTO
- IRLSM
- L_BFGS
- COORDINATE_DESCENT (experimental)
- COORDINATE_DESCENT_NAIVE (experimental)

**link** ***(selection)*** The link function relates the linear predictor to the distribution function. The default is the canonical link for the specified family. Only available for gaussian, poisson and gamma families, because only one link type is possible for the others:

- Family: binomial; Link: logit
- Family: multinomial; Link: multinomial
- Family: tweedie; Link: tweedie
- **family_default** Uses identity for gaussian, log for possion and inverse for gamma family.
- **identity** Possible family options: Gaussian, Poisson, Gamma
- **log** Possible family options: Gaussian, Poisson, Gamma
- **inverse** Possible family options: Gaussian, Gamma

**reproducible** ***(boolean)*** Makes model building reproducible. If set then maximum_number_of_threads parameter controls parallelism level of model building. If not set then parallelism level is defined by number of threads in General Preferences.

**maximum number of threads** ***(integer)*** Controls parallelism level of model building.

**specify beta constraints** ***(boolean)*** If enabled, beta constraints for the regular attributes can be provided.

**use regularization** ***(boolean)*** Check this box if regularization should be used. For regularization, you can specify the lambda, alpha and the lambda search related parameters. If alpha or lambda is undefined (default), H2O will calculate default values for them based on the training data and the other parameters. If this parameter is set to false, lambda is set to 0.0 (means no regularization).

**lambda** ***(real)*** The lambda parameter controls the amount of regularization applied. If lambda is 0.0, no regularization is applied and the alpha parameter is ignored (you can set this by disabling the use regularization parameter). The default value for lambda is calculated by H2O using a heuristic based on the training data. Providing multiple lambda values via the advanced parameters triggers a search.

**lambda search** ***(boolean)*** A logical value indicating whether to conduct a search over the space of lambda values, starting from the max lambda, given lambda will be interpreted as the min lambda. Default is false.

**number of lambdas** ***(integer)*** The number of lambda values when lambda search = true. 0 means no preference.

**lambda min ratio** ***(real)*** Smallest value for lambda as a fraction of lambda.max, the entry value, which is the smallest value for which all coefficients in the model are zero. If the number of observations is greater than the number of variables then default lambda_min_ratio = 0.0001; if the number of observations is less than the number of variables then default lambda_min_ratio = 0.01. Default is 0.0, which means no preference.

**early stopping** ***(boolean)*** Check this box if early stopping should be performed on the lambda search based on the stopping rounds and stopping tolerance parameters. The used stopping metric is always deviance.

**stopping rounds** ***(integer)*** Early stopping based on convergence of stopping_metric. Stop if simple moving average of length k of the stopping_metric does not improve for k:=stopping_rounds scoring events.

**stopping tolerance** ***(real)*** Relative tolerance for metric-based stopping criterion (stop if relative improvement is not at least this much).

**alpha** ***(real)*** The alpha parameter controls the distribution between the L1 (Lasso) and L2 (Ridge regression) penalties. A value of 1.0 for alpha represents Lasso, and an alpha value of 0.0 produces Ridge regression. Providing multiple alpha values via the advanced parameters triggers a search. Default is 0.0 for the L-BFGS solver, else 0.5.

**standardize** ***(boolean)*** Standardize numeric columns to have zero mean and unit variance

**non-negative coefficients** ***(boolean)*** Restrict coefficients (not intercept) to be non-negative.

**compute p-values** ***(boolean)*** Request p-values computation. P-values work only with IRLSM solver and no regularization. Intercept must also be added to the model. Moreover, non-negative coefficients and specify beta constraints parameters have to be set to false to compute p-values.

**remove collinear columns** ***(boolean)*** In case of linearly dependent columns remove some of the dependent columns. Works only if intercept is added to the model.

**add intercept** ***(boolean)*** Include constant term in the model.

**missing values handling** ***(selection)*** Handling of missing values. Either Skip or MeanImputation.

- **Skip** Missing values are skipped.
- **MeanImputation** Missing values are replaced with the mean value.

**max iterations** ***(integer)*** Maximum number of iterations. 0 means no limit.

**beta constraints** ***(list)*** Constraints for beta values. A row consists of the following values: Names

- Attribute name: The name of the attribute.
- Category: A value from the attribute's domain. Please take care to provide the exact value. Use more rows to specify constraints for multiple categories.

Constraints

- Lower bound: Lower bound of the beta.
- Upper bound: Upper bound of the beta.
- Beta given: Specifies the given solution in proximal operator interface. The proximal operator interface allows you to run the GLM with a proximal penalty on a distance from a specified given solution.
- Beta start: Starting value of the beta.

**max runtime seconds** ***(integer)*** Maximum allowed runtime in seconds for model training. Use 0 to disable.

**expert parameters** ***(enumeration)*** These parameters are for fine tuning the algorithm. Usually the default values provide a decent model, but in some cases it may be useful to change them. Please use true/false values for boolean parameters and the exact attribute name for columns. Arrays can be provided by splitting the values with the comma (,) character. More information on the parameters can be found in the H2O documentation.

- **score_each_iteration** Whether to score during each iteration of model training. Type: boolean, Default: false
- **fold_assignment** Cross-validation fold assignment scheme, if fold_column is not specified. Options: AUTO, Random, Modulo, Stratified. Type: enumeration, Default: AUTO
- **fold_column** Column name with cross-validation fold index assignment per observation. Type: column, Default: no fold column
- **offset_column** Offset column name. Type: Column, Default: no offset column
- **max_confusion_matrix_size** Maximum size (# classes) for confusion matrices to be printed in the Logs. Type: integer, Default: 20
- **keep_cross_validation_predictions** Keep cross-validation model predictions. Type: boolean, Default: false
- **keep_cross_validation_fold_assignment** Keep cross-validation fold assignment. Type: boolean, Default: false
- **tweedie_variance_power** A numeric value specifying the power for the variance function when family = "tweedie". Type: real, Default: 0
- **tweedie_link_power** A numeric value specifying the power for the link function when family = "tweedie". Type: real, Default: 1

- **prior** A numeric specifying the prior probability of class 1 in the response when family = "binomial". Must be from (0,1) exclusive range or -1 (no prior). The default value is the observation frequency of class 1. Type: real Default: -1 (no prior)
- **beta_epsilon** A non-negative number specifying the magnitude of the maximum difference between the coefficient estimates from successive iterations. Defines the convergence criterion. Type: real, Default: 0.0001
- **objective_epsilon** Specify a threshold for convergence. If the objective value is less than this threshold, the model is converged. Type: real, Default: -1 (no threshold)
- **gradient_epsilon** (For L-BFGS only) Specify a threshold for convergence. If the objective value (using the L-infinity norm) is less than this threshold, the model is converged. Type: real, Default: 0.0001
- **max_active_predictors** Specify the maximum number of active predictors during computation. This value is used as a stopping criterium to prevent expensive model building with many predictors. Type: integer, Default: -1 (no limit)
- **obj_reg** Likelihood divider in objective value computation, Type: real, Default: 1/nobs
- **additional_alphas** Providing additional alphas triggers a search. Ignored if alpha is undefined.
- **additional_lambdas** Providing additional lambdas triggers a search. Ignored if lambda is undefined.
- **nfolds** Number of folds for cross-validation. Use 0 to turn off cross-validation. Type: integer, Default: 0

## Tutorial Processes

### Classification using GLM

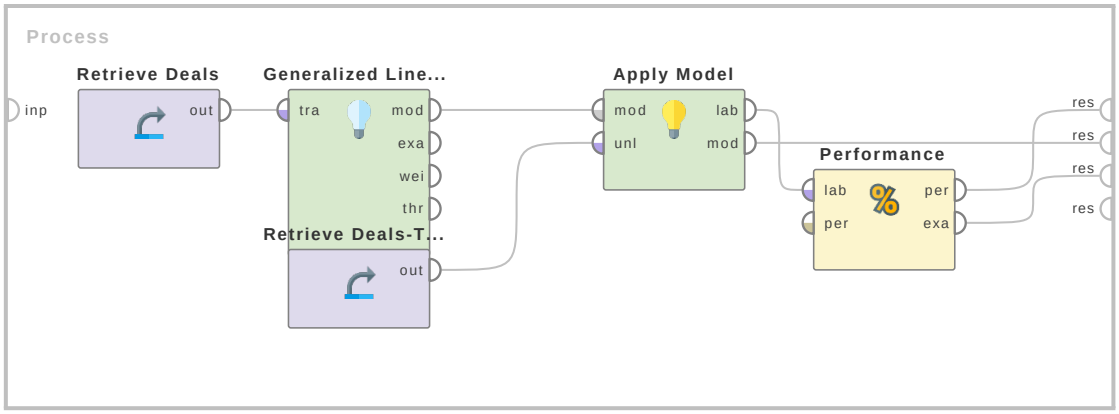



Figure 4.32: Tutorial process 'Classification using GLM'.

The GLM operator is used to predict the Future customer attribute of the Deals sample data set. All parameters are kept at the default value in the GLM. This means that because of the binominal label the Family parameter will be set automatically to "binominal", and the corresponding Link function to "logit". The resulting model is connected to an Apply Model operator that applies the Generalized Linear model on the Deals_Testset sample data. The labeled ExampleSet

is connected to a Performance (Binominal Classification) operator, that calculates the Accuracy metric. On the process output the Performance Vector, the Generalized Linear Model and the output ExampleSet is shown.

### Regression using GLM

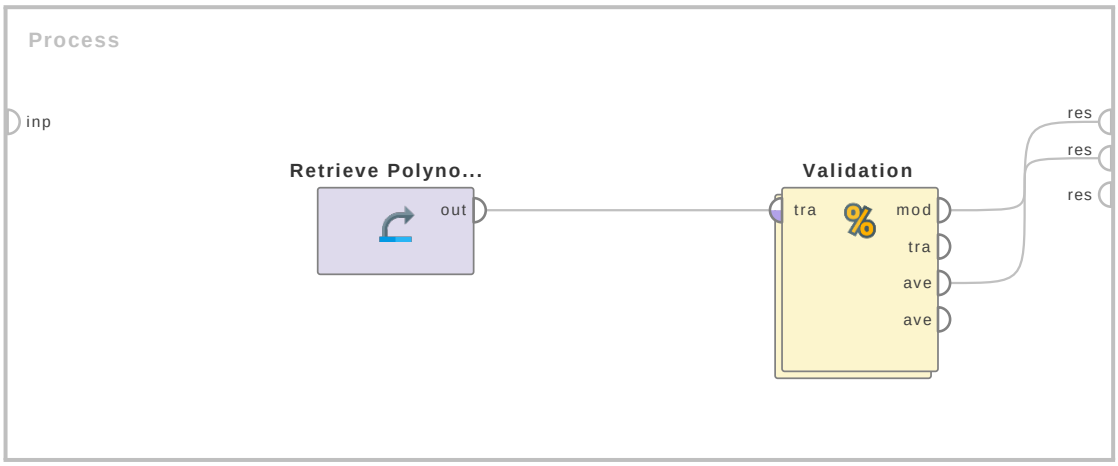
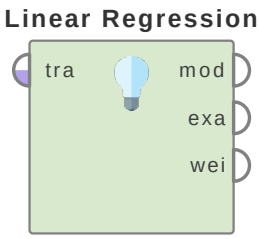



Figure 4.33: Tutorial process 'Regression using GLM'.

The GLM operator is used to predict the label attribute of the Polynominal sample data set using the Split Validation operator. The label is numerical, which means that regression is performed. The "compute p-values" parameter is set to true, which requires multiple parameters to be set: the lambda parameter is set to 0.0 (no regularization), the collinear columns are removed and no beta constraints are specified. The Solver parameter is set to AUTO, which means that the IRLSM solver is used - this allows the computation of the P-values. The resulting model is applied in the Testing subprocess of the Split Validation operator. The labeled ExampleSet is connected to a Performance (Regression Classification) operator, that calculates the Root mean squared error metric. On the process output the Performance Vector and the Generalized Linear Model is shown.

# Linear Regression




This operator calculates a linear regression model from the input ExampleSet.

## Description

Regression is a technique used for numerical prediction. Regression is a statistical measure that attempts to determine the strength of the relationship between one dependent variable ( i.e. the label attribute) and a series of other changing variables known as independent variables (regular attributes). Just like Classification is used for predicting categorical labels, Regression is used for predicting a continuous value. For example, we may wish to predict the salary of university graduates with 5 years of work experience, or the potential sales of a new product given its price. Regression is often used to determine how much specific factors such as the price of a commodity, interest rates, particular industries or sectors influence the price movement of an asset.

Linear regression attempts to model the relationship between a scalar variable and one or more explanatory variables by fitting a linear equation to observed data. For example, one might want to relate the weights of individuals to their heights using a linear regression model.

This operator calculates a linear regression model. It uses the Akaike criterion for model selection. The Akaike information criterion is a measure of the relative goodness of a fit of a statistical model. It is grounded in the concept of information entropy, in effect offering a relative measure of the information lost when a given model is used to describe reality. It can be said to describe the tradeoff between bias and variance in model construction, or loosely speaking between accuracy and complexity of the model.

## Differentiation

- **Polynomial Regression** Polynomial regression is a form of linear regression in which the relationship between the independent variable *x* and the dependent variable *y* is modeled as an *nth* order polynomial. See page 488 for details.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before application of this operator.

## Output Ports

**model *(mod)*** The regression model is delivered from this output port. This model can now be applied on unseen data sets.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**weights *(wei)*** This port delivers the attribute weights.

## Parameters

**feature selection *(selection)*** This is an expert parameter. It indicates the feature selection method to be used during regression. Following options are available: none, M5 prime, greedy, T-Test, iterative T-Test

**alpha *(real)*** This parameter is available only when the *feature selection* parameter is set to 'T-Test'. It specifies the value of *alpha* to be used in the T-Test feature selection.

**max iterations *(integer)*** This parameter is only available when the *feature selection* parameter is set to 'iterative T-Test'. It specifies the maximum number of iterations of the iterative T-Test for feature selection.

**forward alpha *(real)*** This parameter is only available when the *feature selection* parameter is set to 'iterative T-Test'. It specifies the value of *forward alpha* to be used in the T-Test feature selection.

**backward alpha *(real)*** This parameter is only available when the *feature selection* parameter is set to 'iterative T-Test'. It specifies the value of *backward alpha* to be used in the T-Test feature selection.

**eliminate colinear features *(boolean)*** This parameter indicates if the algorithm should try to delete collinear features during the regression or not.

**min tolerance *(real)*** This parameter is only available when the *eliminate colinear features* parameter is set to true. It specifies the minimum tolerance for eliminating collinear features.

**use bias *(boolean)*** This parameter indicates if an intercept value should be calculated or not.

**ridge *(real)*** This parameter specifies the ridge parameter for using in ridge regression.

## Related Documents

- **Polynomial Regression** (page 488)

## Tutorial Processes

### Applying the Linear Regression operator on the Polynomial data set

The 'Polynomial' data set is loaded using the Retrieve operator. The Filter Example Range operator is applied on it. The first example parameter of the Filter Example Range parameter is set to 1 and the last example parameter is set to 100. Thus the first 100 examples of the 'Polynomial' data set are selected. The Linear Regression operator is applied on it with default values of all parameters. The regression model generated by the Linear Regression operator is applied on the last 100 examples of the 'Polynomial' data set using the Apply Model operator. Labeled data from the Apply Model operator is provided to the Performance (Regression) operator. The absolute error and the prediction average parameters are set to true. Thus the Performance Vector generated by the Performance (Regression) operator has information regarding the absolute error and the prediction average in the labeled data set. The absolute error is calculated by adding the difference of all predicted values from the actual values of the label attribute, and dividing this sum by the total number of predictions. The prediction average is calculated by adding all

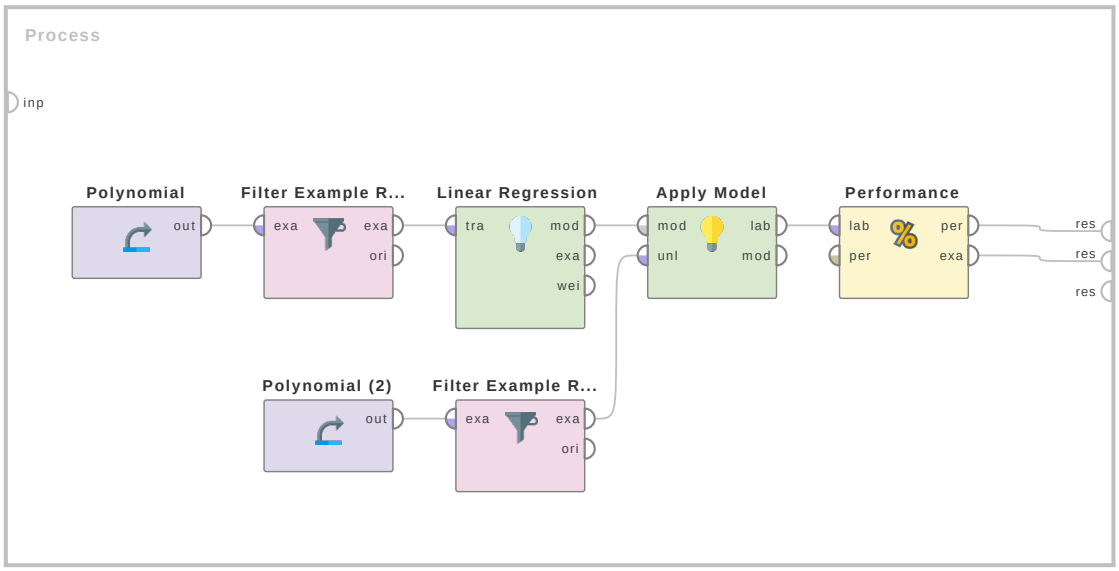
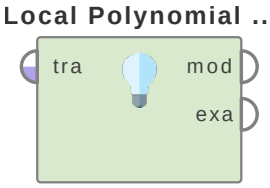



Figure 4.34: Tutorial process 'Applying the Linear Regression operator on the Polynomial data set'.

actual label values and dividing this sum by the total number of examples. You can verify this from the results in the Results Workspace.

# Local Polynomial Regression




This operator generates a local polynomial regression model from the given ExampleSet. Regression is a technique used for numerical prediction.

## Description

The Local Polynomial Regression operator provides functionality to perform a local regression. This means that if the label value for a point in the data space is requested, the local neighborhood of this point is searched. For this search the distance measure specified in the *numerical measure* parameter is used. After the neighborhood has been determined, its data points are used for fitting a polynomial of the specified degree using the weighted least squares optimization. The value of this polynomial at the requested point in the data space is then returned as result. During the fitting of the polynomial, the neighborhoods data points are weighted by their distance to the requested point. Here again the distance function specified in the parameters is used. The weight is calculated from the distance using the kernel smoother, specified by the *smoothing kernel* parameter. The resulting weight is then included into the least squares optimization. If the training ExampleSet contains a weight attribute, the distance based weight is multiplied by the example's weight. If the *use robust estimation* parameter is set to true, a Generate Weight (LPR) is performed with the same parameters as the following Local Polynomial Regression. For different settings the Generate Weight (LPR) operator might be used as a preprocessing step instead of using this parameter. As a result the outliers will be down-weighted so that the least squares fitting will not be affected by them anymore.

Since this is a local method, the computational need for training is minimal. In fact, each example is only stored in a way which provides a fast neighborhood search during the application time. Since all calculations are performed during the application time, it is slower than for example SVM, Linear Regression or Naive Bayes. In fact it really depends on the number of training examples and the number of attributes. If a higher degree than 1 is used, the calculations take much longer, because implicitly the polynomial expansion must be calculated.

Regression is a technique used for numerical prediction. It is a statistical measure that attempts to determine the strength of the relationship between one dependent variable ( i.e. the label attribute) and a series of other changing variables known as independent variables (regular attributes). Just like Classification is used for predicting categorical labels, Regression is used for predicting a continuous value. For example, we may wish to predict the salary of university graduates with 5 years of work experience, or the potential sales of a new product given its price. Regression is often used to determine how much specific factors such as the price of a commodity, interest rates, particular industries or sectors influence the price movement of an asset.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before application of this operator.

## Output Ports

**model *(mod)*** The regression model is delivered from this output port. This model can now be applied on unseen data sets.

**example set *(exa)*** The ExampleSet that was given as input is passed without any modifications to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**degree *(integer)*** This parameter specifies the degree of the local fitted polynomial. Please keep in mind, that a degree higher than 2 will increase calculation time extremely and probably suffer from overfitting.

**ridge factor *(real)*** This parameter specifies the ridge factor. This factor is used to penalize high coefficients. In order to avoid overfitting, the ridge factor might be increased.

**use robust estimation *(boolean)*** If this parameter is set to true, a re-weighting of the examples is performed in order to down-weight outliers.

**use weights *(boolean)*** This parameter indicates if example weights should be used if present in the given example set.

**iterations *(integer)*** This parameter is only available when the *use robust estimation* parameter is set to true. This parameter specifies the number of iterations performed for weight calculation.

**numerical measure *(selection)*** This parameter specifies the numerical measure for distance calculation.

**neighborhood type *(selection)*** This parameter determines which type of neighborhood should be used.

**k *(integer)*** This parameter is only available when the *neighborhood type* parameter is set to ‘Fixed Number ‘. It specifies the number of neighbors in the neighborhood. Regardless of the local density, always *k* samples are returned.

**fixed distance *(real)*** This parameter is only available when the *neighborhood type* parameter is set to ‘Fixed Distance ‘. It specifies the size of the neighborhood. All points within this distance are added.

**relative size *(real)*** This parameter is only available when the *neighborhood type* parameter is set to ‘Relative Number ‘. It specifies the size of the neighborhood relative to the total number of examples. For example, a value of 0.04 would include 4% of the data points into the neighborhood.

**distance *(real)*** This parameter is only available when the *neighborhood type* parameter is set to ‘Distance but at least’. It specifies the size of the neighborhood. All points within this distance are added.

**at least *(integer)*** This parameter is only available when the *neighborhood type* parameter is set to ‘Distance but at least’. If the neighborhood count is less than this number, the distance is increased until this number is met.

**smoothing kernel *(selection)*** This parameter determines which kernel type should be used to calculate the weights of distant examples.

## Tutorial Processes

### Applying the Local Polynomial Regression operator on the Polynomial data set

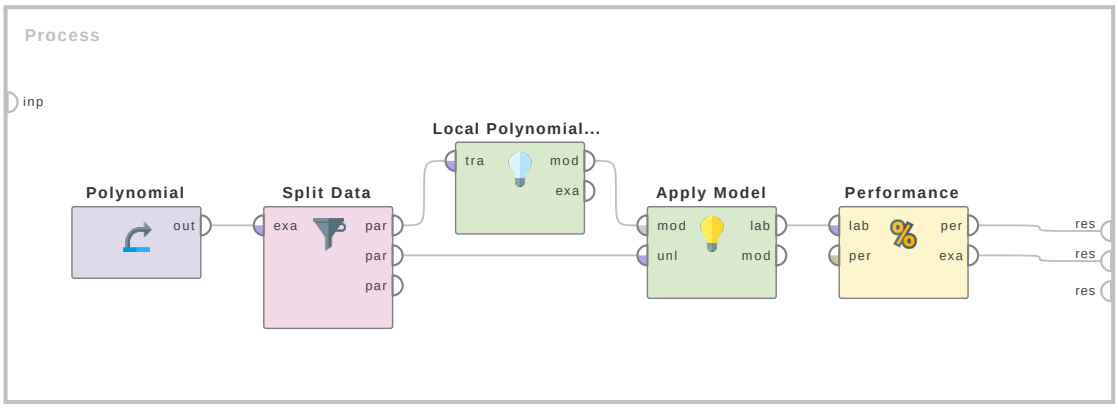
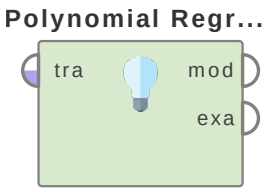



Figure 4.35: Tutorial process 'Applying the Local Polynomial Regression operator on the Polynomial data set'.

The 'Polynomial' data set is loaded using the Retrieve operator. The Split Data operator is applied on it to split the ExampleSet into training and testing data sets. The Local Polynomial Regression operator is applied on the training data set. The degree parameter is set to 3, the neighborhood type parameter is set to 'relative number' and the relative size is set to 0.5. The regression model generated by the Local Polynomial Regression operator is applied on the testing data set of the 'Polynomial' data set using the Apply Model operator. The labeled data set generated by the Apply Model operator is provided to the Performance (Regression) operator. The absolute error and the prediction average parameters are set to true. Thus the Performance Vector generated by the Performance (Regression) operator has information regarding the absolute error and the prediction average in the labeled data set. The absolute error is calculated by adding the difference of all predicted values from the actual values of the label attribute, and dividing this sum by the total number of predictions. The prediction average is calculated by adding all actual label values and dividing this sum by the total number of examples. You can verify this from the results in the Results Workspace.

# Polynomial Regression




This operator generates a polynomial regression model from the given ExampleSet. Polynomial regression is considered to be a special case of multiple linear regression.

## Description

Polynomial regression is a form of linear regression in which the relationship between the independent variable *x* and the dependent variable *y* is modeled as an *nth* order polynomial. In RapidMiner, *y* is the label attribute and *x* is the set of regular attributes that are used for the prediction of *y*. Polynomial regression fits a nonlinear relationship between the value of *x* and the corresponding conditional mean of *y*, denoted *E(y | x)*, and has been used to describe nonlinear phenomena such as the growth rate of tissues and the progression of disease epidemics. Although polynomial regression fits a nonlinear model to the data, as a statistical estimation problem it is linear, in the sense that the regression function *E(y | x)* is linear in the unknown parameters that are estimated from the data. For this reason, polynomial regression is considered to be a special case of multiple linear regression.

The goal of regression analysis is to model the expected value of a dependent variable *y* in terms of the value of an independent variable (or vector of independent variables) *x*. In simple linear regression, the following model is used:

*y = w0 + ( w1 * x )*

In this model, for each unit increase in the value of *x*, the conditional expectation of *y* increases by *w1* units.

In many settings, such a linear relationship may not hold. For example, if we are modeling the yield of a chemical synthesis in terms of the temperature at which the synthesis takes place, we may find that the yield improves by increasing amounts for each unit increase in temperature. In this case, we might propose a quadratic model of the form:

*y = w0 + (w1 * x1 ^1) + (w2 * x2 ^2)*

In this model, when the temperature is increased from *x* to *x + 1* units, the expected yield changes by *w1 + w2 + 2 (w2 * x)*. The fact that the change in yield depends on *x* is what makes the relationship nonlinear (this must not be confused with saying that this is nonlinear regression; on the contrary, this is still a case of linear regression). In general, we can model the expected value of *y* as an *nth* order polynomial, yielding the general polynomial regression model:

*y = w0 + (w1 * x1 ^1) + (w2 * x2 ^2) + . . . + (wm * xm ^m)*

Regression is a technique used for numerical prediction. It is a statistical measure that attempts to determine the strength of the relationship between one dependent variable ( i.e. the label attribute) and a series of other changing variables known as independent variables (regular attributes). Just like Classification is used for predicting categorical labels, Regression is used for predicting a continuous value. For example, we may wish to predict the salary of university graduates with 5 years of work experience, or the potential sales of a new product given its price. Regression is often used to determine how much specific factors such as the price of a commodity, interest rates, particular industries or sectors influence the price movement of an asset.

## Differentiation

- **Linear Regression** Polynomial regression is a form of linear regression in which the relationship between the independent variable *x* and the dependent variable *y* is modeled as an *nth* order polynomial. See page 482 for details.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before application of this operator.

## Output Ports

**model *(mod)*** The regression model is delivered from this output port. This model can now be applied on unseen data sets.

**example set *(exa)*** The ExampleSet that was given as input is passed without any modifications to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**max iterations *(integer)*** This parameter specifies the maximum number of iterations to be used for the model fitting.

**replication factor *(integer)*** This parameter specifies the amount of times each input variable is replicated, i.e. how many different degrees and coefficients can be applied to each variable.

**max degree *(integer)*** This parameter specifies the maximal degree to be used for the final polynomial.

**min coefficient *(real)*** This parameter specifies the minimum number to be used for the coefficients and the offset.

**max coefficient *(real)*** This parameter specifies the maximum number to be used for the coefficients and the offset.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of the *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Related Documents

- **Linear Regression** (page 482)

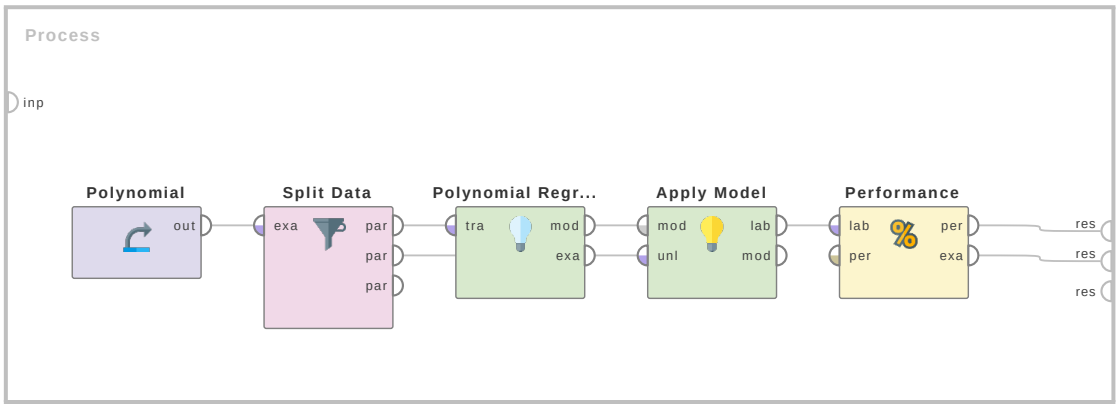


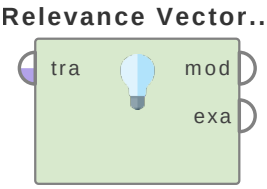

Figure 4.36: Tutorial process 'Applying the Polynomial Regression operator on the Polynomial data set'.

## Tutorial Processes

### Applying the Polynomial Regression operator on the Polynomial data set

The 'Polynomial' data set is loaded using the Retrieve operator. The Split Data operator is applied on it to split the ExampleSet into training and testing data sets. The Polynomial Regression operator is applied on the training data set with default values of all parameters. The regression model generated by the Polynomial Regression operator is applied on the testing data set of the 'Polynomial' data set using the Apply Model operator. The labeled data set generated by the Apply Model operator is provided to the Performance (Regression) operator. The absolute error and the prediction average parameters are set to true. Thus the Performance Vector generated by the Performance (Regression) operator has information regarding the absolute error and the prediction average in the labeled data set. The absolute error is calculated by adding the difference of all predicted values from the actual values of the label attribute, and dividing this sum by the total number of predictions. The prediction average is calculated by adding all actual label values and dividing this sum by the total number of examples. You can verify this from the results in the Results Workspace.

# Relevance Vector Machine




This operator is an implementation of Relevance Vector Machine (RVM) which is a probabilistic method both for classification and regression.

## Description

The Relevance Vector Machine operator is a probabilistic method both for classification and regression. The implementation of the relevance vector machine is based on the original algorithm described by 'Tipping/2001'. The fast version of the marginal likelihood maximization ('Tipping/Faul/2003') is also available if the *rvm type* parameter is set to 'Constructive-Regression-RVM'.

A Relevance Vector Machine (RVM) is a machine learning technique that uses Bayesian inference to obtain parsimonious solutions for regression and classification. The RVM has an identical functional form to the support vector machine, but provides probabilistic classification. It is actually equivalent to a Gaussian process model with a certain covariance function. Compared to that of support vector machines (SVM), the Bayesian formulation of the RVM avoids the set of free parameters of the SVM (that usually require cross-validation-based post-optimizations). However RVMs use an expectation maximization (EM)-like learning method and are therefore at risk of local minima. This is unlike the standard sequential minimal optimization(SMO)-based algorithms employed by SVMs, which are guaranteed to find a global optimum.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before the application of this operator.

## Output Ports

**model *(mod)*** The RVM is applied and the resultant model is delivered from this output port. This model can now be applied on unseen data sets.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**rvm type *(selection)*** This parameter specifies the type of RVM Regression. The following options are available: Regression-RVM, Classification-RVM and Constructive-Regression-RVM.

**kernel type *(selection)*** The type of the kernel function is selected through this parameter. Following kernel types are supported: *rbf, cauchy, laplace, poly, sigmoid, Epanechnikov, gaussian combination, multiquadric*

**kernel lengthscale *(real)*** This parameter specifies the lengthscale to be used in all kernels.

**kernel degree *(real)*** This is the kernel parameter degree. This is only available when the *kernel type* parameter is set to *polynomial* or *epachnenikov*.

**kernel bias *(real)*** This parameter specifies the bias to be used in the poly kernel.

**kernel sigma1 *(real)*** This is the kernel parameter sigma1. This is only available when the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2 *(real)*** This is the kernel parameter sigma2. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3 *(real)*** This is the kernel parameter sigma3. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel shift *(real)*** This is the kernel parameter shift. This is only available when the *kernel type* parameter is set to *multiquadric*.

**kernel a *(real)*** This is the kernel parameter a. This is only available when the *kernel type* parameter is set to *sigmoid*

**kernel b *(real)*** This is the kernel parameter b. This is only available when the *kernel type* parameter is set to *sigmoid*

**max iteration *(integer)*** This parameter specifies the maximum number of iterations to be used.

**min delta log alpha *(real)*** The iteration is aborted if the largest log alpha change is smaller than *min delta log alpha*.

**alpha max *(real)*** The basis function is pruned if its alpha is larger than the *alpha max*.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Introduction to the RVM operator

The 'Polynomial' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a regression model. The Relevance Vector Machine operator is applied in the training subprocess of the Split Validation operator. All parameters are used with default values. The Relevance Vector Machine operator generates a regression model. The Apply Model operator is used in the testing subprocess to apply this model on the testing data set. The resultant labeled ExampleSet is used by the Performance operator for measuring the performance of the model. The regression model and its performance vector are connected to the output and it can be seen in the Results Workspace.

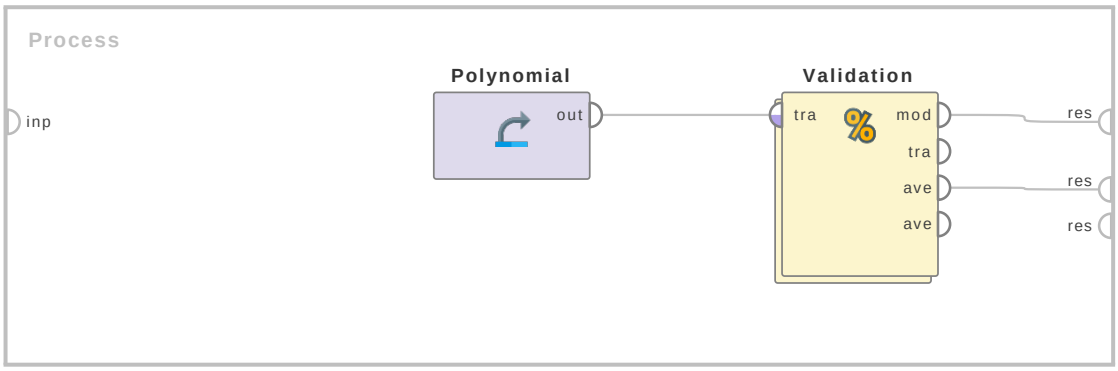
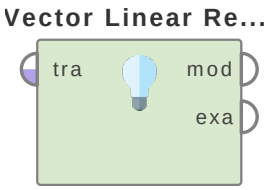



Figure 4.37: Tutorial process 'Introduction to the RVM operator'.

# Vector Linear Regression




This operator calculates a vector linear regression model from the input ExampleSet.

## Description

Regression is a technique used for numerical prediction. Regression is a statistical measure that attempts to determine the strength of the relationship between one dependent variable ( i.e. the label attribute) and a series of other changing variables known as independent variables (regular attributes). Just like Classification is used for predicting categorical labels, Regression is used for predicting a continuous value. For example, we may wish to predict the salary of university graduates with 5 years of work experience, or the potential sales of a new product given its price. Regression is often used to determine how much specific factors such as the price of a commodity, interest rates, particular industries or sectors influence the price movement of an asset.

Linear regression attempts to model the relationship between a scalar variable and one or more explanatory variables by fitting a linear equation to observed data. For example, one might want to relate the weights of individuals to their heights using a linear regression model.

This operator performs a vector linear regression. It regresses all regular attributes upon a vector of labels. The attributes forming the vector should be marked as special, the special role names of all label attributes should start with 'label'.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before application of this operator.

## Output Ports

**model *(mod)*** The regression model is delivered from this output port. This model can now be applied on unseen data sets.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**use bias *(boolean)*** This parameter indicates if an intercept value should be calculated or not.

**ridge *(real)*** This parameter specifies the ridge parameter for using in ridge regression.

## Tutorial Processes

### Applying the Vector Linear Regression operator on the Polynomial data set

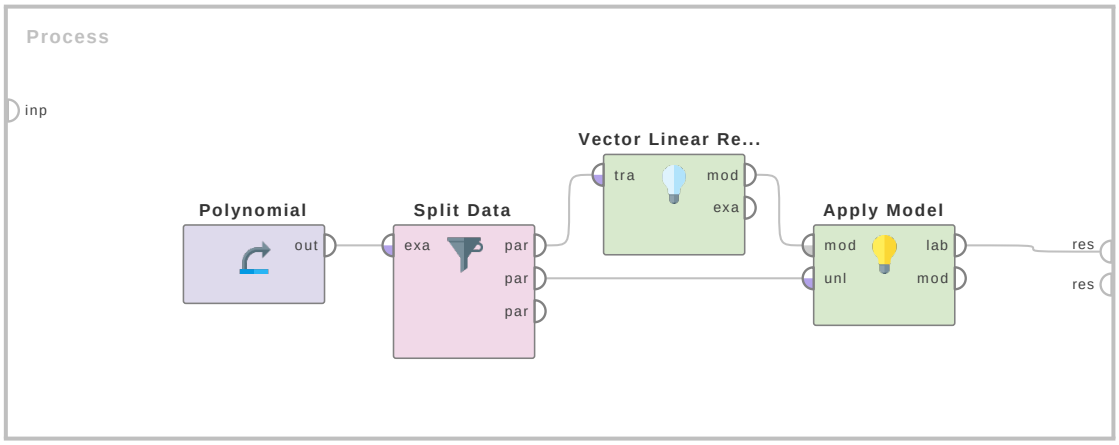
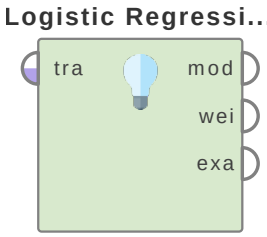



Figure 4.38: Tutorial process 'Applying the Vector Linear Regression operator on the Polynomial data set'.

The 'Polynomial' data set is loaded using the Retrieve operator. The Split Data operator is applied for splitting the ExampleSet into training and testing data sets. The Vector Linear Regression operator is applied on the training data set with default values of all parameters. The regression model generated by the Vector Linear Regression operator is applied on the testing data set of the 'Polynomial' data set using the Apply Model operator. The resultant labeled data from the Apply Model operator can be seen in the Results Workspace.

# 4.1.7 Logistic Regression

## Logistic Regression (SVM)




This operator is a Logistic Regression Learner. It is based on the internal Java implementation of the *myKLR* by Stefan Rueping.

### Description

This learner uses the Java implementation of the *myKLR* by Stefan Rueping. *myKLR* is a tool for large scale kernel logistic regression based on the algorithm of Keerthi etal (2003) and the code of *mySVM*. For compatibility reasons, the model of *myKLR* differs slightly from that of Keerthi etal (2003). As *myKLR* is based on the code of *mySVM*; the format of example files, parameter files and kernel definition are identical. Please see the documentation of the SVM operator for further information. This learning method can be used for both regression and classification and provides a fast algorithm and good results for many learning tasks. *mySVM* works with linear or quadratic and even asymmetric loss functions.

This operator supports various kernel types including *dot, radial, polynomial, neural, anova, epachnenikov, gaussian combination* and *multiquadric*. Explanation of these kernel types is given in the parameters section.

### Input Ports

**training set *(tra)*** This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before application of this operator.

### Output Ports

**model *(mod)*** The Logistic Regression model is delivered from this output port. This model can now be applied on unseen data sets.

**weights *(wei)*** This port delivers the attribute weights. This is only possible when the dot kernel type is used, it is not possible with other kernel types.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

### Parameters

**kernel type *(selection)*** The type of the kernel function is selected through this parameter. Following kernel types are supported: *dot, radial, polynomial, neural, anova, epachnenikov, gaussian combination, multiquadric*

- **dot** The dot kernel is defined by$k(x,y)=x*y$ i.e. it is inner product of$x$ and $y$.

- **radial** The radial kernel is defined by *exp(-g ||x-y||^2)* where *g* is the *gamma*, it is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by *k(x,y)=(x*y+1)^d* where *d* is the degree of polynomial and it is specified by the *kernel degree* parameter. The polynomial kernels are well suited for problems where all the training data is normalized.
- **neural** The neural kernel is defined by a two layered neural net *tanh(a x*y+b)* where *a* is *alpha* and *b* is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is 1/N, where N is the data dimension. Note that not all choices of *a* and *b* lead to a valid kernel function.
- **anova** The anova kernel is defined by raised to power *d* of summation of *exp(-g (x-y))* where *g* is *gamma* and *d* is *degree*. gamma and degree are adjusted by the *kernel gamma* and *kernel degree* parameters respectively.
- **epachnenikov** The epachnenikov kernel is this function *(3/4)(1-u2)* for *u* between -1 and 1 and zero for *u* outside that range. It has two adjustable parameters *kernel sigma1* and *kernel degree*.
- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters *kernel sigma1, kernel sigma2* and *kernel sigma3*.
- **multiquadric** The multiquadric kernel is defined by the square root of *||x-y||^2 + c^2*. It has adjustable parameters *kernel sigma1* and *kernel sigma shift*.

**kernel gamma *(real)*** This is the SVM kernel parameter gamma. This is only available when the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1 *(real)*** This is the SVM kernel parameter sigma1. This is only available when the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2 *(real)*** This is the SVM kernel parameter sigma2. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3 *(real)*** This is the SVM kernel parameter sigma3. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel shift *(real)*** This is the SVM kernel parameter shift. This is only available when the *kernel type* parameter is set to *multiquadric*.

**kernel degree *(real)*** This is the SVM kernel parameter degree. This is only available when the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a *(real)*** This is the SVM kernel parameter a. This is only available when the *kernel type* parameter is set to *neural*.

**kernel b *(real)*** This is the SVM kernel parameter b. This is only available when the *kernel type* parameter is set to *neural*.

**kernel cache *(real)*** This is an expert parameter. It specifies the size of the cache for kernel evaluations in megabytes.

**C *(real)*** This is the SVM complexity constant which sets the tolerance for misclassification, where higher C values allow for 'softer' boundaries and lower values create 'harder' boundaries. A complexity constant that is too large can lead to over-fitting, while values that are too small may result in over-generalization.

**convergence epsilon** This is an optimizer parameter. It specifies the precision on the KKT conditions.

**max iterations *(integer)*** This is an optimizer parameter. It specifies to stop iterations after a specified number of iterations.

**scale *(boolean)*** This is a global parameter. If checked, the example values are scaled and the scaling parameters are stored for a test set.

## Tutorial Processes

### Introduction to the Logistic Regression operator

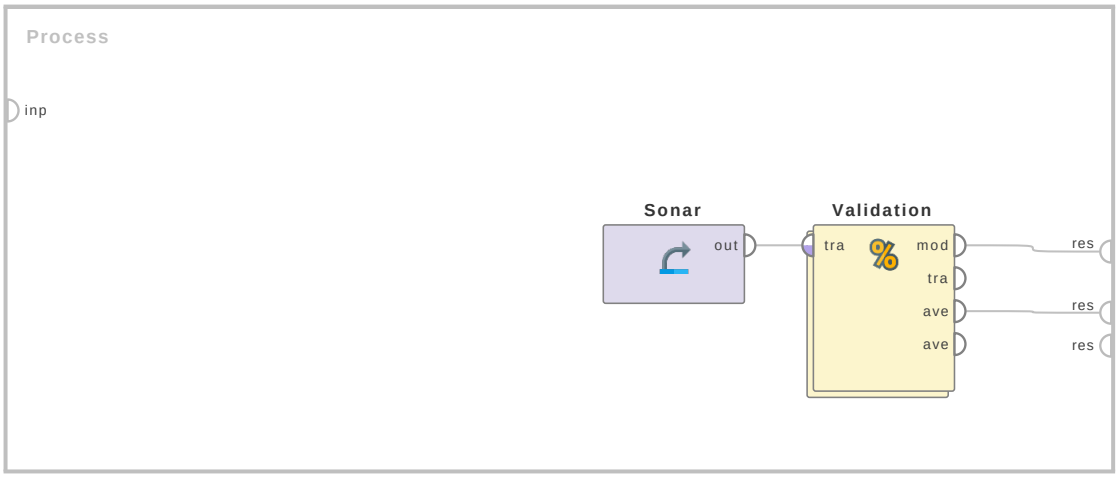
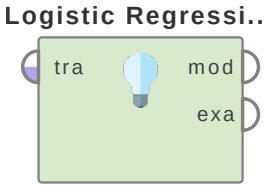



Figure 4.39: Tutorial process 'Introduction to the Logistic Regression operator'.

The 'Sonar' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a regression model. The Logistic Regression operator is applied in the training subprocess of the Split Validation operator. All parameters are used with default values. The Logistic Regression operator generates a regression model. The Apply Model operator is used in the testing subprocess to apply this model on the testing data set. The resultant labeled ExampleSet is used by the Performance operator for measuring the performance of the model. The regression model and its performance vector are connected to the output and it can be seen in the Results Workspace.

# Logistic Regression (Evolutionary)




This operator is a kernel logistic regression learner for binary classification tasks.

## Description

Logistic regression is a type of regression analysis used for predicting the outcome of a categorical (a variable that can take on a limited number of categories) criterion variable based on one or more predictor variables. The probabilities describing the possible outcome of a single trial are modeled, as a function of explanatory variables, using a logistic function. Logistic regression measures the relationship between a categorical dependent variable and usually a continuous independent variable (or several), by converting the dependent variable to probability scores

This operator supports various kernel types including *dot, radial, polynomial, sigmoid, anova, epachnenikov, gaussian combination* and *multiquadric*. An explanation of these kernel types is given in the parameters section.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before application of this operator.

## Output Ports

**model *(mod)*** The Logistic Regression model is delivered from this output port. This model can now be applied on unseen data sets.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**kernel type *(selection)*** The type of the kernel function is selected through this parameter. Following kernel types are supported: *dot, radial, polynomial, sigmoid, anova, epachnenikov, gaussian combination, multiquadric*

- **dot** The dot kernel is defined by*k(x,y)=x*y* i.e. it is inner product of*x* and *y*.
- **radial** The radial kernel is defined by *exp(-g ||x-y||^2)* where *g* is the *gamma*, it is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by *k(x,y)=(x*y+1)^d* where *d* is the degree of polynomial and it is specified by the *kernel degree* parameter. The polynomial kernels are well suited for problems where all the training data is normalized.

- **sigmoid** The sigmoid kernel is defined by a two layered neural net *tanh(a x*y+b)* where *a* is *alpha* and *b* is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is 1/N, where N is the data dimension. Note that not all choices of *a* and *b* lead to a valid kernel function.
- **anova** The anova kernel is defined by raised to power *d* of summation of *exp(-g (x-y))* where *g* is *gamma* and *d* is *degree*. gamma and degree are adjusted by the *kernel gamma* and *kernel degree* parameters respectively.
- **epachnenikov** The epachnenikov kernel is this function *(3/4)(1-u2)* for *u* between -1 and 1 and zero for *u* outside that range. It has two adjustable parameters *kernel sigma1* and *kernel degree*.
- **gaussian_combination** This is the gaussian combination kernel. It has the adjustable parameters *kernel sigma1, kernel sigma2* and *kernel sigma3*.
- **multiquadric** The multiquadric kernel is defined by the square root of *||x-y||^2 + c^2*. It has the adjustable parameters *kernel sigma1* and *kernel sigma shift*.

**kernel gamma *(real)*** This is the kernel parameter gamma. This is only available when the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1 *(real)*** This is the kernel parameter sigma1. This is only available when the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2 *(real)*** This is the kernel parameter sigma2. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3 *(real)*** This is the kernel parameter sigma3. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel shift *(real)*** This is the kernel parameter shift. This is only available when the *kernel type* parameter is set to *multiquadric*.

**kernel degree *(real)*** This is the kernel parameter degree. This is only available when the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a *(real)*** This is the kernel parameter a. This is only available when the *kernel type* parameter is set to *sigmoid*

**kernel b *(real)*** This is the kernel parameter b. This is only available when the *kernel type* parameter is set to *sigmoid*

**C *(real)*** This is the complexity constant which sets the tolerance for misclassification, where higher C values allow for 'softer' boundaries and lower values create 'harder' boundaries. A complexity constant that is too large can lead to over-fitting, while values that are too small may result in over-generalization.

**start population type *(selection)*** This parameter specifies the type of start population initialization.

**max generations *(integer)*** This parameter specifies the number of generations after which the algorithm should be terminated.

**generations without improval *(integer)*** This parameter specifies the stop criterion for early stopping i.e. it stops after *n* generations without improvement in the performance. *n* is specified by this parameter.

**population size** ***(integer)*** This parameter specifies the population size i.e. the number of individuals per generation. If set to -1, all examples are selected.

**tournament fraction** ***(real)*** This parameter specifies the fraction of the current population which should be used as tournament members.

**keep best** ***(boolean)*** This parameter specifies if the best individual should survive. This is also called elitist selection. Retaining the best individuals in a generation unchanged in the next generation, is called elitism or elitist selection.

**mutation type** ***(selection)*** This parameter specifies the type of the mutation operator.

**selection type** ***(selection)*** This parameter specifies the selection scheme of this evolutionary algorithms.

**crossover prob** ***(real)*** The probability for an individual to be selected for crossover is specified by this parameter.

**use local random seed** ***(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed** ***(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

**show convergence plot** ***(boolean)*** This parameter indicates if a dialog with a convergence plot should be drawn.

## Tutorial Processes

### Introduction to the Logistic Regression (Evolutionary) operator

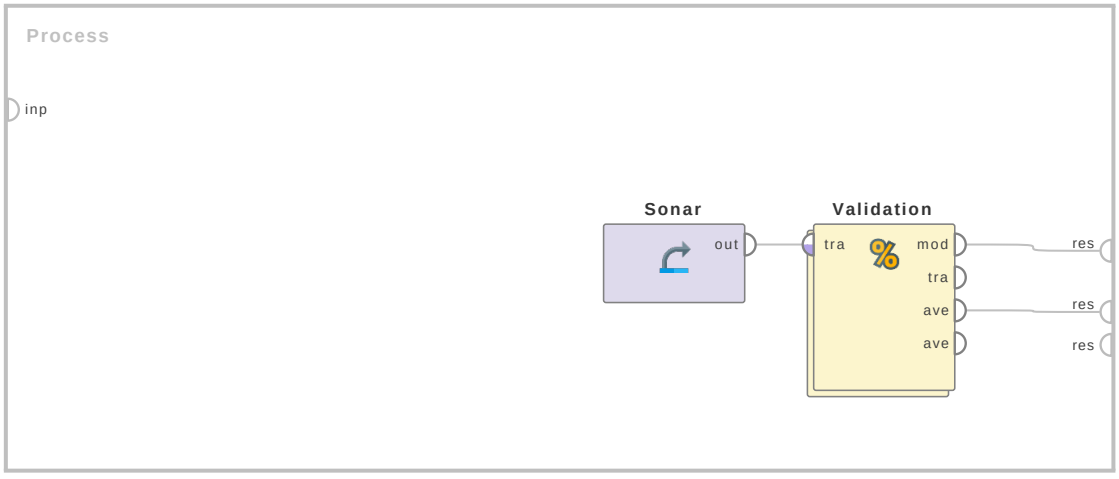
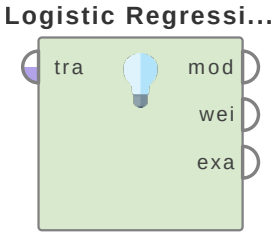



Figure 4.40: Tutorial process 'Introduction to the Logistic Regression (Evolutionary) operator'.

The 'Sonar' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a regression model. The Logistic Regression (Evolutionary)

operator is applied in the training subprocess of the Split Validation operator. All parameters are used with default values. The Logistic Regression (Evolutionary) operator generates a regression model. The Apply Model operator is used in the testing subprocess to apply this model on the testing data set. The resultant labeled ExampleSet is used by the Performance operator for measuring the performance of the model. The regression model and its performance vector are connected to the output and it can be seen in the Results Workspace.

# Logistic Regression (SVM)




This operator is a Logistic Regression Learner. It is based on the internal Java implementation of the *myKLR* by Stefan Rueping.

## Description

This learner uses the Java implementation of the *myKLR* by Stefan Rueping. *myKLR* is a tool for large scale kernel logistic regression based on the algorithm of Keerthi etal (2003) and the code of *mySVM*. For compatibility reasons, the model of *myKLR* differs slightly from that of Keerthi etal (2003). As *myKLR* is based on the code of *mySVM*; the format of example files, parameter files and kernel definition are identical. Please see the documentation of the SVM operator for further information. This learning method can be used for both regression and classification and provides a fast algorithm and good results for many learning tasks. *mySVM* works with linear or quadratic and even asymmetric loss functions.

This operator supports various kernel types including *dot, radial, polynomial, neural, anova, epachnenikov, gaussian combination* and *multiquadric*. Explanation of these kernel types is given in the parameters section.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before application of this operator.

## Output Ports

**model *(mod)*** The Logistic Regression model is delivered from this output port. This model can now be applied on unseen data sets.

**weights *(wei)*** This port delivers the attribute weights. This is only possible when the dot kernel type is used, it is not possible with other kernel types.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**kernel type *(selection)*** The type of the kernel function is selected through this parameter. Following kernel types are supported: *dot, radial, polynomial, neural, anova, epachnenikov, gaussian combination, multiquadric*

- **dot** The dot kernel is defined by$k(x,y)=x*y$ i.e. it is inner product of$x$ and $y$.

- **radial** The radial kernel is defined by *exp(-g ||x-y||^2)* where *g* is the *gamma*, it is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by *k(x,y)=(x*y+1)^d* where *d* is the degree of polynomial and it is specified by the *kernel degree* parameter. The polynomial kernels are well suited for problems where all the training data is normalized.
- **neural** The neural kernel is defined by a two layered neural net *tanh(a x*y+b)* where *a* is *alpha* and *b* is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is 1/N, where N is the data dimension. Note that not all choices of *a* and *b* lead to a valid kernel function.
- **anova** The anova kernel is defined by raised to power *d* of summation of *exp(-g (x-y))* where *g* is *gamma* and *d* is *degree*. gamma and degree are adjusted by the *kernel gamma* and *kernel degree* parameters respectively.
- **epachnenikov** The epachnenikov kernel is this function *(3/4)(1-u2)* for *u* between -1 and 1 and zero for *u* outside that range. It has two adjustable parameters *kernel sigma1* and *kernel degree*.
- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters *kernel sigma1, kernel sigma2* and *kernel sigma3*.
- **multiquadric** The multiquadric kernel is defined by the square root of *||x-y||^2 + c^2*. It has adjustable parameters *kernel sigma1* and *kernel sigma shift*.

**kernel gamma *(real)*** This is the SVM kernel parameter gamma. This is only available when the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1 *(real)*** This is the SVM kernel parameter sigma1. This is only available when the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2 *(real)*** This is the SVM kernel parameter sigma2. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3 *(real)*** This is the SVM kernel parameter sigma3. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel shift *(real)*** This is the SVM kernel parameter shift. This is only available when the *kernel type* parameter is set to *multiquadric*.

**kernel degree *(real)*** This is the SVM kernel parameter degree. This is only available when the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a *(real)*** This is the SVM kernel parameter a. This is only available when the *kernel type* parameter is set to *neural*.

**kernel b *(real)*** This is the SVM kernel parameter b. This is only available when the *kernel type* parameter is set to *neural*.

**kernel cache *(real)*** This is an expert parameter. It specifies the size of the cache for kernel evaluations in megabytes.

**C *(real)*** This is the SVM complexity constant which sets the tolerance for misclassification, where higher C values allow for 'softer' boundaries and lower values create 'harder' boundaries. A complexity constant that is too large can lead to over-fitting, while values that are too small may result in over-generalization.

**convergence epsilon** This is an optimizer parameter. It specifies the precision on the KKT conditions.

**max iterations *(integer)*** This is an optimizer parameter. It specifies to stop iterations after a specified number of iterations.

**scale *(boolean)*** This is a global parameter. If checked, the example values are scaled and the scaling parameters are stored for a test set.

## Tutorial Processes

### Introduction to the Logistic Regression operator

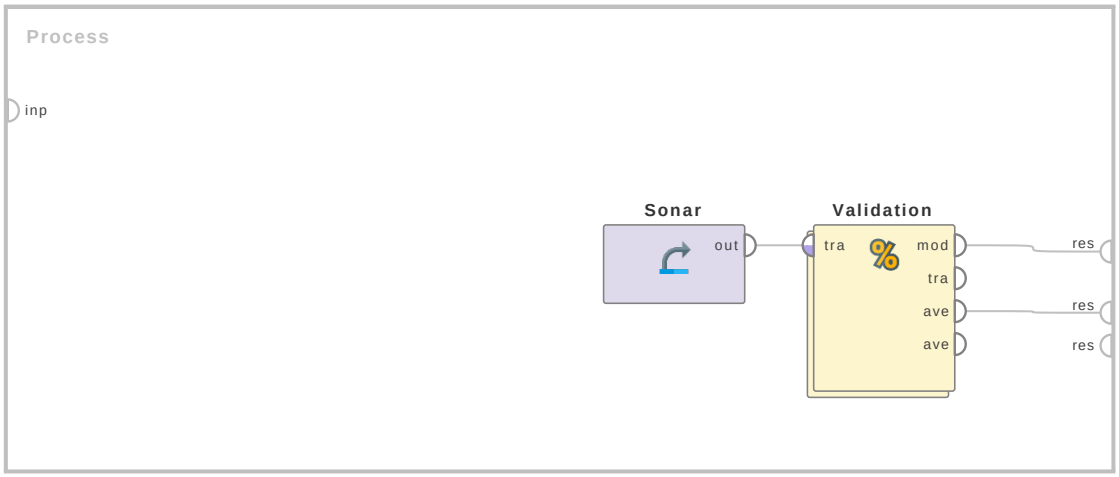
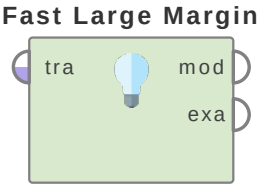



Figure 4.41: Tutorial process ‘Introduction to the Logistic Regression operator’.

The ‘Sonar’ data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a regression model. The Logistic Regression operator is applied in the training subprocess of the Split Validation operator. All parameters are used with default values. The Logistic Regression operator generates a regression model. The Apply Model operator is used in the testing subprocess to apply this model on the testing data set. The resultant labeled ExampleSet is used by the Performance operator for measuring the performance of the model. The regression model and its performance vector are connected to the output and it can be seen in the Results Workspace.

# 4.1.8 Support Vector Machines

## Fast Large Margin




This operator is a fast learning method for large margin optimizations.

### Description

The Fast Large Margin operator applies a fast margin learner based on the linear support vector learning scheme proposed by R.E. Fan, K.W. Chang, C.J. Hsieh, X.R. Wang, and C.J. Lin. Although the result is similar to those delivered by classical SVM or logistic regression implementations, this linear classifier is able to work on data set with millions of examples and attributes.

Here is a basic description of SVM. The standard SVM takes a set of input data and predicts, for each given input, which of two possible classes comprises the input, making the SVM a non-probabilistic binary linear classifier. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

More formally, a support vector machine constructs a hyperplane or set of hyperplanes in a high- or infinite- dimensional space, which can be used for classification, regression, or other tasks. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training data points of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier. Whereas the original problem may be stated in a finite dimensional space, it often happens that the sets to discriminate are not linearly separable in that space. For this reason, it was proposed that the original finite-dimensional space would be mapped into a much higher-dimensional space, presumably making the separation easier in that space. To keep the computational load reasonable, the mapping used by SVM schemes are designed to ensure that dot products may be computed easily in terms of the variables in the original space, by defining them in terms of a kernel function *K(x,y)* selected to suit the problem. The hyperplanes in the higher dimensional space are defined as the set of points whose inner product with a vector in that space is constant.

### Input Ports

**training set *(tra)*** This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before applying this operator.

### Output Ports

**model *(mod)*** The classification/regression model is delivered from this output port. This model can now be applied on unseen data sets.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**solver *(selection)*** This parameter specifies the solver type for this fast margin method. The following options are available: L2 SVM Dual, L2 SVM Primal, L2 Logistic Regression and L1 SVM Dual.

**C *(real)*** This parameter specifies the cost parameter C. It is the penalty parameter of the error term.

**epsilon *(real)*** This parameter specifies the tolerance of the termination criterion.

**class weights *(list)*** This is an expert parameter. It specifies the weights 'w' for all classes. The *Edit List* button opens a new window with two columns. The first column specifies the class name and the second column specifies the weight for that class. The parameter C is calculated as *weight* of the class multiplied by *C*. If the weight of a class is not specified, that class is assigned *weight = 1*.

**use bias *(boolean)*** This parameter indicates if an intercept value should be calculated.

## Tutorial Processes

### Introduction to the Fast Large Margin operator

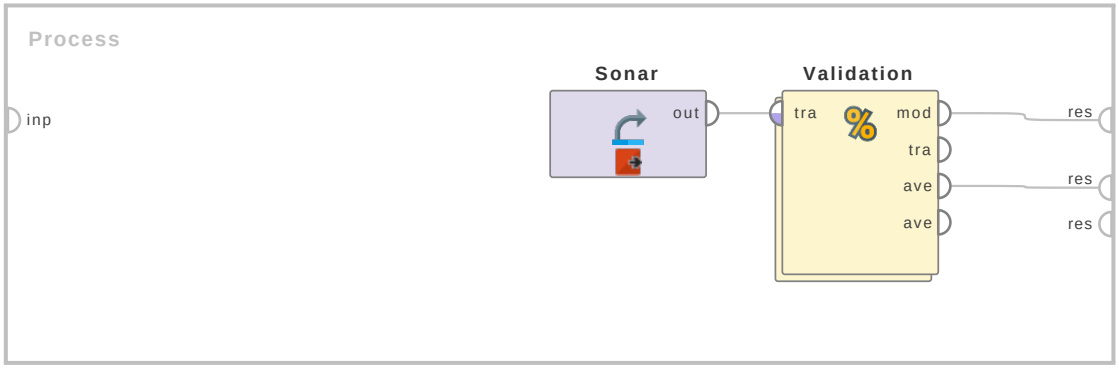
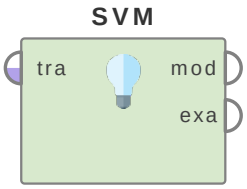



Figure 4.42: Tutorial process 'Introduction to the Fast Large Margin operator'.

The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. The Split Validation operator is applied on this ExampleSet to assist in training and testing a model. Have a look at the subprocesses of this operator. The Fast Large Margin operator is applied in the Training subprocess for training a model. The resultant model is applied in the Testing subprocess on the testing data set using the Apply Model operator. The performance of the operator is measured using the Performance (Classification) operator. The final model and its performance vector are connected to the output and they can be seen in the Results Workspace.

# Support Vector Machine




This operator is an SVM (Support Vector Machine) Learner. It is based on a minimal SVM implementation.

## Description

This learner uses a minimal SVM implementation. The model is built with only one positive and one negative example. Typically this operator is used in combination with a boosting method.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. This operator can only handle numerical attributes with only one positive and one negative example.

## Output Ports

**model *(mod)*** The Hyper model is delivered from this output port. This model can now be applied on unseen data sets.

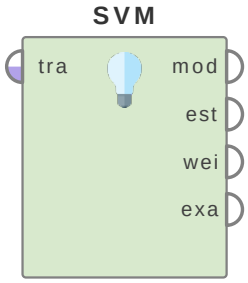
**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same ExampleSet. Changing the value of this parameter changes the way examples are randomized, thus the ExampleSet will have a different set of values.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

# Support Vector Machine




This operator is an SVM (Support Vector Machine) Learner. It is based on the internal Java implementation of the *mySVM* by Stefan Rueping.

## Description

This learner uses the Java implementation of the support vector machine *mySVM* by Stefan Rueping. This learning method can be used for both regression and classification and provides a fast algorithm and good results for many learning tasks. *mySVM* works with linear or quadratic and even asymmetric loss functions.

This operator supports various kernel types including *dot, radial, polynomial, neural, anova, epachnenikov, gaussian combination* and *multiquadric*. Explanation of these kernel types is given in the parameters section.

Here is a basic description of the SVM. The standard SVM takes a set of input data and predicts, for each given input, which of the two possible classes comprises the input, making the SVM a non-probabilistic binary linear classifier. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

More formally, a support vector machine constructs a hyperplane or set of hyperplanes in a high- or infinite- dimensional space, which can be used for classification, regression, or other tasks. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training data points of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier. Whereas the original problem may be stated in a finite dimensional space, it often happens that the sets to discriminate are not linearly separable in that space. For this reason, it was proposed that the original finite-dimensional space be mapped into a much higher-dimensional space, presumably making the separation easier in that space. To keep the computational load reasonable, the mapping used by the SVM schemes are designed to ensure that dot products may be computed easily in terms of the variables in the original space, by defining them in terms of a kernel function *K(x,y)* selected to suit the problem. The hyperplanes in the higher dimensional space are defined as the set of points whose inner product with a vector in that space is constant.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before application of this operator.

## Output Ports

**model *(mod)*** The SVM model is delivered from this output port. This model can now be applied on unseen data sets.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**estimated performance *(est)*** This port delivers a performance vector of the SVM model which gives an estimation of statistical performance of this model.

**weights *(wei)*** This port delivers the attribute weights. This is possible only when the dot kernel type is used, it is not possible with other kernel types.

## Parameters

**kernel type *(selection)*** The type of the kernel function is selected through this parameter. Following kernel types are supported: *dot, radial, polynomial, neural, anova, epachnenikov, gaussian combination, multiquadric*

- **dot** The dot kernel is defined by*k(x,y)=x*y* i.e. it is inner product of*x* and *y*.
- **radial** The radial kernel is defined by *exp(-g ||x-y||^2)* where *g* is the *gamma*, it is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by *k(x,y)=(x*y+1)^d* where *d* is the degree of polynomial and it is specified by the *kernel degree* parameter. The polynomial kernels are well suited for problems where all the training data is normalized.
- **neural** The neural kernel is defined by a two layered neural net *tanh(a x*y+b)* where *a* is *alpha* and *b* is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is 1/N, where N is the data dimension. Note that not all choices of *a* and *b* lead to a valid kernel function.
- **anova** The anova kernel is defined by raised to power *d* of summation of *exp(-g (x-y))* where *g* is *gamma* and *d* is *degree*. gamma and degree are adjusted by the *kernel gamma* and *kernel degree* parameters respectively.
- **epachnenikov** The epachnenikov kernel is this function *(3/4)(1-u2)* for *u* between -1 and 1 and zero for *u* outside that range. It has two adjustable parameters *kernel sigma1* and *kernel degree*.
- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters *kernel sigma1, kernel sigma2* and *kernel sigma3*.
- **multiquadric** The multiquadric kernel is defined by the square root of *||x-y||^2 + c^2*. It has adjustable parameters *kernel sigma1* and *kernel sigma shift*.

**kernel gamma *(real)*** This is the SVM kernel parameter gamma. This is available only when the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1 *(real)*** This is the SVM kernel parameter sigma1. This is available only when the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2 *(real)*** This is the SVM kernel parameter sigma2. This is available only when the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3** ***(real)*** This is the SVM kernel parameter sigma3. This is available only when the *kernel type* parameter is set to *gaussian combination*.

**kernel shift** ***(real)*** This is the SVM kernel parameter shift. This is available only when the *kernel type* parameter is set to *multiquadric*.

**kernel degree** ***(real)*** This is the SVM kernel parameter degree. This is available only when the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a** ***(real)*** This is the SVM kernel parameter a. This is available only when the *kernel type* parameter is set to *neural*.

**kernel b** ***(real)*** This is the SVM kernel parameter b. This is available only when the *kernel type* parameter is set to *neural*.

**kernel cache** ***(real)*** This is an expert parameter. It specifies the size of the cache for kernel evaluations in megabytes.

**C** ***(real)*** This is the SVM complexity constant which sets the tolerance for misclassification, where higher C values allow for 'softer' boundaries and lower values create 'harder' boundaries. A complexity constant that is too large can lead to over-fitting, while values that are too small may result in over-generalization.

**convergence epsilon** This is an optimizer parameter. It specifies the precision on the KKT conditions.

**max iterations** ***(integer)*** This is an optimizer parameter. It specifies to stop iterations after a specified number of iterations.

**scale** ***(boolean)*** This is a global parameter. If checked, the example values are scaled and the scaling parameters are stored for a test set.

**L pos** ***(real)*** A factor for the SVM complexity constant for positive examples. This parameter is part of the loss function.

**L neg** ***(real)*** A factor for the SVM complexity constant for negative examples.This parameter is part of the loss function.

**epsilon** ***(real)*** This parameter specifies the insensitivity constant. No loss if the prediction lies this close to true value. This parameter is part of the loss function.

**epsilon plus** ***(real)*** This parameter is part of the loss function. It specifies epsilon for positive deviation only.

**epsilon minus** ***(real)*** This parameter is part of the loss function. It specifies epsilon for negative deviation only.

**balance cost** ***(boolean)*** If checked, adapts Cpos and Cneg to the relative size of the classes.

**quadratic loss pos** ***(boolean)*** Use quadratic loss for positive deviation. This parameter is part of the loss function.

**quadratic loss neg** ***(boolean)*** Use quadratic loss for negative deviation. This parameter is part of the loss function.

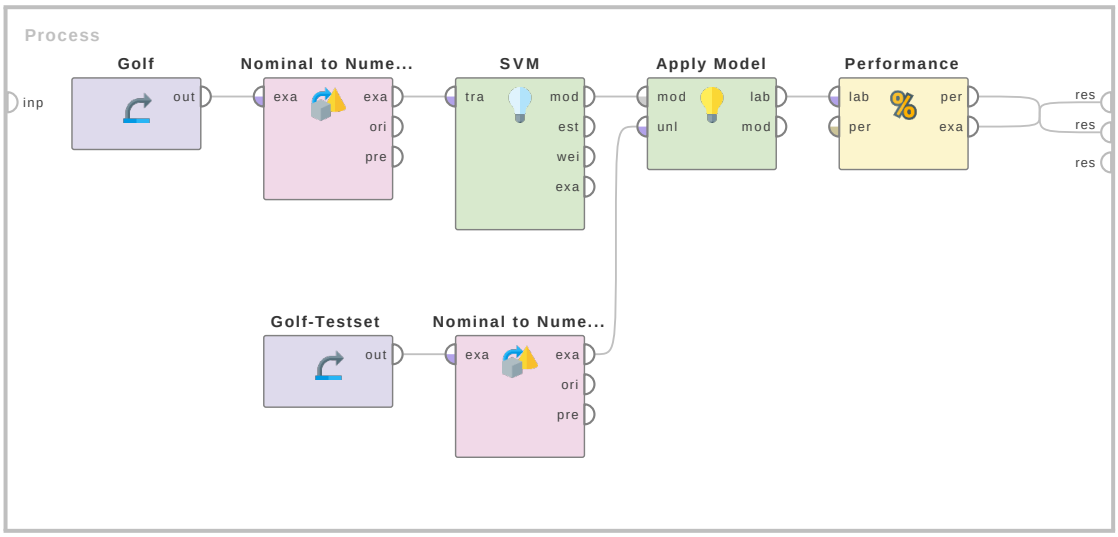



Figure 4.43: Tutorial process 'Getting started with SVM'.

## Tutorial Processes

### Getting started with SVM

This is a simple Example Process which gets you started with the SVM operator. The Retrieve operator is used to load the 'Golf' data set. The Nominal to Numerical operator is applied on it to convert its nominal attributes to numerical form. This step is necessary because the SVM operator cannot take nominal attributes, it can only classify using numerical attributes. The model generated from the SVM operator is then applied on the 'Golf-Testset' data set. Nominal to Numerical operator was applied on this data set as well. This is necessary because the testing and training data set should be in the same format. The statistical performance of this model is measured using the Performance operator. This is a very basic process. It is recommended that you develop a deeper understanding of SVM for getting better results through this operator. The support vector machine (SVM) is a popular classification technique. However, beginners who are not familiar with SVM often get unsatisfactory results since they miss some easy but significant steps.

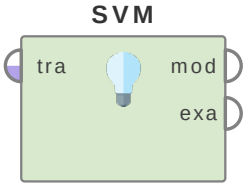
Using 'm' numbers to represent an m-category attribute is recommended. Only one of the 'm' numbers is 1, the others are 0. For example, a three-category attribute such as Outlook {overcast, sunny, rain} can be represented as (0,0,1), (0,1,0), and (1,0,0). This can be achieved by setting the coding type parameter to 'dummy coding' in the Nominal to Numerical operator. Generally, if the number of values in an attribute is not too large, this coding might be more stable than using a single number.

To get a more accurate classification model from SVM, scaling is recommended. The main advantage of scaling is to avoid attributes in greater numeric ranges dominating those in smaller numeric ranges. Another advantage is to avoid numerical difficulties during the calculation. Because kernel values usually depend on the inner products of feature vectors, e.g. the linear kernel and the polynomial kernel, large attribute values might cause numerical problems. Scaling should be performed on both training and testing data sets. In this process the scale parameter is checked. Uncheck the scale parameter and run the process again. You will see that this time it takes a lot longer than the time taken with scaling.

You should have a good understanding of kernel types and different parameters associated with each kernel type in order to get better results from this operator. The gaussian combination kernel was used in this example process. All parameters were used with default values. The accuracy of this model was just 35.71%. Try changing different parameters to get better results. If you change the parameter C to 1 instead of 0, you will see that accuracy of the model rises to 64.29%. Thus, you can see how making small changes in parameters can have a significant effect on overall results. Thus it is very necessary to have a good understanding of parameters of kernel type in use. It is equally important to have a good understanding of different kernel types, and choosing the most suitable kernel type for your ExampleSet. Try using the polynomial kernel in this Example Process (also set the parameter C to 0); you will see that accuracy is around 71.43% with default values for all parameters. Change the value of the parameter C to 1 instead of 0. Doing this increased the accuracy of model with gaussian combination kernel, but here you will see that accuracy of the model drops.

We used default values for most of the parameters. To get more accurate results these values should be carefully selected. Usually techniques like cross-validation are used to find the best values of these parameters for the ExampleSet under consideration.

# Support Vector Machine (Evolutionary)




This operator is a SVM implementation using an evolutionary algorithm to solve the dual optimization problem of an SVM.

## Description

The Support Vector Machine (Evolutionary) uses an Evolutionary Strategy for optimization. This operator is a SVM implementation using an evolutionary algorithm to solve the dual optimization problem of an SVM. It turns out that on many datasets this simple implementation is as fast and accurate as the usual SVM implementations. In addition, it is also capable of learning with Kernels which are not positive semi-definite and can also be used for multi-objective learning which makes the selection of the parameter *C* unnecessary before learning. For more information please study 'Evolutionary Learning with Kernels: A Generic Solution for Large Margin Problems' by Ingo Mierswa.

This operator supports various kernel types including *dot, radial, polynomial, sigmoid, anova, epachnenikov, gaussian combination* and *multiquadric*. Explanation of these kernel types is given in the parameters section.

Here is a basic description of the SVM. The standard SVM takes a set of input data and predicts, for each given input, which of the two possible classes comprises the input, making the SVM a non-probabilistic binary linear classifier. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

More formally, a support vector machine constructs a hyperplane or set of hyperplanes in a high- or infinite- dimensional space, which can be used for classification, regression, or other tasks. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training data points of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier. Whereas the original problem may be stated in a finite dimensional space, it often happens that the sets to discriminate are not linearly separable in that space. For this reason, it was proposed that the original finite-dimensional space be mapped into a much higher-dimensional space, presumably making the separation easier in that space. To keep the computational load reasonable, the mapping used by the SVM schemes are designed to ensure that dot products may be computed easily in terms of the variables in the original space, by defining them in terms of a kernel function *K(x,y)* selected to suit the problem. The hyperplanes in the higher dimensional space are defined as the set of points whose inner product with a vector in that space is constant.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before the application of this operator.

## Output Ports

**model *(mod)*** The SVM model is delivered from this output port. This model can now be applied on unseen data sets.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**kernel type *(selection)*** The type of the kernel function is selected through this parameter. Following kernel types are supported: *dot, radial, polynomial, sigmoid, anova, epachnenikov, gaussian combination, multiquadric*

- **dot** The dot kernel is defined by*k(x,y)=x*y* i.e. it is inner product of*x* and *y*.
- **radial** The radial kernel is defined by *exp(-g ||x-y||^2)* where *g* is the *gamma*, it is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by *k(x,y)=(x*y+1)^d* where *d* is the degree of polynomial and it is specified by the *kernel degree* parameter. The polynomial kernels are well suited for problems where all the training data is normalized.
- **sigmoid** The sigmoid kernel is defined by a two layered neural net *tanh(a x*y+b)* where *a* is *alpha* and *b* is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is 1/N, where N is the data dimension. Note that not all choices of *a* and *b* lead to a valid kernel function.
- **anova** The anova kernel is defined by raised to power *d* of summation of *exp(-g (x-y))* where *g* is *gamma* and *d* is *degree*. gamma and degree are adjusted by the *kernel gamma* and *kernel degree* parameters respectively.
- **epachnenikov** The epachnenikov kernel is this function *(3/4)(1-u2)* for *u* between -1 and 1 and zero for *u* outside that range. It has two adjustable parameters *kernel sigma1* and *kernel degree*.
- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters *kernel sigma1, kernel sigma2* and *kernel sigma3*.
- **multiquadric** The multiquadric kernel is defined by the square root of *||x-y||^2 + c^2*. It has adjustable parameters *kernel sigma1* and *kernel sigma shift*.

**kernel gamma *(real)*** This is the kernel parameter gamma. This is only available when the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1 *(real)*** This is the kernel parameter sigma1. This is only available when the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2 *(real)*** This is the kernel parameter sigma2. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3 *(real)*** This is the kernel parameter sigma3. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel shift *(real)*** This is the kernel parameter shift. This is only available when the *kernel type* parameter is set to *multiquadric*.

**kernel degree** ***(real)*** This is the kernel parameter degree. This is only available when the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a** ***(real)*** This is the kernel parameter a. This is only available when the *kernel type* parameter is set to *sigmoid*

**kernel b** ***(real)*** This is the kernel parameter b. This is only available when the *kernel type* parameter is set to *sigmoid*

**C** ***(real)*** This is the complexity constant which sets the tolerance for misclassification, where higher *C* values allow for 'softer' boundaries and lower values create 'harder' boundaries. A complexity constant that is too large can lead to over-fitting, while values that are too small may result in over-generalization.

**epsilon** This parameter specifies the width of the regression tube loss function of the regression SVM.

**start population type** ***(selection)*** This parameter specifies the type of start population initialization.

**max generations** ***(integer)*** This parameter specifies the number of generations after which the algorithm should be terminated.

**generations without improval** ***(integer)*** This parameter specifies the stop criterion for early stopping i.e. it stops after *n* generations without improvement in the performance. *n* is specified by this parameter.

**population size** ***(integer)*** This parameter specifies the population size i.e. the number of individuals per generation. If set to -1, all examples are selected.

**tournament fraction** ***(real)*** This parameter specifies the fraction of the current population which should be used as tournament members.

**keep best** ***(boolean)*** This parameter specifies if the best individual should survive. This is also called elitist selection. Retaining the best individuals in a generation unchanged in the next generation, is called elitism or elitist selection.

**mutation type** ***(selection)*** This parameter specifies the type of the mutation operator.

**selection type** ***(selection)*** This parameter specifies the selection scheme of this evolutionary algorithms.

**crossover prob** ***(real)*** The probability for an individual to be selected for crossover is specified by this parameter.

**use local random seed** ***(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed** ***(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

**hold out set ratio** ***(real)*** This operator uses this amount as a hold out set to estimate generalization error after learning.

**show convergence plot** ***(boolean)*** This parameter indicates if a dialog with a convergence plot should be drawn.

**show population plot *(boolean)*** This parameter indicates if the population plot in case of the non-dominated sorting should be shown.

**return optimization performance *(boolean)*** This parameter indicates if final optimization fitness should be returned as performance.

## Tutorial Processes

### Introduction to the Support Vector Machine (Evolutionary) operator

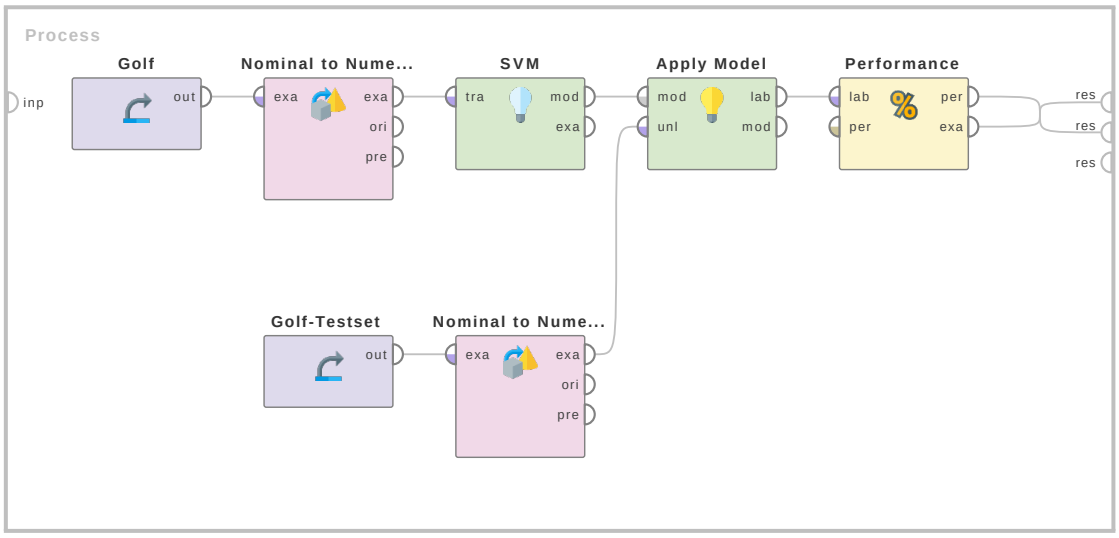



Figure 4.44: Tutorial process 'Introduction to the Support Vector Machine (Evolutionary) operator'.

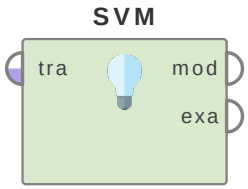
This is a simple Example Process which gets you started with the SVM (Evolutionary) operator. The Retrieve operator is used to load the 'Golf' data set. The Nominal to Numerical operator is applied on it to convert its nominal attributes to numerical form. This step is necessary because the SVM (Evolutionary) operator cannot take nominal attributes, it can only classify using numerical attributes. The model generated from the SVM (Evolutionary) operator is then applied on the 'Golf-Testset' data set. The Nominal to Numerical operator was applied on this data set as well. This is necessary because the testing and training data set should be in the same format. The statistical performance of this model is measured using the Performance operator. This is a very basic process. It is recommended that you develop a deeper understanding of SVM for getting better results through this operator. The support vector machine (SVM) is a popular classification technique. However, beginners who are not familiar with SVM often get unsatisfactory results since they miss some easy but significant steps.

You should have a good understanding of kernel types and different parameters associated with each kernel type in order to get better results from this operator. The gaussian combination kernel was used in this example process. All parameters were used with default values. The accuracy of this model was just 35.71%. Try changing different parameters to get better results. If you change the parameter C to 1 instead of 0, you will see that accuracy of the model rises to 64.29%. Thus, you can see how making small changes in parameters can have a significant effect on overall results. Thus it is very necessary to have a good understanding of parameters of kernel

type in use. It is equally important to have a good understanding of different kernel types, and choosing the most suitable kernel type for your ExampleSet. Try using the polynomial kernel in this Example Process (also set the parameter C to 0); you will see that accuracy is around 64.29% with default values for all parameters. Change the value of the parameter C to 1 instead of 0. Doing this increased the accuracy of model with gaussian combination kernel, but here you will see that accuracy of the model drops.

Default values were used for most of the parameters in the process. To get more accurate results these values should be carefully selected. Usually techniques like cross-validation are used to find the best values of these parameters for the ExampleSet under consideration.

# Support Vector Machine (LibSVM)




This operator is an SVM (Support vector machine) Learner. It is based on the Java libSVM.

## Description

This operator applies the “http://www.csie.ntu.edu.tw/ cjlin/libsvm” libsvm learner by Chih-Chung Chang and Chih-Jen Lin. SVM is a powerful method for both classification and regression. This operator supports the *C-SVC* and *nu-SVC* SVM types for classification tasks as well as the *epsilon-SVR* and *nu-SVR* SVM types for regression tasks. Additionally *one-class* SVM type is supported for distribution estimation. The *one-class* SVM type gives the possibility to learn from just one class of examples and later on test if new examples match the known ones. In contrast to other SVM learners, the libsvm supports internal multiclass learning and probability estimation based on Platt scaling for proper confidence values after applying the learned model on a classification data set.

Here is a basic description of SVM. The standard SVM takes a set of input data and predicts, for each given input, which of two possible classes comprises the input, making the SVM a non-probabilistic binary linear classifier. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

More formally, a support vector machine constructs a hyperplane or set of hyperplanes in a high- or infinite- dimensional space, which can be used for classification, regression, or other tasks. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training data points of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier. Whereas the original problem may be stated in a finite dimensional space, it often happens that the sets to discriminate are not linearly separable in that space. For this reason, it was proposed that the original finite-dimensional space would be mapped into a much higher-dimensional space, presumably making the separation easier in that space. To keep the computational load reasonable, the mapping used by SVM schemes are designed to ensure that dot products may be computed easily in terms of the variables in the original space, by defining them in terms of a kernel function *K(x,y)* selected to suit the problem. The hyperplanes in the higher dimensional space are defined as the set of points whose inner product with a vector in that space is constant.

For more information regarding libsvm you can visit ”http://www.csie.ntu.edu.tw/ cjlin/libsvm”.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before applying this operator.

## Output Ports

**model *(mod)*** The SVM model is delivered from this output port. This model can now be applied on unseen data sets.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**svm type *(selection)*** The SVM type is selected through this parameter. This operator supports the *C-SVC* and *nu-SVC* SVM types for classification tasks. The *epsilon-SVR* and *nu-SVR* SVM types are for regression tasks. The *one-class* SVM type is for distribution estimation. The *one-class* SVM type gives the possibility to learn from just one class of examples and later on test if new examples match the known ones.

**kernel type *(selection)*** The type of the kernel function is selected through this parameter. Following kernel types are supported: *linear, poly, rbf, sigmoid, precomputed*. The *rbf* kernel type is the default value. In general, the *rbf* kernel is a reasonable first choice. Here are a few guidelines regarding different kernel types.

- the *rbf* kernel nonlinearly maps samples into a higher dimensional space
- the *rbf* kernel, unlike the *linear* kernel, can handle the case when the relation between class labels and attributes is nonlinear
- the *linear* kernel is a special case of the *rbf* kernel
- the *sigmoid* kernel behaves like the *rbf* kernel for certain parameters
- the number of hyperparameters influence the complexity of model selection. The *poly* kernel has more hyperparameters than the *rbf* kernel
- the *rbf* kernel has fewer numerical difficulties
- the *sigmoid* kernel is not valid under some parameters
- There are some situations where the *rbf* kernel is not suitable. In particular, when the number of features is very large, one may just use the *linear* kernel.

**degree *(real)*** This parameter is only available when the *kernel type* parameter is set to 'poly'. This parameter is used to specify the degree for a polynomial kernel function.

**gamma *(real)*** This parameter is only available when the *kernel type* parameter is set to 'poly', 'rbf' or 'sigmoid'. This parameter specifies *gamma* for 'polynomial', 'rbf', and 'sigmoid' kernel functions. The value of *gamma* may play an important role in the SVM model. Changing the value of *gamma* may change the accuracy of the resulting SVM model. So, it is a good practice to use cross-validation to find the optimal value of *gamma*.

**coef0 *(real)*** This parameter is only available when the *kernel type* parameter is set to 'poly' or 'precomputed'. This parameter specifies coef0 for 'poly' and 'precomputed' kernel functions.

**C *(real)*** This parameter is only available when the *svm type* parameter is set to 'c-SVC', 'epsilon-SVR' or 'nu-SVR'. This parameter specifies the cost parameter C for 'c-SVC', 'epsilon-SVR' and 'nu-SVR'. C is the penalty parameter of the error term.

**nu *(real)*** This parameter is only available when the *svm type* parameter is set to 'nu-SVC', 'one-class' and 'nu-SVR'. This parameter specifies the *nu* parameter for 'nu-SVC', 'one-class' and 'nu-SVR'. Its value should be between 0.0 and 0.5.

**cache size *(real)*** This is an expert parameter. It specifies the Cache size in Megabyte.

**epsilon *(real)*** This parameter specifies the tolerance of the termination criterion.

**p *(real)*** This parameter is only available when the *svm type* parameter is set to 'epsilon-SVR'. This parameter specifies tolerance of loss function of 'epsilon-SVR'.

**class weights *(list)*** This is an expert parameter. It specifies the weights 'w' for all classes. The *Edit List* button opens a new window with two columns. The first column specifies the class name and the second column specifies the weight for that class. Parameter C is calculated as *weight* of class multiplied by *C*. If weight of a class is not specified, that class is assigned *weight = 1*.

**shrinking *(boolean)*** This is an expert parameter. It specifies whether to use the shrinking heuristics.

**calculate confidences *(boolean)*** This parameter indicates if proper confidence values should be calculated.

**confidence for multiclass *(boolean)*** This is an expert parameter. It indicates if the class with the highest confidence should be selected in the multiclass setting. Uses binary majority vote over all 1-vs-1 classifiers otherwise (selected class must not be the one with highest confidence in that case).

## Tutorial Processes

### SVM with rbf kernel

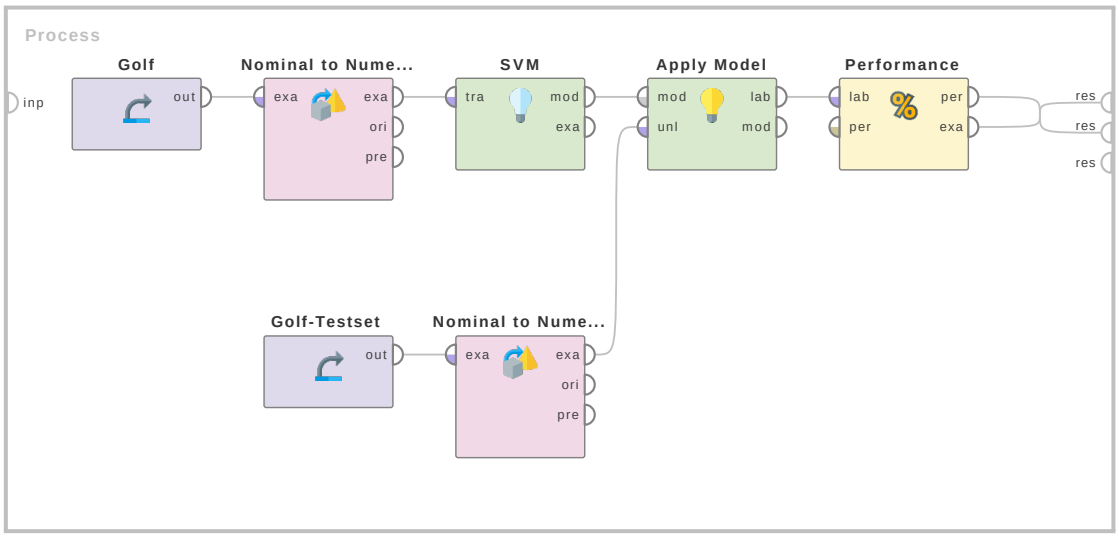



Figure 4.45: Tutorial process 'SVM with rbf kernel'.

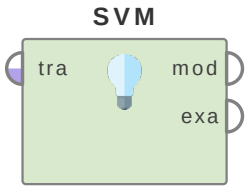
This is a simple Example Process which gets you started with the SVM(libSVM) operator. The Retrieve operator is used to load the 'Golf' data set. The Nominal to Numerical operator is applied on it to convert its nominal attributes to numerical form. This step is necessary because the SVM(libSVM) operator cannot take nominal attributes, it can only classify using numerical attributes. The model generated from the SVM(libSVM) operator is then applied on the 'Golf-Testset' data set using the Apply Model operator. The Nominal to Numerical operator was also applied on this data set. This is necessary because the testing and training data sets should be in the same format. The statistical performance of this model is measured using the Performance operator. This is a very basic process. It is recommended that you develop a deeper understanding of the SVM(libSVM) for getting better results through this operator. The support vector machine (SVM) is a popular classification technique. However, beginners who are not familiar with SVM often get unsatisfactory results since they miss some easy but significant steps.

Using 'm' numbers to represent an m-category attribute is recommended. Only one of the 'm' numbers is 1, and others are 0. For example, a three-category attribute such as Outlook {overcast, sunny, rain} can be represented as (0,0,1), (0,1,0), and (1,0,0). This can be achieved by setting the coding type parameter to 'dummy coding' in the Nominal to Numerical operator. Generally, if the number of values in an attribute is not too large, this coding might be more stable than using a single number.

This basic process omitted various essential steps that are necessary for getting acceptable results from this operator. For example to get a more accurate classification model from SVM, scaling is recommended. The main advantage of scaling is to avoid attributes in greater numeric ranges dominating those in smaller numeric ranges. Another advantage is to avoid numerical difficulties during the calculation. Because kernel values usually depend on the inner products of feature vectors, e.g. the linear kernel and the polynomial kernel, large attribute values might cause numerical problems. Scaling should be performed on both training and testing data sets.

We have used default values of the parameters C, gamma and epsilon. To get more accurate results these values should be carefully selected. Usually techniques like cross-validation are used to find best values of these parameters for the ExampleSet under consideration.

# Support Vector Machine (PSO)




This operator is a Support Vector Machine (SVM) Learner which uses Particle Swarm Optimization (PSO) for optimization. PSO is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality.

## Description

This operator implements a hybrid approach which combines support vector classifier with particle swarm optimization, in order to improve the strength of each individual technique and compensate for each other's weaknesses. Particle Swarm Optimization (PSO) is an evolutionary computation technique in which each potential solution is seen as a particle with a certain velocity flying through the problem space. Support Vector Machine (SVM) classification operates a linear separation in an augmented space by means of some defined kernels satisfying Mercer's condition. These kernels map the input vectors into a very high dimensional space, possibly of infinite dimension, where linear separation is more likely. Then a linear separating hyper plane is found by maximizing the margin between two classes in this space. Hence the complexity of the separating hyper plane depends on the nature and the properties of the used kernel.

### Support Vector Machine (SVM)

Here is a basic description of the SVM. The standard SVM takes a set of input data and predicts, for each given input, which of the two possible classes comprises the input, making the SVM a non-probabilistic binary linear classifier. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on. For more information about SVM please study the description of the SVM operator.

### Particle Swarm Optimization (PSO)

Particle swarm optimization (PSO) is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. PSO is a metaheuristic as it makes few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions. However, metaheuristics such as PSO do not guarantee an optimal solution is ever found. More specifically, PSO does not use the gradient of the problem being optimized, which means PSO does not require that the optimization problem be differentiable as is required by most classic optimization methods. PSO can therefore also be used on optimization problems that are partially irregular, noisy, change over time, etc.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Moreover, this operator can only be applied on ExampleSets with binominal label.

## Output Ports

**model *(mod)*** The SVM model is delivered from this output port. This model can now be applied on unseen data sets.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**show convergence plot *(boolean)*** This parameter indicates if a dialog with a convergence plot should be drawn.

**kernel type *(selection)*** The type of the kernel function is selected through this parameter. Following kernel types are supported: *dot, radial, polynomial, neural, anova, epachnenikov, gaussian combination, multiquadric*

- **dot** The dot kernel is defined by*k(x,y)=x*y* i.e. it is inner product of*x* and *y*.
- **radial** The radial kernel is defined by *exp(-g ||x-y||^2)* where *g* is the *gamma*, it is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by *k(x,y)=(x*y+1)^d* where *d* is the degree of polynomial and it is specified by the *kernel degree* parameter. The polynomial kernels are well suited for problems where all the training data is normalized.
- **neural** The neural kernel is defined by a two layered neural net *tanh(a x*y+b)* where *a* is *alpha* and *b* is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is 1/N, where N is the data dimension. Note that not all choices of *a* and *b* lead to a valid kernel function.
- **anova** The anova kernel is defined by raised to power *d* of summation of *exp(-g (x-y))* where *g* is *gamma* and *d* is *degree*. gamma and degree are adjusted by the *kernel gamma* and *kernel degree* parameters respectively.
- **epachnenikov** The epachnenikov kernel is this function *(3/4)(1-u2)* for *u* between -1 and 1 and zero for *u* outside that range. It has two adjustable parameters *kernel sigma1* and *kernel degree*.
- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters *kernel sigma1, kernel sigma2* and *kernel sigma3*.
- **multiquadric** The multiquadric kernel is defined by the square root of *||x-y||^2 + c^2*. It has adjustable parameters *kernel sigma1* and *kernel sigma shift*.

**kernel gamma *(real)*** This is the SVM kernel parameter gamma. This is available only when the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1 *(real)*** This is the SVM kernel parameter sigma1. This is available only when the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2 *(real)*** This is the SVM kernel parameter sigma2. This is available only when the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3 *(real)*** This is the SVM kernel parameter sigma3. This is available only when the *kernel type* parameter is set to *gaussian combination*.

**kernel shift *(real)*** This is the SVM kernel parameter shift. This is available only when the *kernel type* parameter is set to *multiquadric*.

**kernel degree *(real)*** This is the SVM kernel parameter degree. This is available only when the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a *(real)*** This is the SVM kernel parameter a. This is available only when the *kernel type* parameter is set to *neural*.

**kernel b *(real)*** This is the SVM kernel parameter b. This is available only when the *kernel type* parameter is set to *neural*.

**C *(real)*** This is the SVM complexity constant which sets the tolerance for misclassification, where higher C values allow for 'softer' boundaries and lower values create 'harder' boundaries. A complexity constant that is too large can lead to over-fitting, while values that are too small may result in over-generalization.

**max evaluation *(integer)*** This is an optimizer parameter. It specifies to stop evaluations after the specified number of evaluations.

**generations without improval *(integer)*** This parameter specifies the stop criterion for early stopping i.e. it stops after *n* generations without improvement in the performance. *n* is specified by this parameter.

**population size *(integer)*** This parameter specifies the population size i.e. the number of individuals per generation.

**inertia weight *(real)*** This parameter specifies the (initial) weight for the old weighting.

**local best weight *(real)*** This parameter specifies the weight for the individual's best position during run.

**global best weight *(real)*** This parameter specifies the weight for the population's best position during run.

**dynamic inertia weight *(boolean)*** This parameter specifies if the inertia weight should be improved during run.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Introduction to the SVM (PSO) operator

The 'Ripley-Set' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a classification model. The SVM (PSO) operator is applied in the training subprocess of the Split Validation operator. The SVM (PSO) operator is applied with default values of all parameters. The Apply Model operator is used in the testing subprocess for applying the model generated by the SVM (PSO) operator. The resultant labeled ExampleSet is used by the Performance (Classification) operator for measuring the performance of the model.

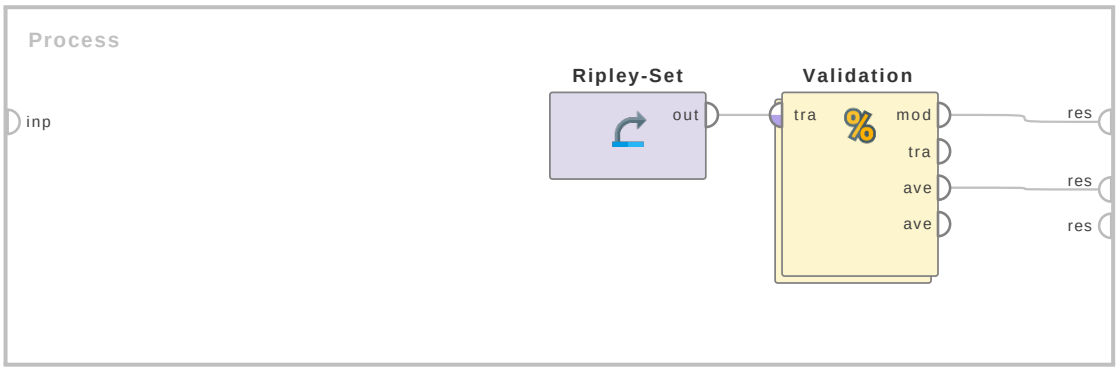


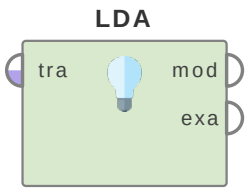

Figure 4.46: Tutorial process ‘Introduction to the SVM (PSO) operator’.

The classification model and its performance vector are connected to the output and they can be seen in the Results Workspace. The accuracy of this model turns out to be around 85%.

Default values were used for most of the parameters. To get more reliable results these values should be carefully selected. Usually techniques like cross-validation are used to find the best values of these parameters for the ExampleSet under consideration.

# 4.1.9 Discriminant Analysis

## Linear Discriminant Analysis




This operator performs linear discriminant analysis (LDA). This method tries to find the linear combination of features which best separate two or more classes of examples. The resulting combination is then used as a linear classifier. Discriminant analysis is used to determine which variables discriminate between two or more naturally occurring groups, it may have a descriptive or a predictive objective.

### Description

This operator performs linear discriminant analysis (LDA). This method tries to find the linear combination of features which best separates two or more classes of examples. The resulting combination is then used as a linear classifier. LDA is closely related to ANOVA (analysis of variance) and regression analysis, which also attempt to express one dependent variable as a linear combination of other features or measurements. In the other two methods however, the dependent variable is a numerical quantity, while for LDA it is a categorical variable (i.e. the class label). LDA is also closely related to principal component analysis (PCA) and factor analysis in that both look for linear combinations of variables which best explain the data. LDA explicitly attempts to model the difference between the classes of data. PCA on the other hand does not take into account any difference in class.

Discriminant analysis is used to determine which variables discriminate between two or more naturally occurring groups. For example, an educational researcher may want to investigate which variables discriminate between high school graduates who decide (1) to go to college, (2) NOT to go to college. For that purpose the researcher could collect data on numerous variables prior to students' graduation. After graduation, most students will naturally fall into one of the two categories. Discriminant Analysis could then be used to determine which variable(s) are the best predictors of students' subsequent educational choice. Computationally, discriminant function analysis is very similar to analysis of variance (ANOVA). For example, suppose the same student graduation scenario. We could have measured students' stated intention to continue on to college one year prior to graduation. If the means for the two groups (those who actually went to college and those who did not) are different, then we can say that intention to attend college as stated one year prior to graduation allows us to discriminate between those who are and are not college bound (and this information may be used by career counselors to provide the appropriate guidance to the respective students). The basic idea underlying discriminant analysis is to determine whether groups differ with regard to the mean of a variable, and then to use that variable to predict group membership (e.g., of new cases).

Discriminant Analysis may be used for two objectives: either we want to assess the adequacy of classification, given the group memberships of the objects under study; or we wish to assign objects to one of a number of (known) groups of objects. Discriminant Analysis may thus have a descriptive or a predictive objective. In both cases, some group assignments must be known before carrying out the Discriminant Analysis. Such group assignments, or labeling, may be arrived at in any way. Hence Discriminant Analysis can be employed as a useful complement to Cluster Analysis (in order to judge the results of the latter) or Principal Components Analysis.

## Differentiation

- **Quadratic Discriminant Analysis** The QDA performs a quadratic discriminant analysis (QDA). QDA is closely related to linear discriminant analysis (LDA), where it is assumed that the measurements are normally distributed. Unlike LDA however, in QDA there is no assumption that the covariance of each of the classes is identical.

  See page 528 for details.

- **Regularized Discriminant Analysis** The RDA regularized discriminant analysis (RDA) which is a generalization of the LDA and QDA. Both algorithms are special cases of this algorithm. If the alpha parameter is set to 1, RDA operator performs LDA. Similarly if the alpha parameter is set to 0, RDA operator performs QDA.

  See page 531 for details.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The Discriminant Analysis is performed and the resultant model is delivered from this output port

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**approximate covariance inverse *(boolean)*** This parameter indicates whether the inverse of the covariance matrices should be approximated if the actual inverse does not exist. This is activated by default.

## Related Documents

- **Quadratic Discriminant Analysis** (page 528)
- **Regularized Discriminant Analysis** (page 531)

## Tutorial Processes

### Introduction to the LDA operator

The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at this ExampleSet. The Linear Discriminant Analysis operator is applied on this ExampleSet. The Linear Discriminant Analysis operator performs the discriminant analysis and the resultant model can be seen in the Results Workspace.

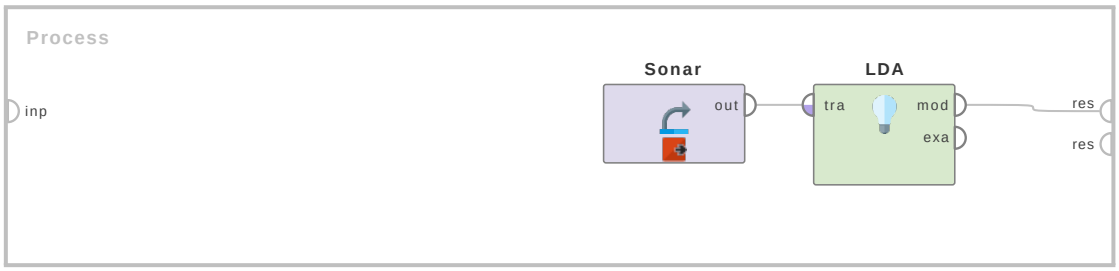
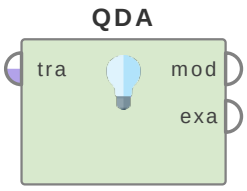



Figure 4.47: Tutorial process 'Introduction to the LDA operator'.

## Quadratic Discriminant Analysis




This operator performs quadratic discriminant analysis (QDA) for nominal labels and numerical attributes. Discriminant analysis is used to determine which variables discriminate between two or more naturally occurring groups, it may have a descriptive or a predictive objective.

### Description

This operator performs a quadratic discriminant analysis (QDA). QDA is closely related to linear discriminant analysis (LDA), where it is assumed that the measurements are normally distributed. Unlike LDA however, in QDA there is no assumption that the covariance of each of the classes is identical. To estimate the parameters required in quadratic discrimination more computation and data is required than in the case of linear discrimination. If there is not a great difference in the group covariance matrices, then the latter will perform as well as quadratic discrimination. Quadratic Discrimination is the general form of Bayesian discrimination.

Discriminant analysis is used to determine which variables discriminate between two or more naturally occurring groups. For example, an educational researcher may want to investigate which variables discriminate between high school graduates who decide (1) to go to college, (2) NOT to go to college. For that purpose the researcher could collect data on numerous variables prior to students' graduation. After graduation, most students will naturally fall into one of the two categories. Discriminant Analysis could then be used to determine which variable(s) are the best predictors of students' subsequent educational choice. Computationally, discriminant function analysis is very similar to analysis of variance (ANOVA). For example, suppose the same student graduation scenario. We could have measured students' stated intention to continue on to college one year prior to graduation. If the means for the two groups (those who actually went to college and those who did not) are different, then we can say that the intention to attend college as stated one year prior to graduation allows us to discriminate between those who are and are not college bound (and this information may be used by career counselors to provide the appropriate guidance to the respective students). The basic idea underlying discriminant analysis is to determine whether groups differ with regard to the mean of a variable, and then to use that variable to predict group membership (e.g. of new cases).

Discriminant Analysis may be used for two objectives: either we want to assess the adequacy of classification, given the group memberships of the objects under study; or we wish to assign objects to one of a number of (known) groups of objects. Discriminant Analysis may thus have

a descriptive or a predictive objective. In both cases, some group assignments must be known before carrying out the Discriminant Analysis. Such group assignments, or labeling, may be arrived at in any way. Hence Discriminant Analysis can be employed as a useful complement to Cluster Analysis (in order to judge the results of the latter) or Principal Components Analysis.

## Differentiation

- **Linear Discriminant Analysis** The QDA performs a quadratic discriminant analysis (QDA). QDA is closely related to linear discriminant analysis (LDA), where it is assumed that the measurements are normally distributed. Unlike LDA however, in QDA there is no assumption that the covariance of each of the classes is identical.

  See page 526 for details.

- **Regularized Discriminant Analysis** The RDA regularized discriminant analysis (RDA) which is a generalization of the LDA and QDA. Both algorithms are special cases of this algorithm. If the alpha parameter is set to 1, RDA operator performs LDA. Similarly if the alpha parameter is set to 0, RDA operator performs QDA.

  See page 531 for details.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The Discriminant Analysis is performed and the resultant model is delivered from this output port

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**approximate covariance inverse *(boolean)*** This parameter indicates whether the inverse of the covariance matrices should be approximated if the actual inverse does not exist. This is activated by default.

## Related Documents

- **Linear Discriminant Analysis** (page 526)
- **Regularized Discriminant Analysis** (page 531)

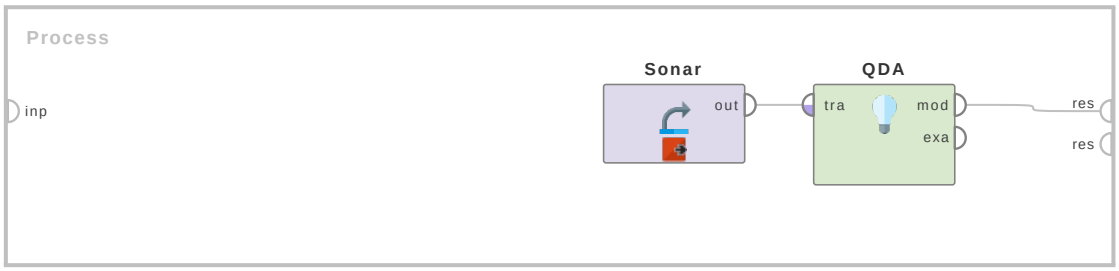


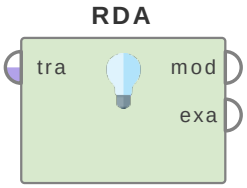

Figure 4.48: Tutorial process ‘Introduction to the QDA operator’.

## Tutorial Processes

### Introduction to the QDA operator

The ‘Sonar’ data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at this ExampleSet. The Quadratic Discriminant Analysis operator is applied on this ExampleSet. The Quadratic Discriminant Analysis operator performs the discriminant analysis and the resultant model can be seen in the Results Workspace.

# Regularized Discriminant Analysis




This operator performs a regularized discriminant analysis (RDA). for nominal labels and numerical attributes. Discriminant analysis is used to determine which variables discriminate between two or more naturally occurring groups, it may have a descriptive or a predictive objective.

## Description

The regularized discriminant analysis (RDA) is a generalization of the linear discriminant analysis (LDA) and the quadratic discreminant analysis (QDA). Both algorithms are special cases of this algorithm. If the *alpha* parameter is set to 1, this operator performs LDA. Similarly if the *alpha* parameter is set to 0, this operator performs QDA. For more information about LDA and QDA please study the documentation of the corresponding operators.

Discriminant analysis is used to determine which variables discriminate between two or more naturally occurring groups. For example, an educational researcher may want to investigate which variables discriminate between high school graduates who decide (1) to go to college, (2) NOT to go to college. For that purpose the researcher could collect data on numerous variables prior to students' graduation. After graduation, most students will naturally fall into one of the two categories. Discriminant Analysis could then be used to determine which variable(s) are the best predictors of students' subsequent educational choice. Computationally, discriminant function analysis is very similar to analysis of variance (ANOVA). For example, suppose the same student graduation scenario. We could have measured students' stated intention to continue on to college one year prior to graduation. If the means for the two groups (those who actually went to college and those who did not) are different, then we can say that intention to attend college as stated one year prior to graduation allows us to discriminate between those who are and are not college bound (and this information may be used by career counselors to provide the appropriate guidance to the respective students). The basic idea underlying discriminant analysis is to determine whether groups differ with regard to the mean of a variable, and then to use that variable to predict group membership (e.g., of new cases).

Discriminant Analysis may be used for two objectives: either we want to assess the adequacy of classification, given the group memberships of the objects under study; or we wish to assign objects to one of a number of (known) groups of objects. Discriminant Analysis may thus have a descriptive or a predictive objective. In both cases, some group assignments must be known before carrying out the Discriminant Analysis. Such group assignments, or labeling, may be arrived at in any way. Hence Discriminant Analysis can be employed as a useful complement to Cluster Analysis (in order to judge the results of the latter) or Principal Components Analysis.

## Differentiation

- **Linear Discriminant Analysis** The RDA operator performs regularized discriminant analysis (RDA) which is a generalization of the LDA which is special cases of this algorithm. If the *alpha* parameter is set to 1, the RDA operator performs LDA.

  See page 526 for details.

- **Quadratic Discriminant Analysis** The RDA operator performs regularized discriminant analysis (RDA) which is a generalization of the QDA which is special cases of this algorithm. If the *alpha* parameter is set to 0, the RDA operator performs QDA.

See page 528 for details.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The Discriminant Analysis is performed and the resultant model is delivered from this output port

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**alpha *(real)*** This parameter specifies the strength of regularization. If set to 1, only global covariance is used. If set to 0, only per class covariance is used.

**approximate covariance inverse *(boolean)*** This parameter indicates whether the inverse of the covariance matrices should be approximated if the actual inverse does not exist. This is activated by default.

## Related Documents

- **Quadratic Discriminant Analysis** (page 528)
- **Linear Discriminant Analysis** (page 526)

## Tutorial Processes

### Introduction to the RDA operator

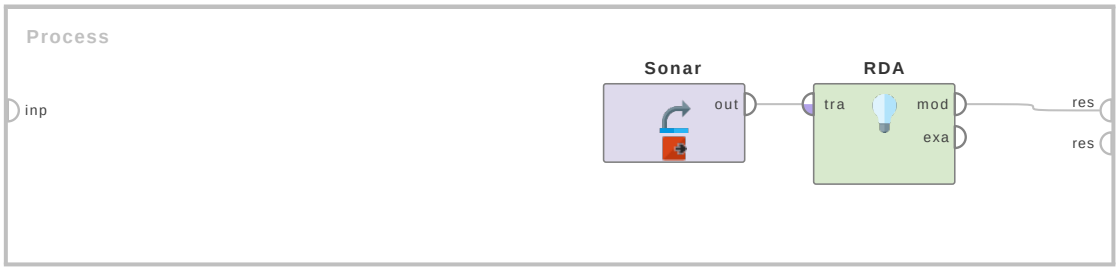


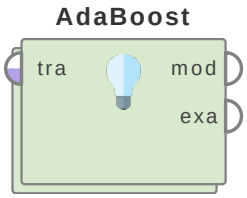

Figure 4.49: Tutorial process 'Introduction to the RDA operator'.

The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at this ExampleSet. The Regularized Discriminant Analysis operator

is applied on this ExampleSet. The Regularized Discriminant Analysis operator performs the discriminant analysis and the resultant model can be seen in the Results Workspace.

# 4.1.10 Ensembles

## AdaBoost




This operator is an implementation of the AdaBoost algorithm and it can be used with all learners available in RapidMiner. AdaBoost is a meta-algorithm which can be used in conjunction with many other learning algorithms to improve their performance.

### Description

The AdaBoost operator is a nested operator i.e. it has a subprocess. The subprocess must have a learner i.e. an operator that expects an ExampleSet and generates a model. This operator tries to build a better model using the learner provided in its subprocess. You need to have a basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

AdaBoost, short for Adaptive Boosting, is a meta-algorithm, and can be used in conjunction with many other learning algorithms to improve their performance. AdaBoost is adaptive in the sense that subsequent classifiers built are tweaked in favor of those instances misclassified by previous classifiers. AdaBoost is sensitive to noisy data and outliers. In some problems, however, it can be less susceptible to the overfitting problem than most learning algorithms. The classifiers it uses can be weak (i.e., display a substantial error rate), but as long as their performance is not random (resulting in an error rate of 0.5 for binary classification), they will improve the final model.

AdaBoost generates and calls a new weak classifier in each of a series of rounds $t = 1,...,T$ . For each call, a distribution of weights *D(t)* is updated that indicates the importance of examples in the data set for the classification. On each round, the weights of each incorrectly classified example are increased, and the weights of each correctly classified example are decreased, so the new classifier focuses on the examples which have so far eluded correct classification.

#### Ensemble Theory

Boosting is an ensemble method, therefore an overview of the Ensemble Theory has been discussed here. Ensemble methods use multiple models to obtain better predictive performance than could be obtained from any of the constituent models. In other words, an ensemble is a technique for combining many weak learners in an attempt to produce a strong learner. Evaluating the prediction of an ensemble typically requires more computation than evaluating the prediction of a single model, so ensembles may be thought of as a way to compensate for poor learning algorithms by performing a lot of extra computation.

An ensemble is itself a supervised learning algorithm, because it can be trained and then used to make predictions. The trained ensemble, therefore, represents a single hypothesis. This hypothesis, however, is not necessarily contained within the hypothesis space of the models from which it is built. Thus, ensembles can be shown to have more flexibility in the functions they can represent. This flexibility can, in theory, enable them to over-fit the training data more than a single model would, but in practice, some ensemble techniques (especially bagging) tend to reduce problems related to over-fitting of the training data.

Empirically, ensembles tend to yield better results when there is a significant diversity among the models. Many ensemble methods, therefore, seek to promote diversity among the models they combine. Although perhaps non-intuitive, more random algorithms (like random decision

trees) can be used to produce a stronger ensemble than very deliberate algorithms (like entropy-reducing decision trees). Using a variety of strong learning algorithms, however, has been shown to be more effective than using techniques that attempt to dumb-down the models in order to promote diversity.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The meta model is delivered from this output port which can now be applied on unseen data sets for prediction of the *label* attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**iterations *(integer)*** This parameter specifies the maximum number of iterations of the AdaBoost algorithm.

## Tutorial Processes

### Using the AdaBoost operator for generating a better Decision Tree

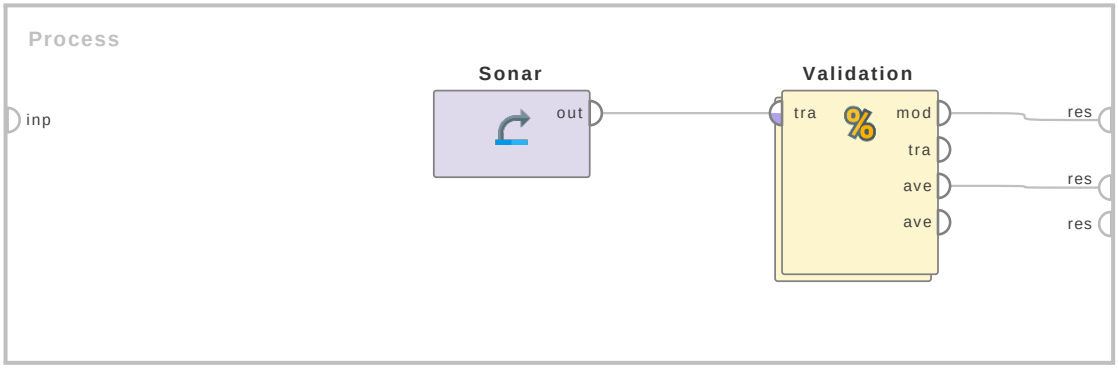
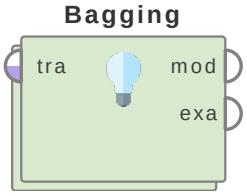



Figure 4.50: Tutorial process 'Using the AdaBoost operator for generating a better Decision Tree'.

The 'Sonar' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a classification model. The AdaBoost operator is applied in the training subprocess of the Split Validation operator. The Decision Tree operator is applied in the subprocess of the AdaBoost operator. The iterations parameter of the AdaBoost operator is set to 10, thus there will be at maximum 10 iterations of its subprocess. The Apply Model

operator is used in the testing subprocess for applying the model generated by the AdaBoost operator. The resultant labeled ExampleSet is used by the Performance (Classification) operator for measuring the performance of the model. The classification model and its performance vector is connected to the output and it can be seen in the Results Workspace. You can see that the AdaBoost operator produced a new model in each iteration and there are different weights for each model. The accuracy of this model turns out to be around 69%. If the same process is repeated without AdaBoost operator i.e. only the Decision Tree operator is used in training subprocess. The accuracy of that model turns out to be around 66%. Thus AdaBoost generated a combination of models that performed better than the original model.

# Bagging




Bootstrap aggregating (bagging) is a machine learning ensemble meta-algorithm to improve classification and regression models in terms of stability and classification accuracy. It also reduces variance and helps to avoid overfitting. Although it is usually applied to decision tree models, it can be used with any type of model.

## Description

The Bagging operator is a nested operator i.e. it has a subprocess. The subprocess must have a learner i.e. an operator that expects an ExampleSet and generates a model. This operator tries to build a better model using the learner provided in its subprocess. You need to have basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

The concept of bagging (voting for classification, averaging for regression-type problems with continuous dependent variables of interest) applies to the area of predictive data mining, to combine the predicted classifications (prediction) from multiple models, or from the same type of model for different learning data. It is also used to address the inherent instability of results when applying complex models to relatively small data sets. Suppose your data mining task is to build a model for predictive classification, and the dataset from which to train the model (learning data set, which contains observed classifications) is relatively small. You could repeatedly sub-sample (with replacement) from the dataset, and apply, for example, a tree classifier (e.g., CHAID) to the successive samples. In practice, very different trees will often be grown for the different samples, illustrating the instability of models often evident with small data sets. One method of deriving a single prediction (for new observations) is to use all trees found in the different samples, and to apply some simple voting: The final classification is the one most often predicted by the different trees. Note that some weighted combination of predictions (weighted vote, weighted average) is also possible, and commonly used. A sophisticated algorithm for generating weights for weighted prediction or voting is the Boosting procedure which is available in RapidMiner as AdaBoost operator.

### Ensemble Theory

Bagging is an ensemble method, therefore an overview of the Ensemble Theory has been discussed here. Ensemble methods use multiple models to obtain a better predictive performance than could be obtained from any of the constituent models. In other words, an ensemble is a technique for combining many weak learners in an attempt to produce a strong learner. Evaluating the prediction of an ensemble typically requires more computation than evaluating the prediction of a single model, so ensembles may be thought of as a way to compensate for poor learning algorithms by performing a lot of extra computation.

An ensemble is itself a supervised learning algorithm, because it can be trained and then used to make predictions. The trained ensemble, therefore, represents a single hypothesis. This hypothesis, however, is not necessarily contained within the hypothesis space of the models from which it is built. Thus, ensembles can be shown to have more flexibility in the functions they can represent. This flexibility can, in theory, enable them to over-fit the training data more than a single model would, but in practice, some ensemble techniques (especially bagging) tend to reduce problems related to over-fitting of the training data.

Empirically, ensembles tend to yield better results when there is a significant diversity among the models. Many ensemble methods, therefore, seek to promote diversity among the models they combine. Although perhaps non-intuitive, more random algorithms (like random decision trees) can be used to produce a stronger ensemble than very deliberate algorithms (like entropy-reducing decision trees). Using a variety of strong learning algorithms, however, has been shown to be more effective than using techniques that attempt to dumb-down the models in order to promote diversity.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The meta model is delivered from this output port which can now be applied on an unseen data sets for prediction of the *label* attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**sample ratio *(real)*** This parameter specifies the fraction of examples to be used for training. Its value must be greater than 0 (i.e. zero examples) and should be lower than or equal to 1 (i.e. entire data set).

**iterations *(integer)*** This parameter specifies the maximum number of iterations of the Bagging algorithm.

**average confidences *(boolean)*** This parameter specifies whether to average available prediction confidences or not.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same sample. Changing the value of this parameter changes the way examples are randomized, thus the sample will have a different set of values.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is available only if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Using the Bagging operator for generating a better Decision Tree

The 'Sonar' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a classification model. The Bagging operator is applied in the training subprocess of the Split Validation operator. The Decision Tree operator is applied in the subprocess of the Bagging operator. The iterations parameter of the Bagging operator is set to 10, thus there will be 10 iterations of its subprocess. The Apply Model operator is used in

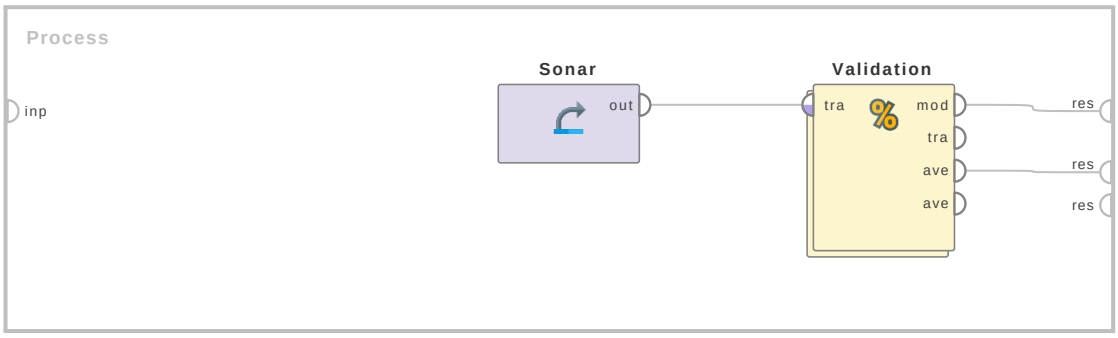
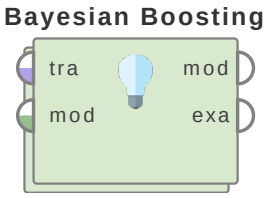



Figure 4.51: Tutorial process 'Using the Bagging operator for generating a better Decision Tree'.

the testing subprocess for applying the model generated by the Bagging operator. The resultant labeled ExampleSet is used by the Performance (Classification) operator for measuring the performance of the model. The classification model and its performance vector are connected to the output and they can be seen in the Results Workspace. You can see that the Bagging operator produced a new model in each iteration. The accuracy of this model turns out to be around 75.81%. If the same process is repeated without the Bagging operator i.e. only the Decision Tree operator is used in the training subprocess then the accuracy of that model turns out to be around 66%. Thus Bagging improved the performance of the base learner (i.e. Decision Tree).

# Bayesian Boosting




This operator is a boosting operator based on Bayes' theorem. It implements a meta-algorithm which can be used in conjunction with many other learning algorithms to improve their performance.

## Description

The Bayesian Boosting operator is a nested operator i.e. it has a subprocess. The subprocess must have a learner i.e. an operator that expects an ExampleSet and generates a model. This operator tries to build a better model using the learner provided in its subprocess. You need to have a basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

This operator trains an ensemble of classifiers for boolean target attributes. In each iteration the training set is reweighted, so that previously discovered patterns and other kinds of prior knowledge are 'sampled out'. An inner classifier, typically a rule or decision tree induction algorithm, is sequentially applied several times, and the models are combined to a single global model. The maximum number of models to be trained are specified by the *iterations* parameter.

If the *rescale label priors* parameter is set to true, then the ExampleSet is reweighted, so that all classes are equally probable (or frequent). For two-class problems this turns the problem of fitting models to maximize weighted relative accuracy into the more common task of classifier induction. Applying a rule induction algorithm as an inner learner allows to do subgroup discovery. This option is also recommended for data sets with class skew, if a very weak learner like a decision stump is used. If the *rescale label priors* parameter is not set, then the operator performs boosting based on probability estimates.

If the *allow marginal skews* parameter is not set, then the support of each subset defined in terms of common base model predictions does not change from one iteration to the next. Analogously the class priors do not change. This is the procedure originally described in 'Scholz/2005b' in the context of subgroup discovery. Setting the *allow marginal skews* option to true leads to a procedure that changes the marginal weights/probabilities of subsets, if this is beneficial in a boosting context, and stratifies the two classes to be equally likely. As for AdaBoost, the total weight upper-bounds the training error in this case. This bound is reduced more quickly by the Bayesian Boosting operator.

To reproduce the sequential sampling, or knowledge-based sampling, from 'Scholz/2005b' for subgroup discovery, two of the default parameter settings of this operator have to be changed: *rescale label priors* must be set to true, and *allow marginal skews* must be set to false. In addition, a boolean (binomial) label has to be used.

This operator requires an ExampleSet as its input. To sample out prior knowledge of a different form it is possible to provide another model as an optional additional input. The predictions of this model are used to produce an initial weighting of the training set. The output of the operator is a classification model applicable for estimating conditional class probabilities or for plain crisp classification. It contains up to the specified number of inner base models. In the case of an optional initial model, this model will also be stored in the output model, in order to produce the same initial weighting during model application.

### Ensemble Theory

Boosting is an ensemble method, therefore an overview of the Ensemble Theory has been discussed here. Ensemble methods use multiple models to obtain better predictive performance than could be obtained from any of the constituent models. In other words, an ensemble is a technique for combining many weak learners in an attempt to produce a strong learner. Evaluating the prediction of an ensemble typically requires more computation than evaluating the prediction of a single model, so ensembles may be thought of as a way to compensate for poor learning algorithms by performing a lot of extra computation.

An ensemble is itself a supervised learning algorithm, because it can be trained and then used to make predictions. The trained ensemble, therefore, represents a single hypothesis. This hypothesis, however, is not necessarily contained within the hypothesis space of the models from which it is built. Thus, ensembles can be shown to have more flexibility in the functions they can represent. This flexibility can, in theory, enable them to over-fit the training data more than a single model would, but in practice, some ensemble techniques (especially bagging) tend to reduce problems related to over-fitting of the training data.

Empirically, ensembles tend to yield better results when there is a significant diversity among the models. Many ensemble methods, therefore, seek to promote diversity among the models they combine. Although perhaps non-intuitive, more random algorithms (like random decision trees) can be used to produce a stronger ensemble than very deliberate algorithms (like entropy-reducing decision trees). Using a variety of strong learning algorithms, however, has been shown to be more effective than using techniques that attempt to dumb-down the models in order to promote diversity.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

**model *(mod)*** The input port expects a model. This is an optional port. To sample out prior knowledge of a different form it is possible to provide a model as an optional input. The predictions of this model are used to produce an initial weighting of the training set. The output of the operator is a classification model applicable for estimating conditional class probabilities or for plain crisp classification. It contains up to the specified number of inner base models. In the case of an optional initial model, this model will also be stored in the output model, in order to produce the same initial weighting during model application.

## Output Ports

**model *(mod)*** The meta model is delivered from this output port which can now be applied on unseen data sets for prediction of the *label* attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**use subset for training *(real)*** This parameter specifies the fraction of examples to be used for training, remaining examples are used to estimate the confusion matrix. If set to 1, the test set is turned off.

**iterations** ***(integer)*** This parameter specifies the maximum number of iterations of this algorithm.

**rescale label priors** ***(boolean)*** This parameter specifies whether the proportion of labels should be equal by construction after first iteration. Please study the description of this operator for more information about this parameter.

**allow marginal skews** ***(boolean)*** This parameter specifies if the skewing of the marginal distribution *(P(x))* should be allowed during learning. Please study the description of this operator for more information about this parameter.

**use local random seed** ***(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same sample. Changing the value of this parameter changes the way examples are randomized, thus the sample will have a different set of values.

**local random seed** ***(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Using the Bayesian Boosting operator for generating a better Decision Tree

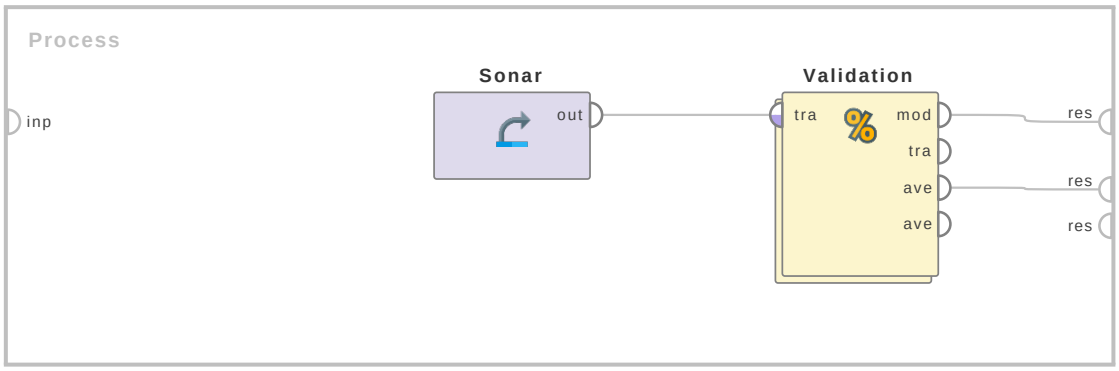
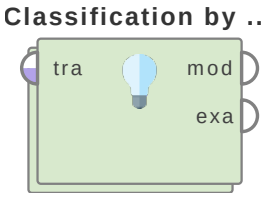



Figure 4.52: Tutorial process 'Using the Bayesian Boosting operator for generating a better Decision Tree'.

The 'Sonar' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a classification model. The Bayesian Boosting operator is applied in the training subprocess of the Split Validation operator. The Decision Tree operator is applied in the subprocess of the Bayesian Boosting operator. The iterations parameter of the Bayesian Boosting operator is set to 10, thus there will be at maximum 10 iterations of its subprocess. The Apply Model operator is used in the testing subprocess for applying the model generated by the Bayesian Boosting operator. The resultant labeled ExampleSet is used by the Performance (Classification) operator for measuring the performance of the model. The classification model and its performance vector is connected to the output and it can be seen in the Results Workspace. You can see that the Bayesian Boosting operator produced a new model in each iteration. The accuracy of this model turns out to be around 67.74%. If the same process is repeated without Bayesian Boosting operator i.e. only the Decision Tree operator is used in

training subprocess. The accuracy of that model turns out to be around 66%. Thus Bayesian Boosting generated a combination of models that performed better than the original model.

# Classification by Regression




This operator builds a polynominal classification model through the given regression learner.

## Description

The Classification by Regression operator is a nested operator i.e. it has a subprocess. The subprocess must have a regression learner i.e. an operator that generates a regression model. This operator builds a classification model using the regression learner provided in its subprocess. You need to have a basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

Here is an explanation of how a classification model is built from a regression learner. For each class *i* of the given ExampleSet, a regression model is trained after setting the label to +1 if the label is *i* and to -1 if it is not. Then the regression models are combined into a classification model. This model can be applied using the Apply Model operator. In order to determine the prediction for an unlabeled example, all regression models are applied and the class belonging to the regression model which predicts the greatest value is chosen.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The classification model is delivered from this output port. This classification model can now be applied on unseen data sets for prediction of the *label* attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Tutorial Processes

### Using the Linear Regression operator for classification

The 'Sonar' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a classification model. The Classification by Regression operator is applied in the training subprocess of the Split Validation operator. The Linear Regression operator is applied in the subprocess of the Classification by Regression operator. Although Linear Regression is a regression learner but it will be used by the Classification by Regression operator to train a classification model. The Apply Model operator is used in the testing subprocess to apply the model. The resultant labeled ExampleSet is used by the Performance (Classification) operator for measuring the performance of the model. The classification model and its performance vector is connected to the output and it can be seen in the Results Workspace.

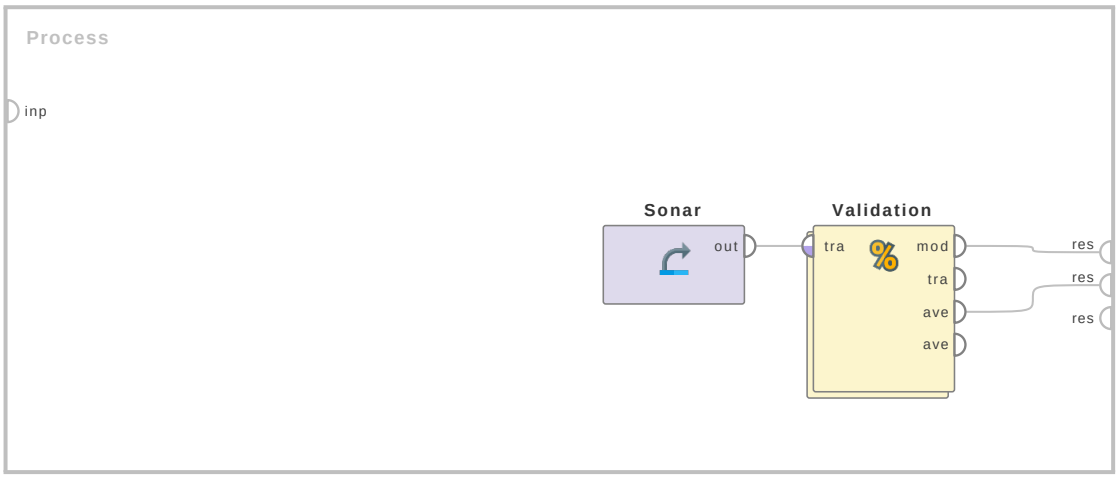
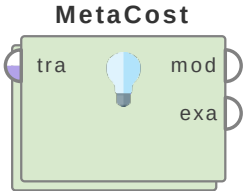



Figure 4.53: Tutorial process 'Using the Linear Regression operator for classification'.

# MetaCost




This metaclassifier makes its base classifier cost-sensitive by using the given cost matrix to compute label predictions according to classification costs.

## Description

The MetaCost operator makes its base classifier cost-sensitive by using the cost matrix specified in the *cost matrix* parameter. The method used by this operator is similar to the MetaCost method described by Pedro Domingos (1999).

The MetaCost operator is a nested operator i.e. it has a subprocess. The subprocess must have a learner i.e. an operator that expects an ExampleSet and generates a model. This operator tries to build a better model using the learner provided in its subprocess. You need to have basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

Most classification algorithms assume that all errors have the same cost, which is seldom the case. For example, in database marketing the cost of mailing to a non-respondent is very small, but the cost of not mailing to someone who would respond is the entire profit lost. In general, misclassification costs may be described by an arbitrary cost matrix *C*, with *C(i,j)* being the cost of predicting that an example belongs to class *i* when in fact it belongs to class *j*. Individually making each classification learner cost-sensitive is laborious, and often non-trivial. MetaCost is a principled method for making an arbitrary classifier cost-sensitive by wrapping a cost-minimizing procedure around it. This procedure treats the underlying classifier as a black box, requiring no knowledge of its functioning or change to it. MetaCost is applicable to any number of classes and to arbitrary cost matrices.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The meta model is delivered from this output port which can now be applied on unseen data sets for prediction of the *label* attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**cost matrix *(string)*** This parameter is used for specifying the cost matrix. The cost matrix is similar in structure to a confusion matrix because it has predicted classes in one dimension and actual classes on the other dimension. Therefore the cost matrix can denote the costs for every possible classification outcome: predicted label vs. actual label. Actually this matrix is a matrix of misclassification costs because you can specify different weights for certain classes misclassified as other classes. Weights can also be assigned to correct classifications but usually they are set to 0. The classes in the matrix are labeled as Class 1, Class 2 etc where classes are numbered according to their order in the internal mapping.

**use subset for training *(real)*** This parameter specifies the fraction of examples to be used for training. Its value must be greater than 0 (i.e. zero examples) and should be lower than or equal to 1 (i.e. entire data set).

**iterations *(integer)*** This parameter specifies the maximum number of iterations of the MetaCost algorithm.

**sampling with replacement *(boolean)*** This parameter indicates if sampling with replacement should be used. In sampling with replacement, at every step all examples have equal probability of being selected. Once an example has been selected for the sample, it remains candidate for selection and it can be selected again in any other coming steps. Thus a sample with replacement can have the same example multiple number of times.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same sample. Changing the value of this parameter changes the way examples are randomized, thus the sample will have a different set of values.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Using the MetaCost operator for generating a better Decision Tree

The 'Sonar' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a classification model. The MetaCost operator is applied in

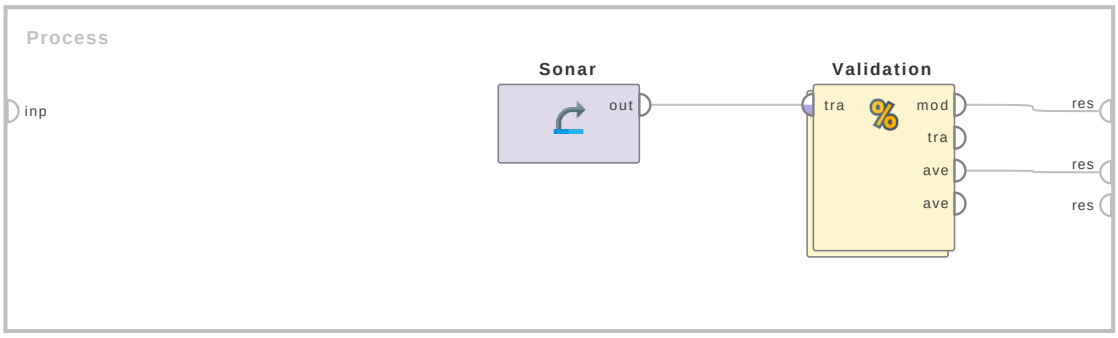
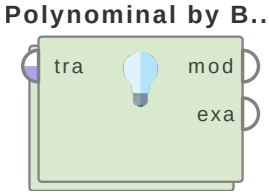



Figure 4.54: Tutorial process 'Using the MetaCost operator for generating a better Decision Tree'.

the training subprocess of the Split Validation operator. The Decision Tree operator is applied in the subprocess of the MetaCost operator. The iterations parameter of the MetaCost operator is set to 10, thus there will be 10 iterations of its subprocess. Have a look at the cost matrix specified in the cost matrix parameter of the MetaCost operator. You can see that the misclassification costs are not equal. The Apply Model operator is used in the testing subprocess for applying the model generated by the MetaCost operator. The resultant labeled ExampleSet is used by the Performance (Classification) operator for measuring the performance of the model. The classification model and its performance vector are connected to the output and they can be seen in the Results Workspace. You can see that the MetaCost operator produced a new model in each iteration. The accuracy of this model turns out to be around 82%. If the same process is repeated without MetaCost operator i.e. only Decision Tree operator is used in training subprocess then the accuracy of that model turns out to be around 66%. Thus MetaCost improved the performance of the base learner (i.e. Decision Tree) by using the cost matrix to compute label predictions according to classification costs.

# Polynomial by Binomial Classification




This operator builds a polynomial classification model through the given binomial classification learner.

## Description

The Polynomial by Binomial Classification operator is a nested operator i.e. it has a subprocess. The subprocess must have a binomial classification learner i.e. an operator that generates a binomial classification model. This operator builds a polynomial classification model using the binomial classification learner provided in its subprocess. You need to have basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

Many classification operators (e.g. the SVM operator) allow for classification only for binomial (binary) label. The Polynomial by Binomial Classification operator uses a binomial classifier and generates binomial classification models for different classes and then aggregates the responses of these binomial classification models for classification of polynomial label.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The polynomial classification model is delivered from this output port. This classification model can now be applied on unseen data sets for prediction of the *label* attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**classification strategies *(selection)*** This parameter specifies the strategy that should be used for multi-class classifications i.e. polynomial classifications.

**random code multiplicator *(real)*** This parameter is only available when the *classification strategies* parameter is set to 'exhaustive code' or 'random code'. This parameter specifies a multiplicator regulating the codeword length in random code modus.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Using the SVM operator for polynomial classification

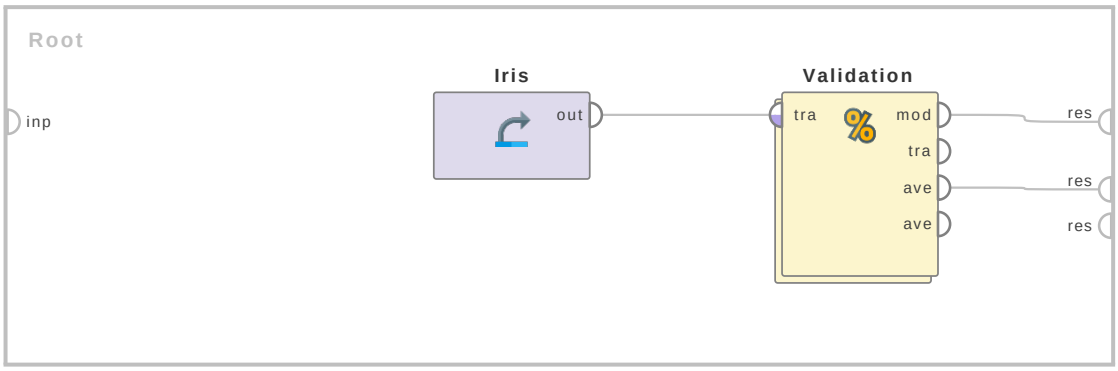



Figure 4.55: Tutorial process 'Using the SVM operator for polynomial classification'.

The 'Iris' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a polynomial classification model. The Polynomial by Binomial Classification operator is applied in the training subprocess of the Split Validation operator. The SVM operator is applied in the subprocess of the Polynomial by Binomial Classification operator. Although SVM is a binomial classification learner but it will be used by the Polynomial by Binomial Classification operator to train a polynomial classification model. The Apply Model operator is used in the testing subprocess to apply the model. The resultant labeled ExampleSet is used by the Performance (Classification) operator for measuring the performance of the model. The polynomial classification model and its performance vector is connected to the output and it can be seen in the Results Workspace.

# Stacking


This operator is an implementation of Stacking which is used for combining the models rather than choosing among them, thereby typically getting a performance better than any single one of the trained models.

## Description

Stacked generalization (or stacking) is a way of combining multiple models, that introduces the concept of a meta learner. Unlike bagging and boosting, stacking may be (and normally is) used to combine models of different types. The procedure is as follows:

1. Split the training set into two disjoint sets.
2. Train several base learners on the first part.
3. Test the base learners on the second part.
4. Using the predictions from step 3 as the inputs, and the correct responses as the outputs, train a higher level learner.

Note that steps 1 to 3 are the same as cross-validation, but instead of using a winner-takes-all approach, we combine the base learners, possibly nonlinearly.

The crucial prior belief underlying the scientific method is that one can judge among a set of models by comparing them on data that was not used to create any of them. This prior belief is used in the cross-validation technique, to choose among a set of models based on a single data set. This is done by partitioning the data set into a training data set and a testing data set; training the models on the training data; and then choosing whichever of those trained models performs best on the testing data.

Stacking exploits this prior belief further. It does this by using performance on the testing data to combine the models rather than choose among them, thereby typically getting a better performance than any single one of the trained models. It has been successfully used on both supervised learning tasks (e.g. regression) and unsupervised learning (e.g. density estimation).

The Stacking operator is a nested operator. It has two subprocess: the Base Learners and the Stacking Model Learner subprocess. You need to have a basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

### Ensemble Theory

Stacking is an ensemble method, therefore an overview of the Ensemble Theory has been discussed here. Ensemble methods use multiple models to obtain a better predictive performance than could be obtained from any of the constituent models. In other words, an ensemble is a technique for combining many weak learners in an attempt to produce a strong learner. Evaluating the prediction of an ensemble typically requires more computation than evaluating the prediction of a single model, so ensembles may be thought of as a way to compensate for poor learning algorithms by performing a lot of extra computation.

An ensemble is itself a supervised learning algorithm, because it can be trained and then used to make predictions. The trained ensemble, therefore, represents a single hypothesis. This hypothesis, however, is not necessarily contained within the hypothesis space of the models from

which it is built. Thus, ensembles can be shown to have more flexibility in the functions they can represent. This flexibility can, in theory, enable them to over-fit the training data more than a single model would, but in practice, some ensemble techniques (especially bagging) tend to reduce problems related to over-fitting of the training data.

Empirically, ensembles tend to yield better results when there is a significant diversity among the models. Many ensemble methods, therefore, seek to promote diversity among the models they combine. Although perhaps non-intuitive, more random algorithms (like random decision trees) can be used to produce a stronger ensemble than very deliberate algorithms (like entropy-reducing decision trees). Using a variety of strong learning algorithms, however, has been shown to be more effective than using techniques that attempt to dumb-down the models in order to promote diversity.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The stacking model is delivered from this output port which can be applied on unseen data sets.

## Parameters

**keep all attributes *(boolean)*** This parameter indicates if all attributes (including the original ones) should be kept in order to learn the stacked model.

## Tutorial Processes

### Introduction to Stacking

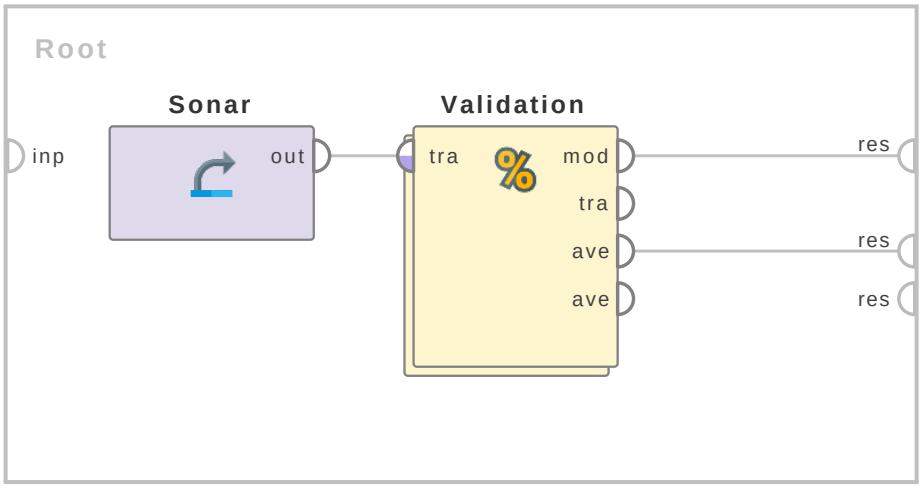
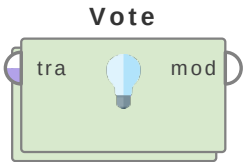



Figure 4.56: Tutorial process 'Introduction to Stacking'.

The ‘Sonar’ data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a model. The Stacking operator is applied in the training subprocess of the Split Validation operator. Three learners are applied in the Base Learner subprocess of the Stacking operator. These learners are: Decision Tree, K-NN and Linear Regression operators. The Naive Bayes operator is applied in the Stacking Model Learner subprocess of the Stacking operator. The Naive Bayes learner is used as a stacking learner which uses the predictions of the preceding three learners to make a combined prediction. The Apply Model operator is used in the testing subprocess of the Split Validation operator for applying the model generated by the Stacking operator. The resultant labeled ExampleSet is used by the Performance operator for measuring the performance of the model. The stacking model and its performance vector is connected to the output and it can be seen in the Results Workspace.

# Vote




This operator uses a majority vote (for classification) or the average (for regression) on top of the predictions of the inner learners (i.e. learning operators in its subprocess).

## Description

The Vote operator is a nested operator i.e. it has a subprocess. The subprocess must have at least two learners, called base learners. This operator builds a classification model or regression model depending upon the ExampleSet and learners. This operator uses a majority vote (for classification) or the average (for regression) on top of the predictions of the base learners provided in its subprocess. You need to have a basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses. All the operator chains in the subprocess must accept an ExampleSet and return a model.

In case of a classification task, all the operators in the subprocess of the Vote operator accept the given ExampleSet and generate a classification model. For prediction of an unknown example, the Vote operator applies all the classification models from its subprocess and assigns the predicted class with maximum votes to the unknown example. Similarly, In case of a regression task, all the operators in the subprocess of the Vote operator accept the given ExampleSet and generate a regression model. For prediction of an unknown example, the Vote operator applies all the regression models from its subprocess and assigns the average of all predicted values to the unknown example.

## Input Ports

**training set *(tra)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**model *(mod)*** The simple vote model for classification or regression is delivered from this output port. This model can now be applied on unseen data sets for prediction of the *label* attribute.

## Tutorial Processes

### Using the Vote operator for classification

The 'Sonar' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a model. The Vote operator is applied in the training subprocess of the Split Validation operator. Three learners are applied in the subprocess of the Vote operator. These base learners are: Decision Tree, Neural Net and SVM. The Vote operator uses the vote of each learner for classification of an example, the prediction with maximum votes is assigned to the unknown example. In other words it uses the predictions of the three base learners to make a combined prediction (using simple voting). The Apply Model operator is used in

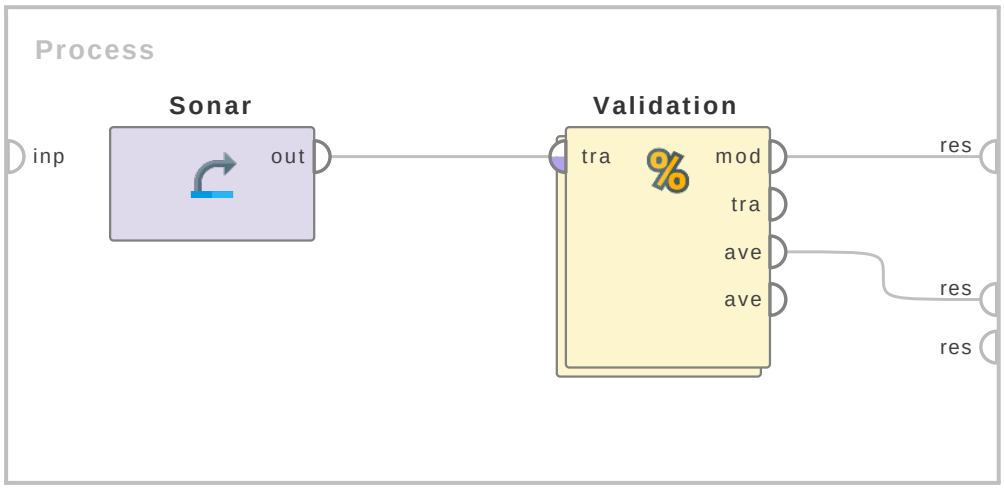
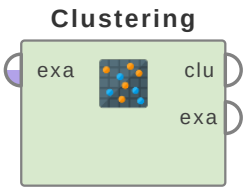



Figure 4.57: Tutorial process 'Using the Vote operator for classification'.

the testing subprocess of the Split Validation operator for applying the model generated by the Vote operator. The resultant labeled ExampleSet is used by the Performance operator for measuring the performance of the model. The Vote model and its performance vector is connected to the output and it can be seen in the Results Workspace.

# 4.2 Segmentation

## Agglomerative Clustering




This operator performs Agglomerative clustering which is a bottom-up strategy of Hierarchical clustering. Three different strategies are supported by this operator: single-link, complete-link and average-link. The result of this operator is a hierarchical cluster model, providing distance information to plot as a dendrogram.

### Description

Agglomerative clustering is a strategy of hierarchical clustering. Hierarchical clustering (also known as Connectivity based clustering) is a method of cluster analysis which seeks to build a hierarchy of clusters. Hierarchical clustering, is based on the core idea of objects being more related to nearby objects than to objects farther away. As such, these algorithms connect 'objects' (or examples, in case of an ExampleSet) to form clusters based on their distance. A cluster can be described largely by the maximum distance needed to connect parts of the cluster. At different distances, different clusters will form, which can be represented using a dendrogram, which explains where the common name 'hierarchical clustering' comes from: these algorithms do not provide a single partitioning of the data set, but instead provide an extensive hierarchy of clusters that merge with each other at certain distances. In a dendrogram, the y-axis marks the distance at which the clusters merge, while the objects are placed along the x-axis so the clusters don't mix.

Strategies for hierarchical clustering generally fall into two types:

- Agglomerative: This is a bottom-up approach: each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy.
- Divisive: This is a top-down approach: all observations start in one cluster, and splits are performed recursively as one moves down the hierarchy.

Hierarchical clustering is a whole family of methods that differ by the way distances are computed. Apart from the usual choice of distance functions, the user also needs to decide on the linkage criterion to use, since a cluster consists of multiple objects, there are multiple candidates to compute the distance to. Popular choices are known as single-linkage clustering (the minimum of object distances), complete-linkage clustering (the maximum of object distances) or average-linkage clustering (also known as UPGMA, 'Unweighted Pair Group Method with Arithmetic Mean').

The algorithm forms clusters in a bottom-up manner, as follows:

1. Initially, put each example in its own cluster.
2. Among all current clusters, pick the two clusters with the smallest distance.
3. Replace these two clusters with a new cluster, formed by merging the two original ones.
4. Repeat the above two steps until there is only one remaining cluster in the pool.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. It is a technique for extracting information from unlabeled data and can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

## Input Ports

**example set** ***(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

## Output Ports

**cluster model** ***(clu)*** This port delivers the hierarchical cluster model. It has information regarding the clustering performed. It explains how clusters were merged to make a hierarchy of clusters.

**example set** ***(exa)*** The ExampleSet that was given as input is passed without any modification to the output through this port.

## Parameters

**mode** ***(selection)*** This parameter specifies the cluster mode or the linkage criterion.

- **SingleLink** In single-link hierarchical clustering, we merge in each step the two clusters whose two closest members have the smallest distance (or: the two clusters with the smallest minimum pairwise distance).
- **CompleteLink** In complete-link hierarchical clustering, we merge in each step the two clusters whose merger has the smallest diameter (or: the two clusters with the smallest maximum pairwise distance).
- **AverageLink** Average-link clustering is a compromise between the sensitivity of complete-link clustering to outliers and the tendency of single-link clustering to form long chains that do not correspond to the intuitive notion of clusters as compact, spherical objects.

**measure types** ***(selection)*** This parameter is used for selecting the type of measure to be used for measuring the distance between points.The following options are available: *mixed measures*, *nominal measures*, *numerical measures* and *Bregman divergences*.

**mixed measure** ***(selection)*** This parameter is available when the *measure type* parameter is set to 'mixed measures'. The only available option is the 'Mixed Euclidean Distance'

**nominal measure** ***(selection)*** This parameter is available when the *measure type* parameter is set to 'nominal measures'. This option cannot be applied if the input ExampleSet has numerical attributes. In this case the 'numerical measure' option should be selected.

**numerical measure** ***(selection)*** This parameter is available when the *measure type* parameter is set to 'numerical measures'. This option cannot be applied if the input ExampleSet has nominal attributes. If the input ExampleSet has nominal attributes the 'nominal measure' option should be selected.

**divergence** ***(selection)*** This parameter is available when the *measure type* parameter is set to 'bregman divergences'.

**kernel type** ***(selection)*** This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance'. The type of the kernel function is selected through this parameter. Following kernel types are supported:

- **dot** The dot kernel is defined by$k(x,y)=x^*y$ i.e.it is inner product of$x$ and $y$.

- **radial** The radial kernel is defined by *exp(-g ||x-y||^2)* where *g* is the *gamma* that is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by *k(x,y)=(x*y+1)^d* where *d* is the degree of the polynomial and it is specified by the *kernel degree* parameter. The Polynomial kernels are well suited for problems where all the training data is normalized.
- **neural** The neural kernel is defined by a two layered neural net *tanh(a x*y+b)* where *a* is *alpha* and *b* is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is 1/N, where N is the data dimension. Note that not all choices of *a* and *b* lead to a valid kernel function.
- **sigmoid** This is the sigmoid kernel. Please note that the *sigmoid* kernel is not valid under some parameters.
- **anova** This is the anova kernel. It has adjustable parameters *gamma* and *degree*.
- **epachnenikov** The Epanechnikov kernel is this function *(3/4)(1-u2)* for *u* between -1 and 1 and zero for *u* outside that range. It has two adjustable parameters *kernel sigma1* and *kernel degree*.
- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters *kernel sigma1, kernel sigma2* and *kernel sigma3*.
- **multiquadric** The multiquadric kernel is defined by the square root of *||x-y||^2 + c^2*. It has adjustable parameters *kernel sigma1* and *kernel sigma shift*.

**kernel gamma *(real)*** This is the SVM kernel parameter gamma. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1 *(real)*** This is the SVM kernel parameter sigma1. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2 *(real)*** This is the SVM kernel parameter sigma2. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3 *(real)*** This is the SVM kernel parameter sigma3. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel shift *(real)*** This is the SVM kernel parameter shift. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *multiquadric*.

**kernel degree *(real)*** This is the SVM kernel parameter degree. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a *(real)*** This is the SVM kernel parameter a. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

**kernel b *(real)*** This is the SVM kernel parameter b. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

## Tutorial Processes

### Agglomerative Clustering of Ripley-Set data set

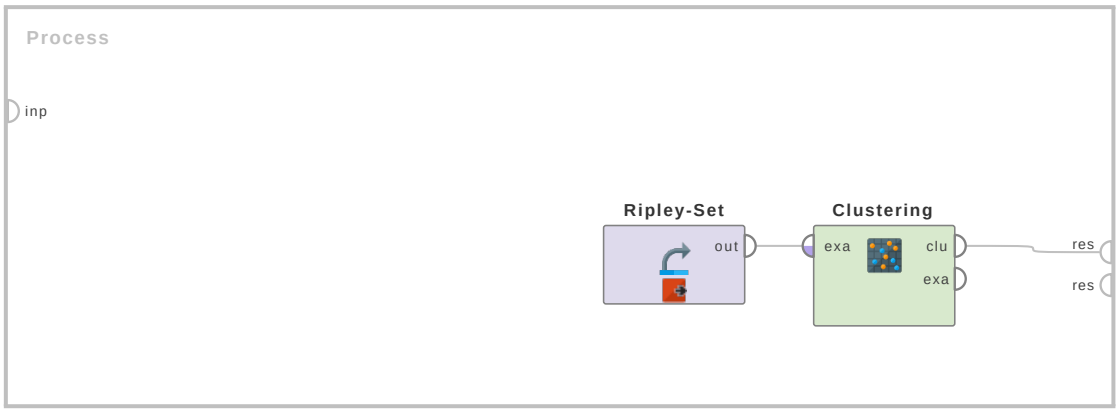
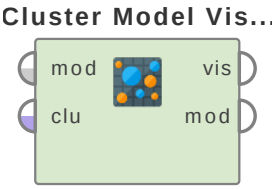



Figure 4.58: Tutorial process 'Agglomerative Clustering of Ripley-Set data set'.

The 'Ripley-Set' data set is loaded using the Retrieve operator. A breakpoint is inserted at this step so that you can have a look at the ExampleSet. The Agglomerative Clustering operator is applied on this ExampleSet. Run the process and switch to the Results Workspace. Note the Graph View of the results. You can see that the algorithm has not created separate groups or clusters as other clustering algorithms (like k-means), instead the result is a hierarchy of clusters. Under the Folder View you can see members of each cluster in folder format. You can see that it is an hierarchy of folders. The Dendogram View shows the dendrogram for this clustering which shows how single-element clusters were joined step by step to make a hierarchy of clusters.

# Cluster Model Visualizer




This operator uses visualization tools for centroid-based cluster models to capture the essential details of each cluster.

## Description

The visualization tools include the following:

- Overview: shows the size of all found clusters, together with some information about the clusters and their quality.
- Heat map:: displays a decision tree describing the main difference between the clusters.
- Centroid Chart: shows the values for the cluster centroids in a parallel chart.
- Centroid table: shows the values for the cluster centroids in a table.
- Scatter plot: with a choice of cluster, displays a scatter plot in terms of the two most important Attributes.

## Input Ports

**model *(mod)*** This input port expects a centroid-based cluster model.

**clustered data *(clu)*** This input port expects a clustered ExampleSet which is the output of the cluster model building process.

## Output Ports

**visualization output *(vis)*** This output port provides visualization tools to help understand clusters.

**model output *(mod)*** The input model is passed without changing to the output through this port.

## Tutorial Processes

### Visualizing Cluster for Iris

This process creates a cluster model on the Iris data set. We use the very common k-Means clustering algorithm with k=3, i.e. we want to find three clusters in the data. The cluster model is then delivered together with the clustered data to the Cluster Model Visualization operator, which creates the visualizations.

Examining the output from each of the visualization tools, we find the following: Overview: Cluster 1 is the biggest cluster with 61 items.Heat map: Cluster 0 has on average much higher values for a1, a3, and a4. The cluster tree, centroid chart, centroid table, and scatter plot show the same results in a different form.

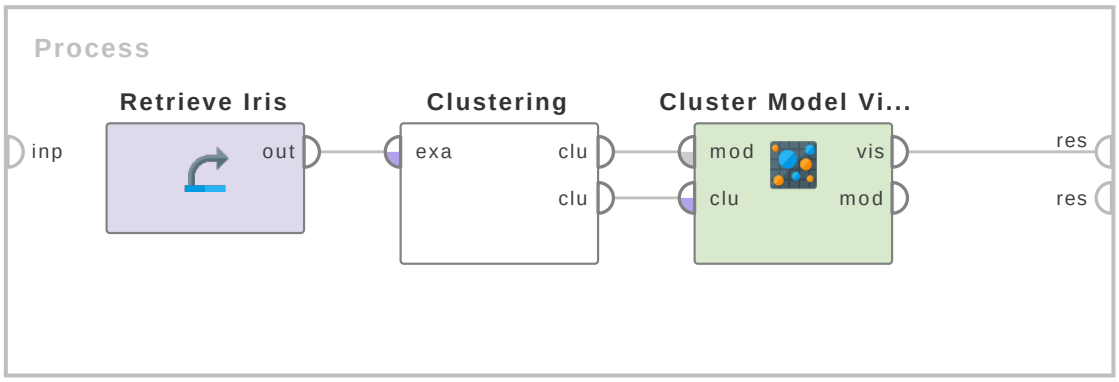
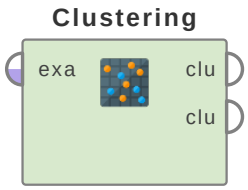



Figure 4.59: Tutorial process 'Visualizing Cluster for Iris'.

# DBSCAN




This operator performs clustering with DBSCAN. DBSCAN (for density-based spatial clustering of applications with noise) is a density-based clustering algorithm because it finds a number of clusters starting from the estimated density distribution of corresponding nodes.

## Description

DBSCAN's definition of a cluster is based on the notion of density reachability. Basically, a point *q* is directly density-reachable from a point *p* if it is not farther away than a given distance epsilon (i.e. it is part of its epsilon-neighborhood) and if *p* is surrounded by sufficiently many points such that one may consider *p* and *q* to be part of a cluster. *q* is called density-reachable (note the distinction from "directly density-reachable") from *p* if there is a sequence *p(1),...,p(n)* of points with *p(1)* = *p* and *p(n)* = *q* where each *p(i+1)* is directly density-reachable from *p(i)*.

Note that the relation of density-reachable is not symmetric. *q* might lie on the edge of a cluster, having insufficiently many neighbors to count as dense itself. This would halt the process of finding a path that stops with the first non-dense point. By contrast, starting the process with *q* would lead to *p* (though the process would halt there, *p* being the first non-dense point). Due to this asymmetry, the notion of density-connected is introduced: two points *p* and *q* are density-connected if there is a point *o* such that both *p* and *q* are density-reachable from *o*. Density-connectedness is symmetric.

A cluster, which is a subset of the points of the data set, satisfies two properties:

1. All points within the cluster are mutually density-connected.

2. If a point is density-connected to any point of the cluster, it is part of the cluster as well.

DBSCAN requires two parameters: epsilon and the minimum number of points required to form a cluster (minPts). epsilon and minPts can be specified through the *epsilon* and *min points* parameters respectively. DBSCAN starts with an arbitrary starting point that has not been visited. This point's epsilon-neighborhood is retrieved, and if it contains sufficiently many points, a cluster is started. Otherwise, the point is labeled as noise. Note that this point might later be

found in a sufficiently sized epsilon-environment of a different point and hence be made part of a cluster.

If a point is found to be a dense part of a cluster, its epsilon-neighborhood is also part of that cluster. Hence, all points that are found within the epsilon-neighborhood are added, as is their own epsilon-neighborhood when they are also dense. This process continues until the density-connected cluster is completely found. Then, a new unvisited point is retrieved and processed, leading to the discovery of a further cluster or noise.

If no id attribute is present, this operator will create one. The 'Cluster 0' assigned by DBSCAN operator corresponds to points that are labeled as noise. These are the points that have less than *min points* points in their epsilon-neighborhood.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. It is a technique for extracting information from unlabeled data and can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**cluster model *(clu)*** This port delivers the cluster model. It has information regarding the clustering performed. It tells which examples are part of which cluster.

**clustered set *(clu)*** The ExampleSet that was given as input is passed with minor changes to the output through this port. An attribute with *id* role is added to the input ExampleSet to distinguish examples. An attribute with *cluster* role may also be added depending on the state of the *add cluster attribute* parameter.

## Parameters

**epsilon *(real)*** This parameter specifies the epsilon parameter of the DBSCAN algorithm. epsilon specifies the size of the neighborhood.

**min points *(integer)*** This parameter specifies the minimal number of points forming a cluster.

**add cluster attribute *(boolean)*** If this parameter is set to true, a new attribute with *cluster* role is generated in the resultant ExampleSet, otherwise this operator does not add the *cluster* attribute. In the latter case you have to use the Apply Model operator to generate the *cluster* attribute.

**add as label *(boolean)*** If this parameter is set to true, the cluster id is stored in an attribute with the *label* role instead of *cluster* role (see *add cluster attribute* parameter).

**remove unlabeled *(boolean)*** If this parameter is set to true, unlabeled examples are deleted from the ExampleSet.

**measure types *(selection)*** This parameter is used for selecting the type of measure to be used for measuring the distance between points.The following options are available: *mixed measures*, *nominal measures*, *numerical measures* and *Bregman divergences*.

**mixed measure** ***(selection)*** This parameter is available when the *measure type* parameter is set to 'mixed measures'. The only available option is the 'Mixed Euclidean Distance'

**nominal measure** ***(selection)*** This parameter is available when the *measure type* parameter is set to 'nominal measures'. This option cannot be applied if the input ExampleSet has numerical attributes. In this case the 'numerical measure' option should be selected.

**numerical measure** ***(selection)*** This parameter is available when the *measure type* parameter is set to 'numerical measures'. This option cannot be applied if the input ExampleSet has nominal attributes. If the input ExampleSet has nominal attributes the 'nominal measure' option should be selected.

**divergence** ***(selection)*** This parameter is available when the *measure type* parameter is set to 'bregman divergences'.

**kernel type** ***(selection)*** This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance'. The type of the kernel function is selected through this parameter. Following kernel types are supported:

- **dot** The dot kernel is defined by*k(x,y)=x*y* i.e.it is inner product of*x* and *y*.
- **radial** The radial kernel is defined by *exp(-g ||x-y||^2)* where *g* is the *gamma* that is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by *k(x,y)=(x*y+1)^d* where *d* is the degree of the polynomial and it is specified by the *kernel degree* parameter. The Polynomial kernels are well suited for problems where all the training data is normalized.
- **neural** The neural kernel is defined by a two layered neural net *tanh(a x*y+b)* where *a* is *alpha* and *b* is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is 1/N, where N is the data dimension. Note that not all choices of *a* and *b* lead to a valid kernel function.
- **sigmoid** This is the sigmoid kernel. Please note that the *sigmoid* kernel is not valid under some parameters.
- **anova** This is the anova kernel. It has adjustable parameters *gamma* and *degree*.
- **epachnenikov** The Epanechnikov kernel is this function *(3/4)(1-u2)* for *u* between -1 and 1 and zero for *u* outside that range. It has two adjustable parameters *kernel sigma1* and *kernel degree*.
- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters *kernel sigma1, kernel sigma2* and *kernel sigma3*.
- **multiquadric** The multiquadric kernel is defined by the square root of *||x-y||^2 + c^2*. It has adjustable parameters *kernel sigma1* and *kernel sigma shift*.

**kernel gamma** ***(real)*** This is the SVM kernel parameter gamma. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1** ***(real)*** This is the SVM kernel parameter sigma1. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2** ***(real)*** This is the SVM kernel parameter sigma2. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3** ***(real)*** This is the SVM kernel parameter sigma3. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel shift** ***(real)*** This is the SVM kernel parameter shift. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *multiquadric*.

**kernel degree** ***(real)*** This is the SVM kernel parameter degree. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a** ***(real)*** This is the SVM kernel parameter a. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

**kernel b** ***(real)*** This is the SVM kernel parameter b. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

## Tutorial Processes

### Clustering of Ripley-Set data set by the DBSCAN operator

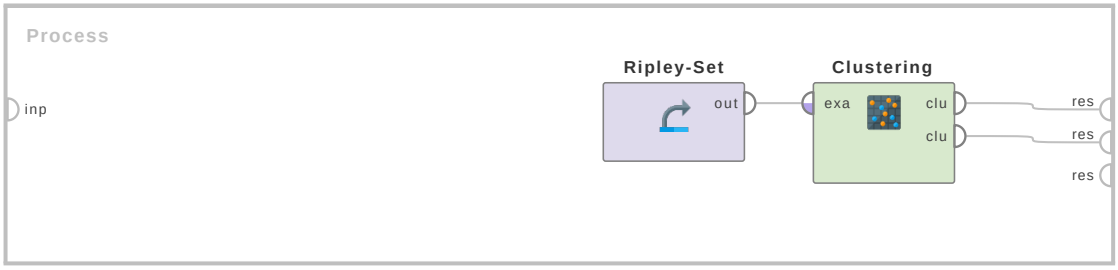


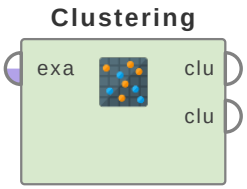

Figure 4.60: Tutorial process 'Clustering of Ripley-Set data set by the DBSCAN operator'.

In many cases, no target attribute (i.e. label) can be defined and the data should be automatically grouped. This procedure is called Clustering. RapidMiner supports a wide range of clustering schemes which can be used in just the same way like any other learning scheme. This includes the combination with all preprocessing operators.

In this Example Process, the 'Ripley-Set' data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A breakpoint is inserted at this step so that you can have a look at the ExampleSet before application of the DBSCAN operator. Other than the label attribute the 'Ripley-Set' has two real attributes; 'att1' and 'att2'. The DBSCAN operator is applied on this data set with default values for all parameters except the epsilon parameter which is set to 0.1. Run the process and you will see that two new attributes are created by the DBSCAN operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which

cluster the examples belong to. Each example is assigned to a particular cluster. The examples in ‘cluster_0’ are considered as noise. Also note the Plot View of this data set. Switch to Plot View and set the the Plotter to ‘Scatter’, x-Axis to ‘att1’, y-Axis to ‘att2’ and Color Column to ‘cluster’. You can clearly see how the algorithm has created three separate groups (noise i.e. cluster_0 is also visible separately). A cluster model is also delivered through the cluster model output port. It has information regarding the clustering performed. Under Folder View you can see members of each cluster in folder format.

# Expectation Maximization Clustering




This operator performs clustering using the Expectation Maximization algorithm. Clustering is concerned with grouping objects together that are similar to each other and dissimilar to the objects belonging to other clusters. But the Expectation Maximization algorithm extends this basic approach to clustering in some important ways.

## Description

The general purpose of clustering is to detect clusters in examples and to assign those examples to the clusters. A typical application for this type of analysis is a marketing research study in which a number of consumer behavior related variables are measured for a large sample of respondents. The purpose of the study is to detect 'market segments', i.e., groups of respondents that are somehow more similar to each other (to all other members of the same cluster) when compared to respondents that belong to other clusters. In addition to identifying such clusters, it is usually equally of interest to determine how the clusters are different, i.e., determine the specific variables or dimensions that vary and how they vary in regard to members in different clusters.

The EM (expectation maximization) technique is similar to the K-Means technique. The basic operation of K-Means clustering algorithms is relatively simple: Given a fixed number of *k* clusters, assign observations to those clusters so that the means across clusters (for all variables) are as different from each other as possible. The EM algorithm extends this basic approach to clustering in two important ways:

- Instead of assigning examples to clusters to maximize the differences in means for continuous variables, the EM clustering algorithm computes probabilities of cluster memberships based on one or more probability distributions. The goal of the clustering algorithm then is to maximize the overall probability or likelihood of the data, given the (final) clusters.

### Expectation Maximization algorithm

The basic approach and logic of this clustering method is as follows. Suppose you measure a single continuous variable in a large sample of observations. Further, suppose that the sample consists of two clusters of observations with different means (and perhaps different standard deviations); within each sample, the distribution of values for the continuous variable follows the normal distribution. The goal of EM clustering is to estimate the means and standard deviations for each cluster so as to maximize the likelihood of the observed data (distribution). Put another way, the EM algorithm attempts to approximate the observed distributions of values based on mixtures of different distributions in different clusters. The results of EM clustering are different from those computed by k-means clustering. The latter will assign observations to clusters to maximize the distances between clusters. The EM algorithm does not compute actual assignments of observations to clusters, but classification probabilities. In other words, each observation belongs to each cluster with a certain probability. Of course, as a final result you can usually review an actual assignment of observations to clusters, based on the (largest) classification probability.

## Differentiation

- **k-Means (Deprecated)** The K-Means operator performs clustering using the k-means algorithm. k-means clustering is an exclusive clustering algorithm i.e. each object is assigned to precisely one of a set of clusters. Objects in one cluster are similar to each other. The similarity between objects is based on a measure of the distance between them. The K-Means operator assigns observations to clusters to maximize the distances between clusters. The Expectation Maximization Clustering operator, on the other hand, computes classification probabilities. See page **??** for details.

## Input Ports

**example set *(exa)*** The input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**cluster model *(clu)*** This port delivers the cluster model which has information regarding the clustering performed. It has information about cluster probabilities and cluster means.

**clustered set *(clu)*** The ExampleSet that was given as input is passed with minor changes to the output through this port. An attribute with *id* role is added to the input ExampleSet to distinguish examples. An attribute with *cluster* role may also be added depending on the state of the *add cluster attribute* parameter. If the *show probabilities* parameter is set to true, one probability column is added for each cluster.

## Parameters

**k *(integer)*** This parameter specifies the number of clusters to form. There is no hard and fast rule of number of clusters to form. But, generally it is preferred to have small number of clusters with examples scattered (not too scattered) around them in a balanced way.

**add cluster attribute *(boolean)*** If enabled, a new attribute with *cluster* role is generated directly in this operator, otherwise this operator does not add the *cluster* attribute. In the latter case you have to use the Apply Model operator to generate the *cluster* attribute.

**add as label *(boolean)*** If true, the cluster id is stored in an attribute with the *label* role instead of *cluster* role (see *add cluster attribute* parameter).

**remove unlabeled *(boolean)*** If set to true, unlabeled examples are deleted.

**max runs *(integer)*** This parameter specifies the maximal number of runs of this operator to be performed with random initialization.

**max optimization steps *(integer)*** This parameter specifies the maximal number of iterations performed for one run of this operator.

**quality *(real)*** This parameter specifies the quality that must be fulfilled before the algorithm stops ( i.e. the rising of the log-likelihood that must be undercut).

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization.

**local random seed** ***(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

**show probabilities** ***(boolean)*** This parameter indicates if the probabilities for every cluster should be inserted with every example in the ExampleSet.

**inital distribution** ***(selection)*** This parameter indicates the initial distribution of the centroids.

**correlated attributes** ***(boolean)*** This parameter should be set to true if the ExampleSet contains correlated attributes.

## Related Documents

- **k-Means (Deprecated)** (page ??)

## Tutorial Processes

### Clustering of the Ripley-Set data set using the Expectation Maximization Clustering operator

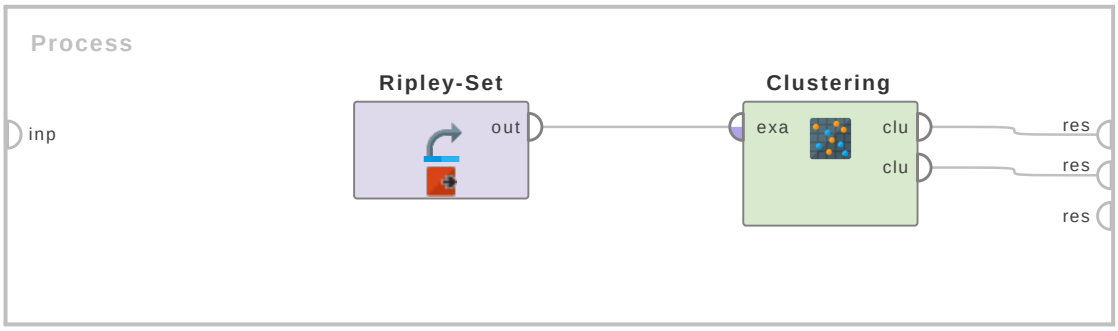
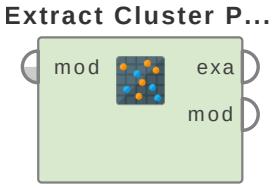



Figure 4.61: Tutorial process 'Clustering of the Ripley-Set data set using the Expectation Maximization Clustering operator'.

The 'Ripley-Set' data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A breakpoint is inserted at this step so that you can have a look at the ExampleSet before the application of the Expectation Maximization Clustering operator. Besides the label attribute the 'Ripley-Set' has two real attributes; 'att1' and 'att2'. The Expectation Maximization Clustering operator is applied on this data set with default values for all parameters. Run the process and you will see that a few new attributes are created by the Expectation Maximization Clustering operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. As parameter k was set to 2, only two clusters are possible. That is why each example is assigned to either 'cluster_0' or 'cluster_1'. Note that the Expectation Maximization Clustering operator has added probability attributes for each cluster that show the probability of an example to be part of that cluster. This operator assigns an example to the cluster with maximum probability. Also note the Plot View of this data. You can clearly see how the algorithm has created two separate groups in the Plot View. A cluster model is also delivered through the cluster model output port. It has information regarding

the clustering performed. It also has information about cluster probabilities and cluster means. Under Folder View you can see members of each cluster in folder format.

# Extract Cluster Prototypes




This operator generates an ExampleSet consisting of the Cluster Prototypes from the Cluster Model. This operator is usually applied after clustering operators to store the Cluster Prototypes in form of an ExampleSet.

## Description

Most clustering algorithms like K-Means or K-Medoids cluster the data around some prototypical data vectors. For example the K-Means algorithm uses the centroid of all examples of a cluster. The Extract Cluster Prototypes operator extracts these prototypes and stores them in an ExampleSet for further use. This operator expects a cluster model as input. The information about the cluster prototypes can be seen in the cluster models generated by most clustering operators but the Extract Cluster Prototypes operator stores this information in form of an ExampleSet thus it can be used easily.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. Clustering is a technique for extracting information from unlabeled data. Clustering can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

## Differentiation

- **k-Medoids** The K-Medoids operator performs the clustering and generates a cluster model and a clustered ExampleSet. The cluster model generated by the K-Medoids operator can be used by the Extract Cluster Prototypes operator to store the *Centroid Table* in form of an ExampleSet. See page 591 for details.

- **k-Means (Deprecated)** The K-Means operator performs the clustering and generates a cluster model and a clustered ExampleSet. The cluster model generated by the K-Means operator can be used by the Extract Cluster Prototypes operator to store the *Centroid Table* in form of an ExampleSet. See page **??** for details.

## Input Ports

**model *(mod)*** This port expects a cluster model. It has information regarding the clustering performed by a clustering operator. It tells which examples are part of which cluster. It also has information regarding centroids of each cluster.

## Output Ports

**example set *(exa)*** The ExampleSet consisting of the Cluster Prototypes is generated from the input Cluster Model and the ExampleSet is delivered through this port

**model *(mod)*** The cluster model that was given as input is passed without any changes to the output through this port.

## Related Documents

- **k-Medoids** (page 591)
- **k-Means (Deprecated)** (page ??)

## Tutorial Processes

### Extracting Centroid Table after application of the K-Means operator on Ripley-Set

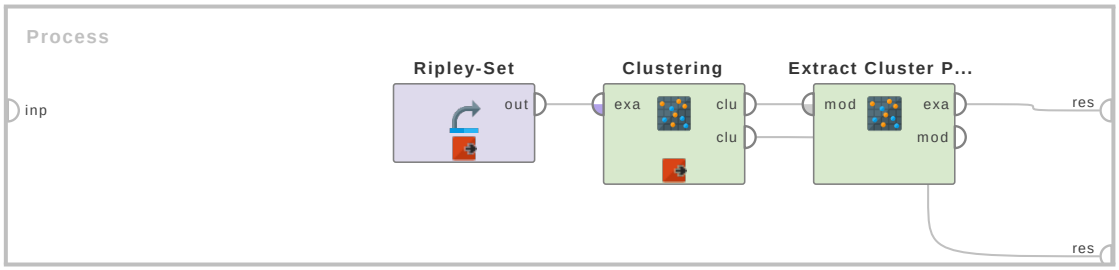
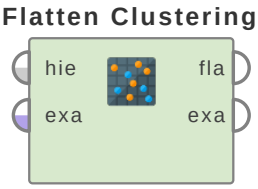



Figure 4.62: Tutorial process 'Extracting Centroid Table after application of the K-Means operator on Ripley-Set'.

In many cases, no target attribute (i.e. label) can be defined and the data should be automatically grouped. This procedure is called Clustering. RapidMiner supports a wide range of clustering schemes which can be used in just the same way like any other learning scheme. This includes the combination with all preprocessing operators.

In this Example Process, the 'Ripley-Set' data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A breakpoint is inserted at this step so that you can have a look at the ExampleSet before application of the K-Means operator. Other than the label attribute the 'Ripley-Set' has two real attributes; 'att1' and 'att2'. The K-Means operator is applied on this data set with default values for all parameters. Run the process and you will see that two new attributes are created by the K-Means operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. As parameter k was set to 2, only two clusters are possible. That is why each example is assigned to either 'cluster_0' or 'cluster_1'. A cluster model is delivered through the cluster model output port. It has information regarding the clustering performed. Under Folder View you can see members of each cluster in folder format. You can see information regarding centroids under the Centroid Table and Centroid Plot View tabs. A breakpoint is inserted at this step so that you can have a look at the cluster model (especially the Centroid Table) before application of the Extract Cluster Prototypes operator. The Extract Cluster Prototypes operator is applied on the cluster model generated by the K-Means operator which stores the Centroid Table in form of an ExampleSet which can be seen in the Results Workspace.

# Flatten Clustering




This operator creates a flat clustering model from the given hierarchical clustering model. Clustering is concerned with grouping objects together that are similar to each other and dissimilar to the objects belonging to other clusters.

## Description

The Flatten Clustering operator creates a flat cluster model from the given hierarchical cluster model by expanding nodes in the order of their distance until the desired number of clusters (specified by the *number of clusters* parameter) is reached. In RapidMiner, operators like the Agglomerative Clustering operator provide hierarchical cluster models. The Flatten Clustering operator takes this hierarchical cluster model and an ExampleSet as input and returns a flat cluster model and the clustered ExampleSet. Please note that RapidMiner also provides operators that perform Flat clustering e.g. the K-Means operator.

Flat clustering creates a flat set of clusters without any explicit structure that would relate clusters to each other. Hierarchical clustering creates a hierarchy of clusters. Flat clustering is efficient and conceptually simple, but it has a number of drawbacks. These algorithms return a flat unstructured set of clusters, require a prespecified number of clusters as input and are nondeterministic. Hierarchical clustering outputs a hierarchy, a structure that is more informative than the unstructured set of clusters returned by flat clustering. Hierarchical clustering does not require us to prespecify the number of clusters and most hierarchical algorithms that have been used in information retrieval are deterministic. These advantages of hierarchical clustering come at the cost of lower efficiency.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. It is a technique for extracting information from unlabeled data and can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

## Input Ports

**hierarchical *(hie)*** This port expects the hierarchical cluster model. Hierarchical clustering operators like the Agglomerative Clustering operator generate such a model.

**example set *(exa)*** The input port expects an ExampleSet. It is the output of the Agglomerative Clustering operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**flat *(fla)*** This port delivers the flat cluster model which has information regarding the clustering performed. It tells which examples are part of which cluster.

**example set *(exa)*** The ExampleSet that was given as input is passed with minor changes to the output through this port. An attribute with *id* role is added to the input ExampleSet to distinguish examples.

## Parameters

**number of clusters *(integer)*** This parameter specifies the desired number of clusters to form. There is no hard and fast rule to form a number of clusters. But, generally it is preferred to have a small number of clusters with examples scattered (not too scattered) around them in a balanced way.

**add as label *(boolean)*** If true, the cluster id is stored in an attribute with the *label* role instead of *cluster* role.

**remove unlabeled *(boolean)*** If set to true, unlabeled examples are deleted.

## Tutorial Processes

### Flattening the Agglomerative Cluster model

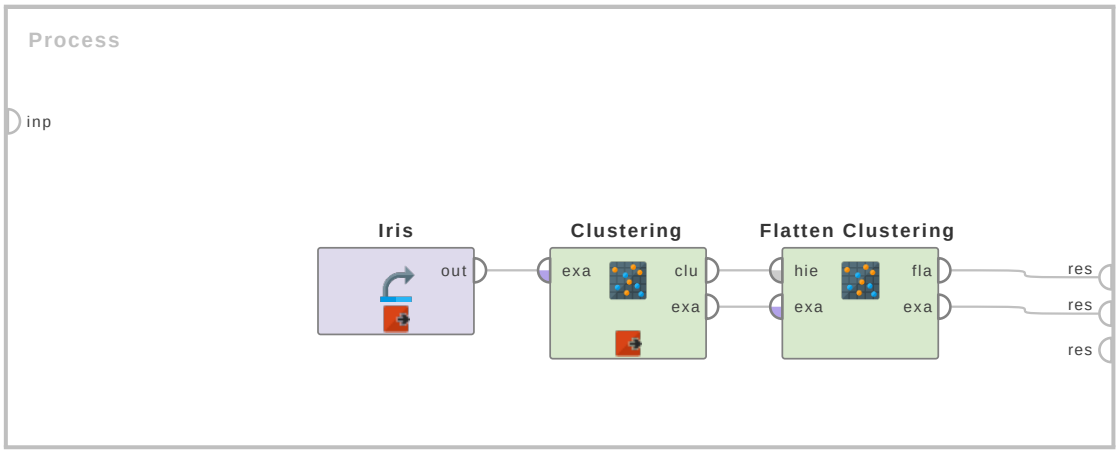


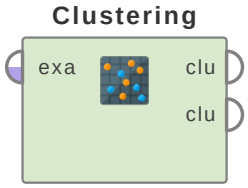

Figure 4.63: Tutorial process ‘Flattening the Agglomerative Cluster model’.

The ‘Iris’ data set is loaded using the Retrieve operator. A breakpoint is inserted at this step so that you can have a look at the ExampleSet. The Agglomerative Clustering operator is applied on this ExampleSet. Run the process and switch to the Results Workspace. Note the Graph View of the results. You can see that the algorithm has not created separate groups or clusters as other clustering algorithms (like k-means), instead the result is a hierarchy of clusters. Under the Folder View you can see members of each cluster in folder format. You can see that it is an hierarchy of folders. The Dendogram View shows the dendrogram for this clustering which shows how single-element clusters were joined step by step to make a hierarchy of clusters. The ExampleSet and the hierarchical cluster model returned by this operator are provided as input to the Flatten Clustering operator.

The Flatten Clustering operator is applied with default values for all parameters. Run the process and you will see that two new attributes are created by the Flatten Clustering operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. As the parameter number of clusters was set to 3, only three clusters are possible. That is why each example is assigned to ‘cluster_0’, ‘cluster_1’ or ‘cluster_2’. Also note the Plot View of this data. You can clearly see how the algorithm has created three separate groups in the Plot View. A cluster model is also delivered through the

cluster model output port. It has information regarding the clustering performed. Under Folder View you can see members of each cluster in folder format.

# Random Clustering




This operator performs a random flat clustering of the given ExampleSet. Clustering is concerned with grouping objects together that are similar to each other and dissimilar to the objects belonging to other clusters.

## Description

This operator performs a random flat clustering of the given ExampleSet. Please note that this algorithm does not guarantee that all clusters will be non-empty. This operator creates a cluster attribute in the resultant ExampleSet if the *add cluster attribute* parameter is set to true. It is important to note that this operator randomly assigns examples to clusters, if you want proper clustering please use an operator that implements a clustering algorithm like the K-Means operator.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. Clustering is a technique for extracting information from unlabeled data. Clustering can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

## Input Ports

**example set *(exa)*** The input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**cluster model *(clu)*** This port delivers the cluster model which has information regarding the clustering performed. It tells which examples are part of which cluster.

**clustered set *(clu)*** The ExampleSet that was given as input is passed with minor changes to the output through this port. An attribute with *id* role is added to the input ExampleSet to distinguish examples. An attribute with *cluster* role may also be added depending on the state of the *add cluster attribute* parameter.

## Parameters

**add cluster attribute *(boolean)*** If enabled, a new attribute with *cluster* role is generated directly in this operator, otherwise this operator does not add the *cluster* attribute. In the latter case you have to use the Apply Model operator to generate the *cluster* attribute.

**add as label *(boolean)*** If true, the cluster id is stored in an attribute with the *label* role instead of *cluster* role (see *add cluster attribute* parameter).

**remove unlabeled *(boolean)*** If set to true, unlabeled examples are deleted.

**number of clusters *(integer)*** This parameter specifies the desired number of clusters to form. There is no hard and fast rule for the number of clusters to form. But, generally it is preferred to have a small number of clusters with examples scattered (not too scattered) around them in a balanced way.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Random clustering of the Ripley-Set data set

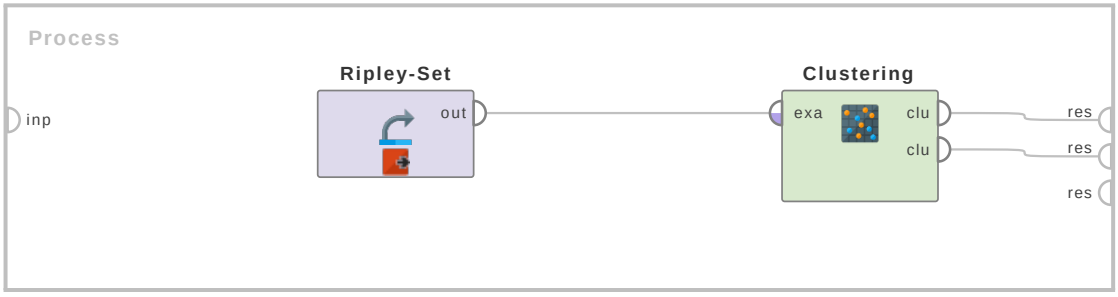
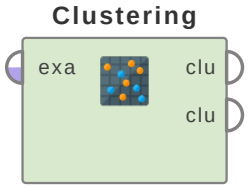



Figure 4.64: Tutorial process ‘Random clustering of the Ripley-Set data set’.

In many cases, no target attribute (i.e. label) can be defined and the data should be automatically grouped. This procedure is called Clustering. RapidMiner supports a wide range of clustering schemes which can be used in just the same way like any other learning scheme. This includes the combination with all preprocessing operators.

In this Example Process, the ‘Ripley-Set’ data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A breakpoint is inserted at this step so that you can have a look at the ExampleSet before the application of the Random Clustering operator. Besides the label attribute the ‘Ripley-Set’ has two real attributes; ‘att1’ and ‘att2’. The Random Clustering operator is applied on this data set with default values for all parameters. Run the process and you will see that two new attributes are created by the Random Clustering operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. As the number of clusters parameter was set to 3, only three clusters are possible. That is why each example is assigned to ‘cluster_0’, ‘cluster_1’ or ‘cluster_2’. Also note the Plot View of this data. You can clearly see how this operator has created three groups in the Plot View. A cluster model is also delivered through the cluster model output port. It has information regarding the clustering performed. Under Folder View you can see members of each cluster in folder format. It is important to note that this operator randomly assigns examples to clusters (this can be seen easily in the Plot View). If you want proper clustering of your ExampleSet please use an operator that implements a clustering algorithm like the K-Means operator.

# Support Vector Clustering




This operator performs clustering with support vectors. Clustering is concerned with grouping objects together that are similar to each other and dissimilar to the objects belonging to other clusters. Clustering is a technique for extracting information from unlabeled data.

## Description

This operator is an implementation of Support Vector Clustering based on Ben-Hur et al (2001). In this Support Vector Clustering (SVC) algorithm data points are mapped from data space to a high dimensional feature space using a Gaussian kernel. In feature space the smallest sphere that encloses the image of the data is searched. This sphere is mapped back to data space, where it forms a set of contours which enclose the data points. These contours are interpreted as cluster boundaries. Points enclosed by each separate contour are associated with the same cluster. As the width parameter of the Gaussian kernel is decreased, the number of disconnected contours in data space increases, leading to an increasing number of clusters. Since the contours can be interpreted as delineating the support of the underlying probability distribution, this algorithm can be viewed as one identifying valleys in this probability distribution.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. It is a technique for extracting information from unlabeled data and can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Generate Data operator in the attached Example Process.

## Output Ports

**cluster model *(clu)*** This port delivers the cluster model. It has information regarding the clustering performed. It tells which examples are part of which cluster.

**clustered set *(clu)*** The ExampleSet that was given as input is passed with minor changes to the output through this port. An attribute with *id* role is added to the input ExampleSet to distinguish examples. An attribute with *cluster* role may also be added depending on the state of the *add cluster attribute* parameter.

## Parameters

**add cluster attribute *(boolean)*** If this parameter is set to true, a new attribute with *cluster* role is generated in the resultant ExampleSet, otherwise this operator does not add the *cluster* attribute. In the latter case you have to use the Apply Model operator to generate the *cluster* attribute.

**add as label *(boolean)*** If this parameter is set to true, the cluster id is stored in an attribute with the *label* role instead of *cluster* role (see *add cluster attribute* parameter).

**remove unlabeled** ***(boolean)*** If this parameter is set to true, unlabeled examples are deleted from the ExampleSet.

**min pts** ***(integer)*** This parameter specifies the minimal number of points in each cluster.

**kernel type** ***(selection)*** The type of the kernel function is selected through this parameter. Following kernel types are supported: *dot, radial, polynomial, neural*

- **dot** The dot kernel is defined by*k(x,y)=x*y* i.e. it is inner product of*x* and *y*.
- **radial** The radial kernel is defined by *exp(-g ||x-y||^2)* where *g* is the *gamma*, it is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by *k(x,y)=(x*y+1)^d* where *d* is the degree of polynomial and it is specified by the *kernel degree* parameter. The polynomial kernels are well suited for problems where all the training data is normalized.
- **neural** The neural kernel is defined by a two layered neural net *tanh(a x*y+b)* where *a* is *alpha* and *b* is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is 1/N, where N is the data dimension. Note that not all choices of *a* and *b* lead to a valid kernel function.

**kernel gamma** ***(real)*** This is the SVM kernel parameter gamma. This is available only when the *kernel type* parameter is set to *radial*.

**kernel degree** ***(real)*** This is the SVM kernel parameter degree. This is available only when the *kernel type* parameter is set to *polynomial*.

**kernel a** ***(real)*** This is the SVM kernel parameter a. This is available only when the *kernel type* parameter is set to *neural*.

**kernel b** ***(real)*** This is the SVM kernel parameter b. This is available only when the *kernel type* parameter is set to *neural*.

**kernel cache** ***(real)*** This is an expert parameter. It specifies the size of the cache for kernel evaluations in megabytes.

**convergence epsilon** ***(real)*** This is an optimizer parameter. It specifies the precision on the KKT conditions.

**max iterations** ***(integer)*** This is an optimizer parameter. It specifies to stop iterations after a specified number of iterations.

**p** ***(real)*** This parameter specifies the fraction of allowed outliers.

**r** ***(real)*** If this parameter is set to -1 then the the calculated radius is used as radius. Otherwise the value specified in this parameter is used as radius.

**number sample points** ***(real)*** This parameter specifies the number of virtual sample points to check for neighborhood.

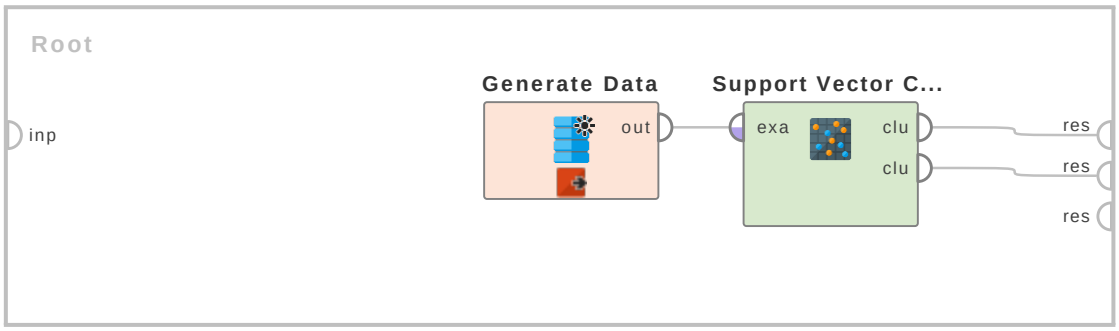



Figure 4.65: Tutorial process 'Clustering of Ripley-Set data set by the Support Vector Clustering operator'.

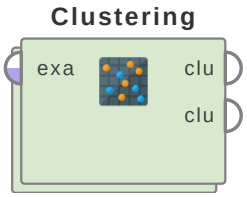
## Tutorial Processes

### Clustering of Ripley-Set data set by the Support Vector Clustering operator

In many cases, no target attribute (i.e. label) can be defined and the data should be automatically grouped. This procedure is called Clustering. RapidMiner supports a wide range of clustering schemes which can be used in just the same way like any other learning scheme. This includes the combination with all preprocessing operators.

In this Example Process, the Generate Data operator is used for generating an ExampleSet. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A breakpoint is inserted at this step so that you can have a look at the ExampleSet before application of the clustering operator. Other than the label attribute the ExampleSet has two real attributes; 'att1' and 'att2'. The Support Vector Clustering operator is applied on this data set. Run the process and you will see that two new attributes are created by the Support Vector Clustering operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. Each example is assigned to a particular cluster. The examples that are not in any cluster are considered as noise. Also note the Plot View of this data set. Switch to Plot View and set the the Plotter to 'Scatter', x-Axis to 'att1', y-Axis to 'att2' and Color Column to 'cluster'. You can clearly see how the algorithm has created three separate cluster (noise is also visible separately). A cluster model is also delivered through the cluster model output port. It has information regarding the clustering performed. Under Folder View you can see members of each cluster in folder format.

# Top Down Clustering




This operator performs top down clustering by applying the inner flat clustering scheme recursively. Top down clustering is a strategy of hierarchical clustering. The result of this operator is an hierarchical cluster model.

## Description

This operator is a nested operator i.e. it has a subprocess. The subprocess must have a flat clustering operator e.g. the K-Means operator. This operator builds a Hierarchical clustering model using the clustering operator provided in its subprocess. You need to have a basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

The basic idea of Top down clustering is that all observations start in one cluster, and splits are performed recursively as one moves down the hierarchy. Top down clustering is a strategy of hierarchical clustering. Hierarchical clustering (also known as Connectivity based clustering) is a method of cluster analysis which seeks to build a hierarchy of clusters. Hierarchical clustering, is based on the core idea of objects being more related to nearby objects than to objects farther away. As such, these algorithms connect 'objects' (or examples, in case of an ExampleSet) to form clusters based on their distance. A cluster can be described largely by the maximum distance needed to connect parts of the cluster. At different distances, different clusters will form. These algorithms do not provide a single partitioning of the data set, but instead provide an extensive hierarchy of clusters that merge with each other at certain distances.

Strategies for hierarchical clustering generally fall into two types:

- Agglomerative: This is a bottom-up approach: each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy. This type of clustering is implemented in RapidMiner as the Agglomerative Clustering operator.
- Divisive: This is a top-down approach: all observations start in one cluster, and splits are performed recursively as one moves down the hierarchy.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. It is a technique for extracting information from unlabeled data and can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

## Output Ports

**cluster model *(clu)*** This port delivers the hierarchical cluster model. It has information regarding the clustering performed.

**clustered set *(clu)*** The ExampleSet that was given as input is passed with minor changes to the output through this port. An attribute with *id* role is added to the input ExampleSet to distinguish examples. An attribute with *cluster* role may also be added depending on the state of the *add cluster label* parameter.

## Parameters

**create cluster label *(boolean)*** This parameter specifies if a cluster label should be created. If this parameter is set to true, a new attribute with *cluster* role is generated in the resultant ExampleSet, otherwise this operator does not add the *cluster* attribute.

**max depth *(integer)*** This parameter specifies the maximal depth of the cluster tree.

**max leaf size *(integer)*** This parameter specifies the maximal number of items in each cluster leaf.

## Tutorial Processes

### Top down clustering of Ripley-Set data set

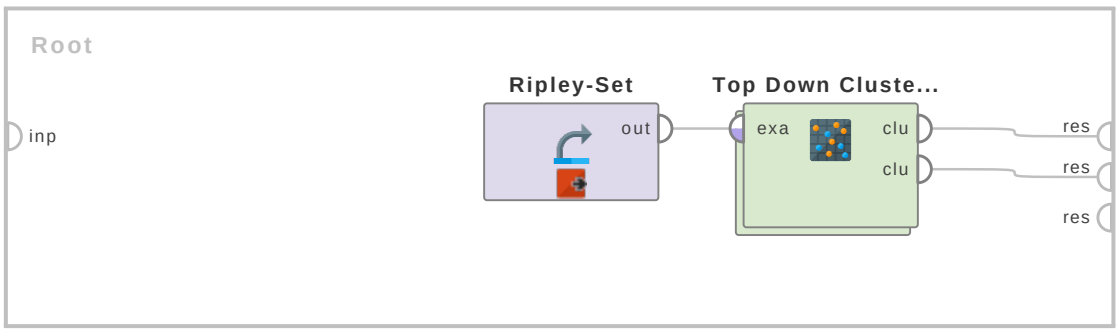
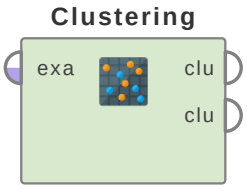



Figure 4.66: Tutorial process 'Top down clustering of Ripley-Set data set'.

The 'Ripley-Set' data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A breakpoint is inserted at this step so that you can have a look at the ExampleSet before application of the Top Down Clustering operator. Other than the label attribute the 'Ripley-Set' has two real attributes; 'att1' and 'att2'. The Top Down Clustering operator is applied on this data set. Run the process and you will see that two new attributes are created by the Top Down Clustering operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. Each example is assigned to a particular cluster. Note the Graph View of the results. You can see that the algorithm has not created separate groups or clusters as other clustering algorithms (like k-means), instead the result is a hierarchy of clusters. Under the Folder View you can see members of each cluster in folder format. You can see that it is an hierarchy of folders.

# k-Means




This Operator performs clustering using the *k-means* algorithm.

## Description

This Operator performs clustering using the *k-means* algorithm. Clustering groups Examples together which are similar to each other. As no *Label* Attribute is necessary, Clustering can be used on unlabelled data and is an algorithm of unsupervised machine learning.

The k-means algorithm determines a set of *k* clusters and assignes each Examples to exact one cluster. The clusters consist of similar Examples. The similarity between Examples is based on a distance measure between them.

A cluster in the k-means algorithm is determined by the position of the center in the n-dimensional space of the n Attributes of the ExampleSet. This position is called centroid. It can, but do not have to be the position of an Example of the ExampleSets.

The k-means algorithm starts with *k* points which are treated as the centroid of *k* potential clusters. These start points are either the position of *k* randomly drawn Examples of the input ExampleSet, or are determined by the k-means++ heuristic if *determine good start values* is set to true.

All Examples are assigned to their nearest cluster (nearest is defined by the *measure type*). Next the centroids of the clusters are recalculated by averaging over all Examples of one cluster. The previous steps are repeated for the new centroids until the the centroids no longer move or *max optimization steps* is reached. Be aware that it is not ensured that the k-means algorithm converges if the measure type is not based on Euclidean Distance calculation (cause the recalculation of the centroids by averaging is assuming Euclidean space).

The procedure is repeated *max runs* times with each time a different set of start points. The set of clusters is delivered which has the minimal sum of squared distances of all Examples to their corresponding centroids.

## Differentiation

- **k-Medoids**

  In case of the k-medoids algorithm the centroid of a cluster will always be one of the points in the cluster. This is the major difference between the k-means and k-medoids algorithm.

  See page 591 for details.

- **k-Means (Kernel)**

  Kernel k-means uses kernels to estimate distances between Examples and clusters. Because of the nature of kernels it is necessary to sum over all Examples of a cluster to calculate one distance. So this algorithm is quadratic in number of Examples and does not return a Centroid Cluster Model (on the contrary the K-Means operator returns a Centroid Cluster Model).

  See page 587 for details.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet.

## Output Ports

**cluster model *(clu)*** This port delivers the cluster model. It contains the information which Examples are part of which cluster. It also stores the position of the centroids of the clusters.

It can be used by the Apply Model Operator to perform the specified clustering on another ExampleSet.

The cluster model can also be grouped together with other clustering models, preprocessing models and learning models by the Group Models Operator.

**clustered set *(clu)*** An Attribute 'id' with special role 'Id' is added to the input ExampleSet to distinguish Examples. Depending on the *add cluster attribute* and the *add as label* parameters an Attribute 'cluster' with special role 'Cluster' or 'Label' is also added. The resulting ExampleSet is delivered at this output port.

## Parameters

**add cluster attribute** If true, a new Attribute called 'cluster' with the cluster_id for each Example is generated. By default the Attribute is created with the special role 'Cluster', except the parameter *add as label* is true. If so the new Attribute is called 'label' and has the special role 'Label'. The Attribute can also be generated later by using the Apply Model Operator.

**add as label** If true the new Attribute with the cluster_id is called 'label' and has the special role 'Label'. If the parameter *add cluster attribute* is false, no new Attribute is created.

**remove unlabeled** If set to true, Examples which cannot be assigned to a cluster are removed from the output ExampleSet.

**k** This parameter specifies the number of clusters to determine.

**max runs** This parameter specifies the maximal number of runs of k-Means with random initialization of the start points that are performed.

**determine good start values** If true the *k* start points are determined using the k-means++ heuristic, descriped in "k-means++: The Advantages of Careful Seeding" by David Arthur and Sergei Vassilvitskii 2007.

**max optimization steps** This parameter specifies the maximal number of iterations performed for one run of k-Means.

**measure types** This parameter is used for selecting the type of measure to be used for finding the nearest neighbors. The following options are available:

- **MixedMeasures** Mixed Measures are used to calculate distances in case of both nominal and numerical Attributes.
- **NominalMeasures** In case of only nominal Attributes different distance metrices can be used to calculate distances on this nominal Attributes.

- **NumericalMeasures** In case of only numerical Attributes different distance metrices can be used to calculate distances on this numerical Attributes.
- **BregmannDivergences** Bregmann divergences are more generic "closeness" measure types with does not satisfy the triangle inequality or symmetry. For more details see the parameter *divergence*.

**mixed measure** The only available option for *mixed measure* is the 'Mixed Euclidean Distance'. For numerical values the euclidean distance is calculated. For nomimal values, a distance of 0 is taken if both values are the same and a distance of one is taken otherwise. This parameter is available when the *measure type* parameter is set to 'mixed measures'.

**nominal measure** This parameters defines how to calculate distances for only nominal Attributes in the input ExampleSet, in case the *measure type* is set to nominal measure. In case of using a similarity as a distance measure, the actual distance is calculated as the negative similarity. For the different similarities the following variables are defined:

e: number of Attribute for which both Examples have equal and non-zero values

u: number of Attribute for which both Examples have not equal values

z: number of Attribute for which both Examples have zero values

- **NominalDistance** Distance of two values is 0 if both values are the same and 1 otherwise.
- **DiceSimilarity** With the above mentioned definitions the *DiceSimilarity* is: 2*e/(2*e+u)
- **JaccardSimilarity** With the above mentioned definitions the *JaccardSimilarity* is: e/(e+u)
- **KulczynskiSimilarity** With the above mentioned definitions the *KulczynskiSimilarity* is: e/u
- **RogersTanimotoSimilarity** With the above mentioned definitions the *RogersTanimotoSimilarity* is: (e+z)/(e+2*u+z)
- **RussellRaoSimilarity** With the above mentioned definitions the *RussellRaoSimilarity* is: e/(e+u+z)
- **SimpleMatchingSimilarity** With the above mentioned definitions the *SimpleMatchingSimilarity* is: (e+z)/(e+u+z)

**numerical measure** This parameters defines how to calculate distances for only numerical Attributes in the input ExampleSet, in case the *measure type* is set to numerical measure. For the different distance measures the following variable is defined:

y(i,j) : Value of the j.th Attribute of the i.th Example. Hence y(1,3) - y(2,3) is the difference of the values of the third Attribute of the first and second Example.

In case of using a similarity as a distance measure, the actual distance is calculated as the negative similarity.

- **EuclideanDistance** Square root of the sum of quadratic differences over all Attributes. Dist = Sqrt ( Sum_(j=1) [y(1,j)-y(2,j)]^2 )
- **CanberraDistance** Sum over all Attributes. The summand is the absolute of the difference of the value, divided by the sum of the absolute values. Dist = Sum_(j=1) |y(1,j)-y(2,j)| / (|y(1,j)|+|y(2,j)|) The CanberraDistance is often used to compare ranked list or for intrusion detection in computer security.
- **ChebychevDistance** Maximum of all differences of all Attributes. Dist = max_(j=1) (|y(1,j)-y(2,j)|)

- **CorrelationSimilarity** The similarity is calculated as the correlation between the Attribute vectors of the two Examples.
- **CosineSimilarity** Similarity measure measuring the cosine of the angle between the Attribute vectors of the two Examples.
- **DiceSimilarity** The DiceSimilarity for numerical Attributes is calculated as 2*Y1Y2/(Y1+Y2). Y1Y2 = Sum over product of values = Sum_(j=1) y(1,j)*y(2,j). Y1 = Sum over values of first Example = Sum_(j=1) y(1,j) Y2 = Sum over values of second Example = Sum_(j=1) y(2,j)
- **DynamicTimeWarpingDistance** Dynamic Time Warping is often use in Time Series analysis for measuring the distance between two temporal sequences. Here the distance on an optimal "warping" path from the Attribute vector of the first Example to the second Example is calculated.
- **InnerProductSimilarity** The similarity is calculated as the sum of the product of the Attribute vectors of the two Examples. Dist = -Similarity = -Sum_(j=1) y(1,j)*y(2,j)
- **JaccardSimilarity** The JaccardSimilarity is calculated as Y1Y2/(Y1+Y2-Y1Y2). See *DiceSimilarity* for the definition of Y1Y2, Y1 and Y2.
- **KernelEuclideanDistance** The distance is calculated by the euclidean distance of the two Examples, in a transformed space. The transformation is defined by the chosen kernel and configured by the parameters *kernel type, gamma, sigma1, sigma2, sigma 3, shift, degree, a, b*.
- **ManhattanDistance** Sum of the absolute distances of the Attribute values. Dist = Sum_(j=1) |y(1,j)-y(2,j)|
- **MaxProductSimilarity** The similarity is the maximum of all products of all Attribute values. If the maximum is less or equal to zero the similarity is not defined. Dist = -Similarity = -max_(j=1) (y(1,j)*y(2,j))
- **OverlapSimilarity** The similarity is a variant of simple matching for numerical Attributes and is calculated as minY1Y2 / min(Y1,Y2). See *DiceSimilarity* for the definition of Y1, Y2. minY1Y2 = Sum over the minimum of values = Sum_(j=1) min [y(1,j),y(2,j)] .

**divergence** This parameter defines which type of Bregman divergence is used when the *measure type* parameter is set to 'bregman divergences'. For the different distance measures the following variable is defined:

y(i,j) : Value of the j.th Attribute of the i.th Example. Hence y(1,3) - y(2,3) is the difference of the values of the third Attribute of the first and second Example.

- **GeneralizedIDivergence** The distance is calculated as Sum1 Sum2. It is not applicable if any Attribute value is less or equal to 0. Sum1 = Sum_(j=1) y(1,j)*ln[y(1,j)/y(2,j)] Sum2 = Sum_(j=1) [y(1,j)-y(2,j)]
- **ItakuraSaitoDistance** The ItakuraSaitoDistance can only be calculated for ExampleSets with 1 Attribute and values larger 0. Dist = y(1,1)/y(2,1)-ln[y(1,1)/y(2,1)]-1
- **KLDivergence** The Kullback-Leibler divergence is a measure of how one probability distribution diverges from a second expected probability distribution. Dist = Sum_(j=1) [y(1,j)*log_2(y(1,j)/y(2,j))]
- **LogarithmicLoss** The LogarithmicLoss can only be calculated for ExampleSets with 1 Attribute and values larger 0. Dist = y(1,1)*ln[y(1,1)/y(2,1)]-(y(1,1)-y(2,1))

- **LogisticLoss** The LogisticLoss can only be calculated for ExampleSets with 1 Attribute and values larger 0. Dist = y(1,1)*ln[y(1,1)/y(2,1)]+(1-y(1,1))*ln[(1-y(1,1))/(1-y(2,1))]
- **MahalanobisDistance** The Mahalanobis distance measures the distance between the two Examples under the assumption they are both random vectors of the same distribution. Therefore the covariance matrix S is calculated on the whole ExampleSet and the Distance is calculated as: Dist = Sqrt [ (vecY1-vecY2) S (vecY1-vecY2) ] vecY1 = Attribute vector of Example 1 vecY2 = Attribute vector of Example 2
- **SquaredEuclideanDistance** Sum of quadratic differences over all Attributes. Dist = Sum_(j=1) [y(1,j)-y(2,j)]^2
- **SquaredLoss** The SquaredLoss can only be calculated for ExampleSets with 1 Attribute. Dist = [y(1,1)-y(2,1)]^2

**kernel type** This parameter is available only when the *numerical measure* parameter is set to 'Kernel Euclidean Distance'. The type of the kernel function is selected through this parameter. Following kernel types are supported:

- **dot** The dot kernel is defined by k(x,y) = x*y i.e. it is the inner product of x and y.
- **radial** The radial kernel is defined by k(x,y) = exp(-g*||x-y||^2) where g is gamma, specified by the *kernel gamma* parameter. The adjustable parameter gamma plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by k(x,y) = (x*y+1)^d where d is the degree of the polynomial and is specified by the _kernel degree_ parameter. Polynomial kernels are well suited for problems where all the training data is normalized.
- **sigmoid** This is a hyperbolic tangent sigmoid kernel. The distance is calculated as tanh[a*Y1Y2+b] where Y1Y2 is the inner product of the Attribute vector of the two Examples. a and b can be adjusted using the *kernel a* and *kernel b* parameters. A common value for a is 1/N, where N is the data dimension. Note that not all choices of a and b lead to a valid kernel function.
- **anova** The anova kernel is defined by the raised to the power d of summation of exp(-g(x-y)) where g is gamma and d is degree. The two are adjusted by the *kernel gamma* and *kernel degree* parameters respectively.
- **epachnenikov** The Epanechnikov kernel is this function (3/4)(1-u2) for u between -1 and 1 and zero for u outside that range. It has the two adjustable parameters kernel sigma1 and kernel degree.
- **gaussian_combination** This is the gaussian combination kernel. It has the adjustable parameters kernel sigma1, kernel sigma2 and kernel sigma3.
- **multiquadric** The multiquadric kernel is defined by the square root of ||x-y||^2+c^2. It has the adjustable parameters kernel sigma1 and kernel sigma shift.

**kernel gamma** This is the SVM kernel parameter gamma. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1** This is the SVM kernel parameter sigma1. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2** This is the SVM kernel parameter sigma2. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3** This is the SVM kernel parameter sigma3. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel shift** This is the SVM kernel parameter shift. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *multiquadric*.

**kernel degree** This is the SVM kernel parameter degree. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a** This is the SVM kernel parameter a. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

**kernel b** This is the SVM kernel parameter b. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

**use local random seed** This parameter indicates if a *local random seed* should be used for randomization of the *k* different starting points of the algorithm.

**local random seed** If the *use local random seed* parameter is checked this parameter determines the local random seed.

## Tutorial Processes

### Clustering of the Iris Data Set

In this tutorial process the Iris data set is clustered using the k-means Operator. The Iris data set is retrieved from the Samples folder. Also the label Attribute is also retrieved, but only for comparison of the cluster assignment with the class of the Examples. The label Attribute is not used in the Clustering. For more details see the comments in the Process.

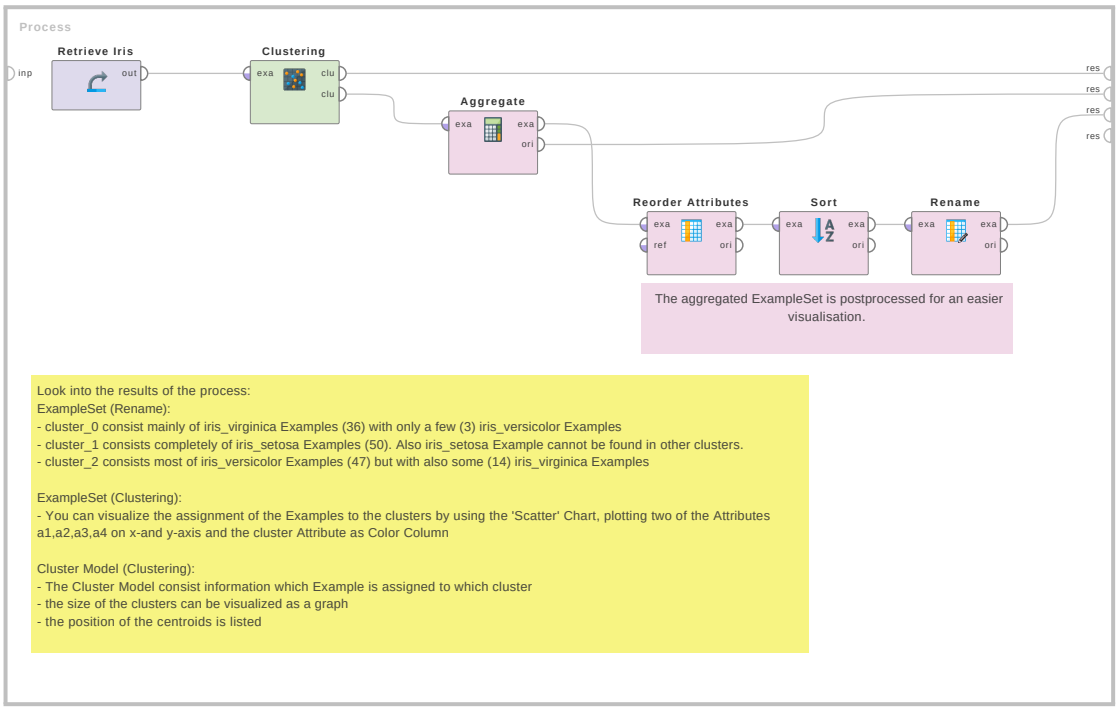
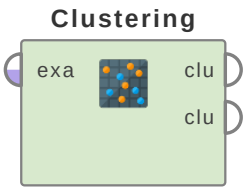



Figure 4.67: Tutorial process ‘Clustering of the Iris Data Set’.

# K-Means (Kernel)




This operator performs clustering using the kernel k-means algorithm. Clustering is concerned with grouping objects together that are similar to each other and dissimilar to the objects belonging to other clusters. Kernel k-means uses kernels to estimate the distance between objects and clusters. K-means is an exclusive clustering algorithm.

## Description

This operator performs clustering using the kernel k-means algorithm. The k-means is an exclusive clustering algorithm i.e. each object is assigned to precisely one of a set of clusters. Objects in one cluster are similar to each other. The similarity between objects is based on a measure of the distance between them. Kernel k-means uses kernels to estimate the distance between objects and clusters. Because of the nature of kernels it is necessary to sum over all elements of a cluster to calculate one distance. So this algorithm is quadratic in number of examples and does not return a Centroid Cluster Model contrary to the K-Means operator. This operator creates a cluster attribute in the resultant ExampleSet if the *add cluster attribute* parameter is set to true.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. Clustering is a technique for extracting information from unlabeled data. Clustering can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

## Differentiation

- **k-Means (Deprecated)** Kernel k-means uses kernels to estimate the distance between objects and clusters. Because of the nature of kernels it is necessary to sum over all elements of a cluster to calculate one distance. So this algorithm is quadratic in number of examples and does not return a Centroid Cluster Model which does the K-Means operator. See page **??** for details.

## Input Ports

**example set *(exa)*** The input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**cluster model *(clu)*** This port delivers the cluster model which has information regarding the clustering performed. It tells which examples are part of which cluster.

**clustered set *(clu)*** The ExampleSet that was given as input is passed with minor changes to the output through this port. An attribute with *id* role is added to the input ExampleSet to distinguish examples. An attribute with *cluster* role may also be added depending on the state of the *add cluster attribute* parameter.

## Parameters

**add cluster attribute *(boolean)*** If enabled, a new attribute with *cluster* role is generated directly in this operator, otherwise this operator does not add the *cluster* attribute. In the latter case you have to use the Apply Model operator to generate the *cluster* attribute.

**add as label *(boolean)*** If true, the cluster id is stored in an attribute with the *label* role instead of *cluster* role (see *add cluster attribute* parameter).

**remove unlabeled *(boolean)*** If set to true, unlabeled examples are deleted.

**use weights *(boolean)*** This parameter indicates if the weight attribute should be used.

**k *(integer)*** This parameter specifies the number of clusters to form. There is no hard and fast rule of number of clusters to form. But, generally it is preferred to have small number of clusters with examples scattered (not too scattered) around them in a balanced way.

**max optimization steps *(integer)*** This parameter specifies the maximal number of iterations performed for one run of k-Means

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

**kernel type *(selection)*** The type of the kernel function is selected through this parameter. Following kernel types are supported: *dot, radial, polynomial, neural, anova, epachnenikov, gaussian combination, multiquadric*

- **dot** The dot kernel is defined by$k(x,y)=x*y$ i.e. it is inner product of$x$ and $y$.

- **radial** The radial kernel is defined by *exp(-g ||x-y||^2)* where *g* is the *gamma*, it is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by *k(x,y)=(x*y+1)^d* where *d* is the degree of polynomial and it is specified by the *kernel degree* parameter. The polynomial kernels are well suited for problems where all the training data is normalized.
- **neural** The neural kernel is defined by a two layered neural net *tanh(a x*y+b)* where *a* is *alpha* and *b* is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is 1/N, where N is the data dimension. Note that not all choices of *a* and *b* lead to a valid kernel function.
- **anova** The anova kernel is defined by raised to power *d* of summation of *exp(-g (x-y))* where *g* is *gamma* and *d* is *degree*. gamma and degree are adjusted by the *kernel gamma* and *kernel degree* parameters respectively.
- **epachnenikov** The epachnenikov kernel is this function *(3/4)(1-u2)* for *u* between -1 and 1 and zero for *u* outside that range. It has two adjustable parameters *kernel sigma1* and *kernel degree*.
- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters *kernel sigma1, kernel sigma2* and *kernel sigma3*.
- **multiquadric** The multiquadric kernel is defined by the square root of *||x-y||^2 + c^2*. It has adjustable parameters *kernel sigma1* and *kernel sigma shift*.

**kernel gamma *(real)*** This is the kernel parameter gamma. This is only available when the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1 *(real)*** This is the kernel parameter sigma1. This is only available when the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2 *(real)*** This is the kernel parameter sigma2. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3 *(real)*** This is the kernel parameter sigma3. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel shift *(real)*** This is the kernel parameter shift. This is only available when the *kernel type* parameter is set to *multiquadric*.

**kernel degree *(real)*** This is the kernel parameter degree. This is only available when the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a *(real)*** This is the kernel parameter a. This is only available when the *kernel type* parameter is set to *neural*.

**kernel b *(real)*** This is the kernel parameter b. This is only available when the *kernel type* parameter is set to *neural*.

## Related Documents

- **k-Means (Deprecated)** (page **??**)

## Tutorial Processes

### Clustering of the Ripley-Set data set using the Kernel K-Means operator

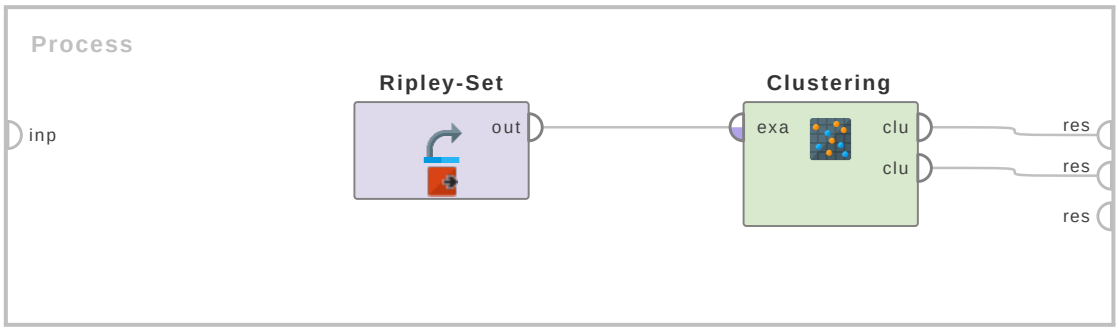
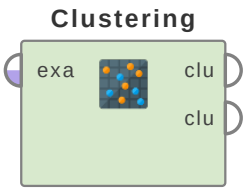



Figure 4.68: Tutorial process ‘Clustering of the Ripley-Set data set using the Kernel K-Means operator’.

In many cases, no target attribute (i.e. label) can be defined and the data should be automatically grouped. This procedure is called Clustering. RapidMiner supports a wide range of clustering schemes which can be used in just the same way like any other learning scheme. This includes the combination with all preprocessing operators.

In this Example Process, the ‘Ripley-Set’ data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A breakpoint is inserted at this step so that you can have a look at the ExampleSet before application of the Kernel K-Means operator. Besides the label attribute the ‘Ripley-Set’ has two real attributes; ‘att1’ and ‘att2’. The Kernel K-Means operator is applied on this data set with default values for all parameters. Run the process and you will see that two new attributes are created by the Kernel K-Means operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. As parameter k was set to 2, only two clusters are possible. That is why each example is assigned to either ‘cluster_0’ or ‘cluster_1’. Also note the Plot View of this data. You can clearly see how the algorithm has created two separate groups in the Plot View. A cluster model is also delivered through the cluster model output port. It has information regarding the clustering performed. Under Folder View you can see members of each cluster in folder format.

# K-Medoids




This operator performs clustering using the *k-medoids* algorithm. Clustering is concerned with grouping objects together that are similar to each other and dissimilar to the objects belonging to other clusters. Clustering is a technique for extracting information from unlabelled data. k-medoids clustering is an exclusive clustering algorithm i.e. each object is assigned to precisely one of a set of clusters.

## Description

This operator performs clustering using the k-medoids algorithm. K-medoids clustering is an exclusive clustering algorithm i.e. each object is assigned to precisely one of a set of clusters. Objects in one cluster are similar to each other. The similarity between objects is based on a measure of the distance between them.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. It is a technique for extracting information from unlabeled data and can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

Here is a simple explanation of how the k-medoids algorithm works. First of all we need to introduce the notion of the center of a cluster, generally called its centroid. Assuming that we are using Euclidean distance or something similar as a measure we can define the centroid of a cluster to be the point for which each attribute value is the average of the values of the corresponding attribute for all the points in the cluster. The centroid of a cluster will always be one of the points in the cluster. This is the major difference between the k-means and k-medoids algorithm. In the k-means algorithm the centroid of a cluster will frequently be an imaginary point, not part of the cluster itself, which we can take to mark its center. For more information about the k-means algorithm please study the k-means operator.

## Differentiation

- **k-Means (Deprecated)** In case of the k-medoids algorithm the centroid of a cluster will always be one of the points in the cluster. This is the major difference between the k-means and k-medoids algorithm. In the k-means algorithm the centroid of a cluster will frequently be an imaginary point, not part of the cluster itself, which we can take to mark its center. See page **??** for details.

## Input Ports

**example set input *(exa)*** The input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**cluster model *(clu)*** This port delivers the cluster model. It has information regarding the clustering performed. It tells which examples are part of which cluster. It also has information regarding centroids of each cluster.

**clustered set** ***(clu)*** The ExampleSet that was given as input is passed with minor changes to the output through this port. An attribute with *id* role is added to the input ExampleSet to distinguish examples. An attribute with *cluster* role may also be added depending on the state of the *add cluster attribute* parameter.

## Parameters

**add cluster attribute** ***(boolean)*** If enabled, a new attribute with *cluster* role is generated directly in this operator, otherwise this operator does not add the *cluster* attribute. In the latter case you have to use the Apply Model operator to generate the *cluster* attribute.

**add as label** ***(boolean)*** If true, the cluster id is stored in an attribute with the *label* role instead of *cluster* role (see *add cluster attribute* parameter).

**remove unlabeled** ***(boolean)*** If set to true, unlabeled examples are deleted.

**k** ***(integer)*** This parameter specifies the number of clusters to form. There is no hard and fast rule of number of clusters to form. But, generally it is preferred to have a small number of clusters with examples scattered (not too scattered) around them in a balanced way.

**max runs** ***(integer)*** This parameter specifies the maximal number of runs of k-medoids with random initialization that are performed.

**max optimization steps** ***(integer)*** This parameter specifies the maximal number of iterations performed for one run of k-medoids.

**use local random seed** ***(boolean)*** Indicates if a *local random seed* should be used for randomization. Randomization may be used for selecting *k* different points at the start of the algorithm as potential centroids.

**local random seed** ***(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

**measure types** ***(selection)*** This parameter is used for selecting the type of measure to be used for measuring the distance between points.The following options are available: *mixed measures*, *nominal measures*, *numerical measures* and *Bregman divergences*.

**mixed measure** ***(selection)*** This parameter is available when the *measure type* parameter is set to 'mixed measures'. The only available option is the 'Mixed Euclidean Distance'

**nominal measure** ***(selection)*** This parameter is available when the *measure type* parameter is set to 'nominal measures'. This option cannot be applied if the input ExampleSet has numerical attributes. In this case the 'numerical measure' option should be selected.

**numerical measure** ***(selection)*** This parameter is available when the *measure type* parameter is set to 'numerical measures'. This option cannot be applied if the input ExampleSet has nominal attributes. If the input ExampleSet has nominal attributes the 'nominal measure' option should be selected.

**divergence** ***(selection)*** This parameter is available when the *measure type* parameter is set to 'bregman divergences'.

**kernel type** ***(selection)*** This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance'. The type of the kernel function is selected through this parameter. Following kernel types are supported:

- **dot** The dot kernel is defined by*k(x,y)=x*y* i.e.it is inner product of*x* and *y*.
- **radial** The radial kernel is defined by *exp(-g ||x-y||^2)* where *g* is the *gamma* that is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by *k(x,y)=(x*y+1)^d* where *d* is the degree of the polynomial and it is specified by the *kernel degree* parameter. The Polynomial kernels are well suited for problems where all the training data is normalized.
- **neural** The neural kernel is defined by a two layered neural net *tanh(a x*y+b)* where *a* is *alpha* and *b* is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is 1/N, where N is the data dimension. Note that not all choices of *a* and *b* lead to a valid kernel function.
- **sigmoid** This is the sigmoid kernel. Please note that the *sigmoid* kernel is not valid under some parameters.
- **anova** This is the anova kernel. It has adjustable parameters *gamma* and *degree*.
- **epachnenikov** The Epanechnikov kernel is this function *(3/4)(1-u2)* for *u* between -1 and 1 and zero for *u* outside that range. It has two adjustable parameters *kernel sigma1* and *kernel degree*.
- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters *kernel sigma1, kernel sigma2* and *kernel sigma3*.
- **multiquadric** The multiquadric kernel is defined by the square root of *||x-y||^2 + c^2*. It has adjustable parameters *kernel sigma1* and *kernel sigma shift*.

**kernel gamma *(real)*** This is the SVM kernel parameter gamma. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1 *(real)*** This is the SVM kernel parameter sigma1. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2 *(real)*** This is the SVM kernel parameter sigma2. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3 *(real)*** This is the SVM kernel parameter sigma3. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel shift *(real)*** This is the SVM kernel parameter shift. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *multiquadric*.

**kernel degree *(real)*** This is the SVM kernel parameter degree. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a *(real)*** This is the SVM kernel parameter a. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

**kernel b *(real)*** This is the SVM kernel parameter b. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

## Related Documents

- **k-Means (Deprecated)** (page **??**)

## Tutorial Processes

### Clustering of Ripley-Set data set by the K-Medoids operator

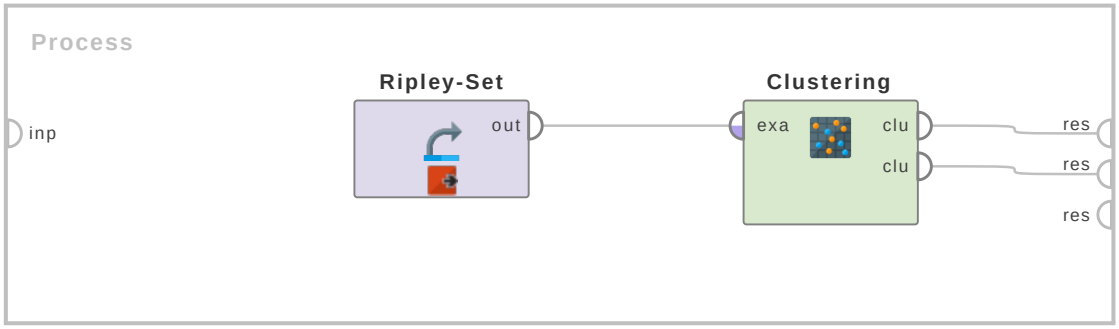


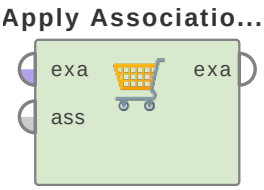

Figure 4.69: Tutorial process 'Clustering of Ripley-Set data set by the K-Medoids operator'.

In many cases, no target attribute (i.e. label) can be defined and the data should be automatically grouped. This procedure is called Clustering. RapidMiner supports a wide range of clustering schemes which can be used in just the same way like any other learning scheme. This includes the combination with all preprocessing operators.

In this Example Process, the 'Ripley-Set' data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A breakpoint is inserted at this step so that you can have a look at the ExampleSet before application of the K-Medoids operator. Other than the label attribute the 'Ripley-Set' has two real attributes; 'att1' and 'att2'. The K-Medoids operator is applied on this data set with default values for all parameters. Run the process and you will see that two new attributes are created by the K-Medoids operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. As parameter k was set to 2, only two clusters are possible. That is why each example is assigned to either 'cluster_0' or 'cluster_1'. Also note the Plot View of this data. You can clearly see how the algorithm has created two separate groups in the Plot View. A cluster model is also delivered through the cluster model output port. It has information regarding the clustering performed. Under Folder View you can see members of each cluster in folder format. You can see information regarding centroids under the Centroid Table and Centroid Plot View tabs.

# 4.3 Associations

## Apply Association Rules




This operator applies the given association rules on an ExampleSet.

### Description

This operator creates a new confidence attribute for each item occurring in at least one conclusion of an association rule. Then it checks for each example and for each rule, if the example fulfills the premise of the rule, which it does, if it covers all items in the premise. An example covers an item, if the attribute representing the item contains the positive value. If the check is positive, a confidence value for each item in the conclusion is derived. Which value is used, depends on the selected confidence aggregation method. There are two types: The binary choice will set a 1, for any item contained inside a fulfilled rule's conclusion. This is independent of how confident the rule was. Any aggregation choice will select the maximum of the previous and the new value of the selected confidence function.

Association rules are if/then statements that help uncover relationships between seemingly unrelated data. An example of an association rule would be "If a customer buys eggs, he is 80% likely to also purchase milk." An association rule has two parts, an antecedent (if) and a consequent (then). An antecedent or premise is an item (or itemset) found in the data. A consequent or conclusion is an item (or itemset) that is found in combination with the antecedent.

Association Rules can be created by using the Create Association Rules operator. Association rules are created by analyzing data for frequent if/then patterns and using the criteria *support* and *confidence* to identify the most important relationships. Support is an indication of how frequently the items appear in the database. Confidence indicates the number of times the if/then statements have been found to be true. The frequent if/then patterns are mined using the operators like the FP-Growth operator. The Create Association Rules operator takes these frequent itemsets and generates association rules.

Such information can be used as the basis for decisions about marketing activities such as, e.g., promotional pricing or product placements. In addition to the above example from market basket analysis association rules are employed today in many application areas including Web usage mining, intrusion detection and bioinformatics.

### Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Subprocess operator in the attached Example Process.

**association rules *(ass)*** This input port expects association rules.

### Output Ports

**example set *(exa)*** The association rules are applied and the resultant ExampleSet is output of this port.

## Parameters

**confidence aggregation method *(selection)*** This parameter selects the method for aggregation of the confidence on the items in each fulfilled conclusion.

**positive value *(string)*** This parameter determines, which value of the binominal attributes is treated as positive. Attributes with that value are considered as part of a transaction. If left blank, the ExampleSet determines which value to use.

## Tutorial Processes

### Applying association rules

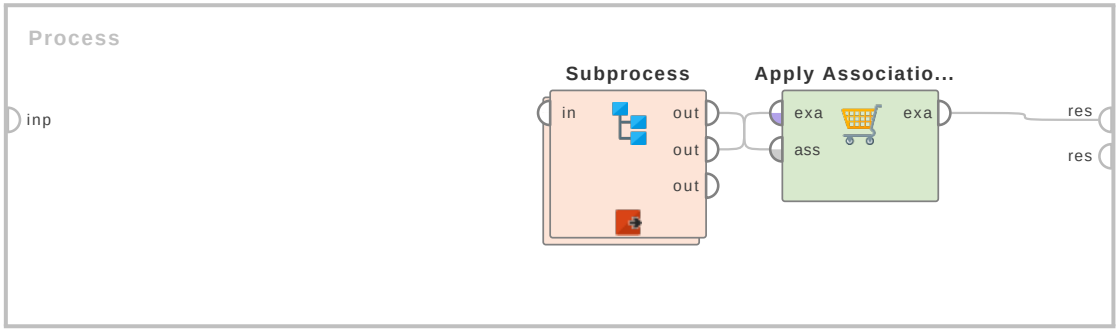
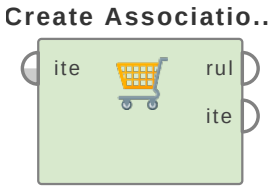



Figure 4.70: Tutorial process 'Applying association rules'.

This Example Process starts with the Subprocess operator which provides an ExampleSet and Association Rules. A breakpoint is inserted here so that you can view the ExampleSet and the Association Rules. This Example Process deals with the application of these rules. If you want to know how these association rules were created, please study the Example Process of the Create Association Rules operator. The ExampleSet and Association Rules are provided as input to the Apply Association Rules operator. All parameters of the Apply Association Rules operator are used with default values. The resultant ExampleSet can be viewed in the Results Workspace. You can see that this operator has created several confidence attributes in the ExampleSet. The explanation of these confidence attributes is given in the description of this operator.

# Create Association Rules




This operator generates a set of association rules from the given set of frequent itemsets.

## Description

Association rules are if/then statements that help uncover relationships between seemingly unrelated data. An example of an association rule would be "If a customer buys eggs, he is 80% likely to also purchase milk." An association rule has two parts, an antecedent (if) and a consequent (then). An antecedent is an item (or itemset) found in the data. A consequent is an item (or itemset) that is found in combination with the antecedent.

Association rules are created by analyzing data for frequent if/then patterns and using the criteria *support* and *confidence* to identify the most important relationships. Support is an indication of how frequently the items appear in the database. Confidence indicates the number of times the if/then statements have been found to be true. The frequent if/then patterns are mined using the operators like the FP-Growth operator. The Create Association Rules operator takes these frequent itemsets and generates association rules.

Such information can be used as the basis for decisions about marketing activities such as, e.g., promotional pricing or product placements. In addition to the above example from market basket analysis association rules are employed today in many application areas including Web usage mining, intrusion detection and bioinformatics.

## Input Ports

**item sets *(ite)*** This input port expects frequent itemsets. Operators like the FP-Growth operator can be used for providing these frequent itemsets.

## Output Ports

**item sets *(ite)*** The itemsets that was given as input is passed without changing to the output through this port. This is usually used to reuse the same itemsets in further operators or to view the itemsets in the Results Workspace.

**rules *(rul)*** The association rules are delivered through this output port.

## Parameters

**criterion *(selection)*** This parameter specifies the criterion which is used for the selection of rules.

- **confidence** The confidence of a rule is defined $conf(X \text{ implies } Y) = supp(X \cup Y)/supp(X)$ . Be careful when reading the expression: here $supp(X \cup Y)$ means "support for occurrences of transactions where X and Y both appear", not "support for occurrences of transactions where either X or Y appears". Confidence ranges from 0 to 1. Confidence is an estimate of $Pr(Y \mid X)$, the probability of observing Y given X. The support $supp(X)$ of an itemset X is defined as the proportion of transactions in the data set which contain the itemset.

- **lift** The lift of a rule is defined as *lift(X implies Y) = supp(X ∪ Y)/((supp(Y) x supp(X))* or the ratio of the observed support to that expected if X and Y were independent. Lift can also be defined as *lift(X implies Y) =conf(X implies Y)/supp(Y)*. Lift measures how far from independence are X and Y. It ranges within 0 to positive infinity. Values close to 1 imply that X and Y are independent and the rule is not interesting.
- **conviction** conviction is sensitive to rule direction i.e. *conv(X implies Y)* is not same as *conv(Y implies X)*. Conviction is somewhat inspired in the logical definition of implication and attempts to measure the degree of implication of a rule. Conviction is defined as *conv(X implies Y) =(1 - supp(Y))/(1 - conf(X implies Y))*
- **gain** When this option is selected, the gain is calculated using the *gain theta* parameter.
- **laplace** When this option is selected, the Laplace is calculated using the *laplace k* parameter.
- **ps** When this option is selected, the ps criteria is used for rule selection.

**min confidence *(real)*** This parameter specifies the minimum confidence of the rules.

**min criterion value *(real)*** This parameter specifies the minimum value of the rules for the selected criterion.

**gain theta *(real)*** This parameter specifies the parameter *Theta* which is used in the Gain calculation.

**laplace k *(real)*** This parameter specifies the parameter *k* which is used in the Laplace function calculation.

## Tutorial Processes

### Introduction to the Create Association Rules operator

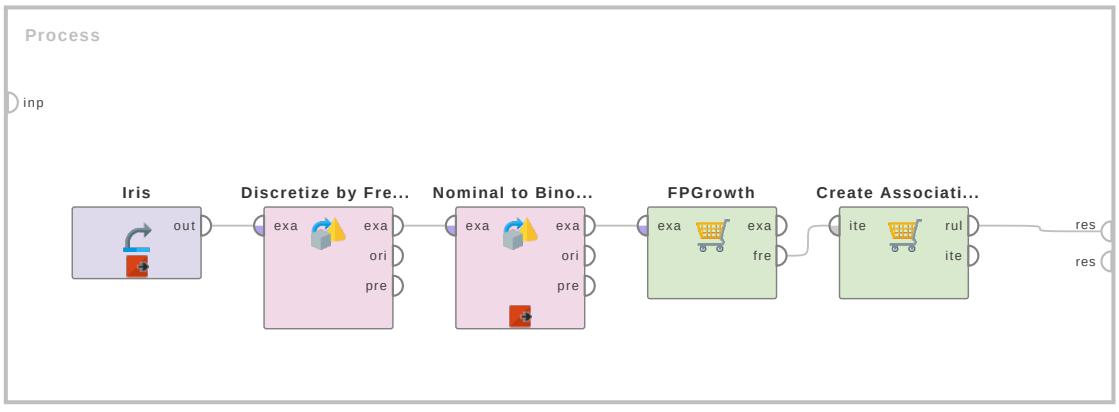



Figure 4.71: Tutorial process 'Introduction to the Create Association Rules operator'.

The 'Iris' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the ExampleSet. As you can see, the ExampleSet has real attributes. Thus the FP-Growth operator cannot be applied on it directly because the FP-Growth operator requires all attributes to be binominal. We have to do some preprocessing to mold the ExampleSet into

desired form. The Discretize by Frequency operator is applied to change the real attributes to nominal attributes. Then the Nominal to Binominal operator is applied to change these nominal attributes to binominal attributes. Finally, the FP-Growth operator is applied to generate frequent itemsets. The frequent itemsets generated from the FP-Growth operator are provided to the Create Association Rules operator. The resultant association rules can be viewed in the Results Workspace. Run this process with different values for different parameters to get a better understanding of this operator.

# FP-Growth


This Operator efficiently calculates all frequently-occurring itemsets in an ExampleSet, using the FP-tree data structure.

## Description

When online shopping, you will sometimes get a suggestion of the following form: "Customers who bought item X also bought item Y." This suggestion is an example of an association rule. To derive it, you first have to know which items on the market most frequently co-occur in customers' shopping baskets, and here the FP-Growth algorithm has a role to play.

The FP-Growth algorithm is an efficient algorithm for calculating frequently co-occurring items in a transaction database. To understand how it works, let's start with some terminology, using a customer transaction as an example:

- *item* - any object that is sold on the market
- *basket* - a container for one or more items selected by the customer
- *itemset* - any subset of items that are sold together, in the same shopping basket
- *transaction* - the complete set of items in an individual shopping basket, at the moment of purchase
- *transaction database* - the complete set of shopping baskets / transactions recorded by the merchant

Here, the words "basket" and "transaction" are used interchangably, because we identify the customer's shopping basket with the items that were purchased. To make these definitions concrete, consider the following transaction database:

- transaction1 = (product1, product2, product7)
- transaction2 = (product2, product5, product7)
- transaction3 = (product6, product7, product8, product9)
- transaction4 = (product1, product3, product4, product6, product7)

Nine distinct items are for sale, and there are four baskets / transactions, with a varying number of items. The item appearing most frequently, product7, appears four times in the transactions database. Each of the following itemsets occurs twice: (product1, product7), (product2, product7), (product6, product7).

An FP-tree data structure can be efficiently created, compressing the data so much that, in many cases, even large databases will fit into main memory. In the example above, the FP-tree would have product7, the most frequently occurring product, next to the root, with branches from product7 to product1, product2, and product6. If we insist that a product must appear more than once in the transaction database, then the remaining products are excluded from the FP-tree. The transaction database might have started out as a 4 x 9 (transactions x products) data table, with many zero entries, but now it is reduced to a minimalistic tree that captures only the relevant frequency data.

Even with an efficient tree structure, the number of itemsets considered by the algorithm can grow very large. With the help of the parameter *max number of itemsets*, you can if necessary reduce runtime and memory.

Remember that online shopping is merely an example; the FP-Growth algorithm can be applied to any problem that can be formulated in terms of items, itemsets, and baskets / transactions. The typical setting for the algorithm is a large transaction database (many baskets), with only a small number of items in each basket – small compared to the set of all items.

- *support* = (Number of times an item or itemset appears in the database) / (Number of baskets in the database)

In general, the concept of "minimum support" creates a cutoff, defining what is meant by frequent or not-so-frequent occurrences of an itemset. If an item or an itemset appears in only a few baskets, it is excluded, via the parameters *min support* or *min frequency*. The exclusion of infrequently-occurring items and itemsets helps to compress the data and improves the statistical significance of the results. On the other hand, if the value for *min support* or *min frequency* is set too high, the algorithm may find zero itemsets. Hence, this Operator provides two major modes, via the checkbox *find min number of itemsets*:

1. if unchecked, with a fixed minimum support value, and
2. if checked, with a dynamic minimum support value, to ensure that the result includes a minimum number of itemsets.

FP-Growth supports several different formats for the input data. Please note the following requirements:

- in the ExampleSet, one transaction = one row. For a discussion of the columns, see below.
- all the item values must be nominal
- a transaction ID is optional and, if present, it should have the "id" role, so that it is not identified as an item.

For the columns, the three available input formats are illustrated in the second tutorial, together with necessary pre-processing. Here's the summary:

- *item list in a column*: All the items belonging to a transaction appear in a single column, separated by *item separators*, in a CSV-like format. As with CSV files, the items can be quoted, and escape characters are available. You can *trim item names*.
- *items in separate columns*: All the items belonging to a transaction appear in separate columns. For each transaction, the first item name appears in the first column, the second item name in the second column, etc. The number of columns corresponds to the basket with the maximum number of items. Missing values indicate no item. You can *trim item names*.
- *items in dummy coded columns*: Every item in the set of all items has its own column, and the item name is the column name. For each transaction, the binominal values (true/false) indicate whether the item can be found in the basket. If your data is binominal but does not identify the values as true/false, you may have to set the *positive value* parameter.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. As discussed in detail in the description, this Operator supports several different formats for the input data.

## Output Ports

**example set *(exa)*** The ExampleSet that was given as input is passed through without changes.

**frequent sets *(fre)*** The frequently-occurring itemsets are delivered through this port. Operators such as Create Association Rules can use these frequently-occuring itemsets to generate association rules.

## Parameters

**input format** See the second tutorial for examples. As discussed in detail in the description, this Operator supports several different formats for the input data.

- **item list in a column** All the items belonging to a transaction appear in a single column, separated by *item separators*, in a CSV-like format.
- **items in separate columns** All the items belonging to a transaction appear in separate columns, with the first item name appearing in the first column, the second item name in the second column, etc.
- **items in dummy coded columns** Every item in the set of all items has its own column, and the item name is the column name. For each transaction, the binominal values (true/false) indicate whether the item can be found in the basket.

**item separators** This parameter defines the item separator. It can also be provided as a regular expression.

**use quotes** Check this parameter to define a *quotes character*. As in CSV files, if *item separators* are likely to appear in the item name, quotes can be used to prevent confusion. For example if (,) is the item separator and (") is the *quotes character*, then the row (a,b,c,d) will be interpreted as 4 items. On the other hand, ("a,b,c,d") will be interpreted as a single item, with value a,b,c,d.

**quotes character** This parameter defines the *quotes character* and is only available if *use quotes* is checked.

**escape character** This parameter defines the *escape character*, used to escape the *quotes character* or the item separator. For example, if (") is the *quotes character* and ('\') is the *escape character*, then ("yes") is interpreted as (yes) and (\"yes\") is interpreted as ("yes"). If ('|') is the item separator and ('\') is the *escape character*, then a row (a\|b|c) is interpreted as two items, (a|b) and (c).

**trim item names** If this parameter is checked, whitespace at the beginning and the end of item names is deleted.

**positive value** In the case of *items in dummy coded columns*, with binominal Attributes, this parameter determines which value should be treated as positive, and hence which items belong to a transaction. If this parameter is left blank, the positive value is inferred from the ExampleSet.

**min requirement** This parameter makes available two different methods for defining a cutoff, eliminating infrequently-occurring itemsets.

- **support** The minimum support value (ratio of occurrences to ExampleSet size)
- **frequency** The minimum frequency (number of occurrences)

**min support** Minimum support = (number of occurrences of an itemset) / (size of the ExampleSet)

Decrease this value to increase the number of itemsets in the result.

**min frequency** Minimum frequency = number of occurrences of an itemset

Decrease this value to increase the number of itemsets in the result.

**min items per itemset** The lower bound for the size of an itemset.

**max items per itemset** The upper bound for the size of an itemset (0: no upper bound).

**max number of itemsets** The upper bound for the number of itemsets (0: no upper bound). If you run out of memory, either decrease this value or increase the value for *min support* or *min frequency*.

**find min number of itemsets** If this parameter is checked, the results will contain at least a *minimum number of itemsets*, those with highest support. The minimum support value is automatically decreased until the minimum number of itemsets is found.

**min number of itemsets** This parameter is only available when *find min number of itemsets* is checked. This parameter specifies the minimum number of itemsets that should be included in the results.

**max number of retries** This parameter is only available when *find min number of itemsets* is checked. When automatically decreasing the value for minimum support / minimum frequency, this parameter determines how many times the Operator may decrease the value before giving up. Increase this number to get more results.

**requirement decrease factor** This parameter is only available when *find min number of itemsets* is checked. When automatically decreasing the value for minimum support / minimum frequency, this multiplicative factor determines the new cutoff value. A lower value results in fewer steps to find the desired number of itemsets.

**must contain list** This parameter specifies items that must be included in the frequently-occurring itemsets, if any, via a list of exact item names.

**must contain regexp** This parameter specifies items that must be included in the frequently-occurring itemsets, if any, via a regular expression.

## Tutorial Processes

### Introduction to the FP-Growth Operator

The process shows a market basket analysis. A data set containing transactions is loaded using the Retrieve Operator. A breakpoint is inserted here so that you can view the ExampleSet. We have to do some preprocessing using the Aggregate Operator to mold the ExampleSet into an acceptable input format. A breakpoint is inserted before the FP-Growth Operator so that you can view the input data. The FP-Growth Operator is applied to generate frequent itemsets. Finally, the Create Association Rules Operator is used to create rules from the frequent item sets. The frequent itemsets and the association rules can be viewed in the Results View. Run this process with different values of the parameters to get a better understanding of this Operator.

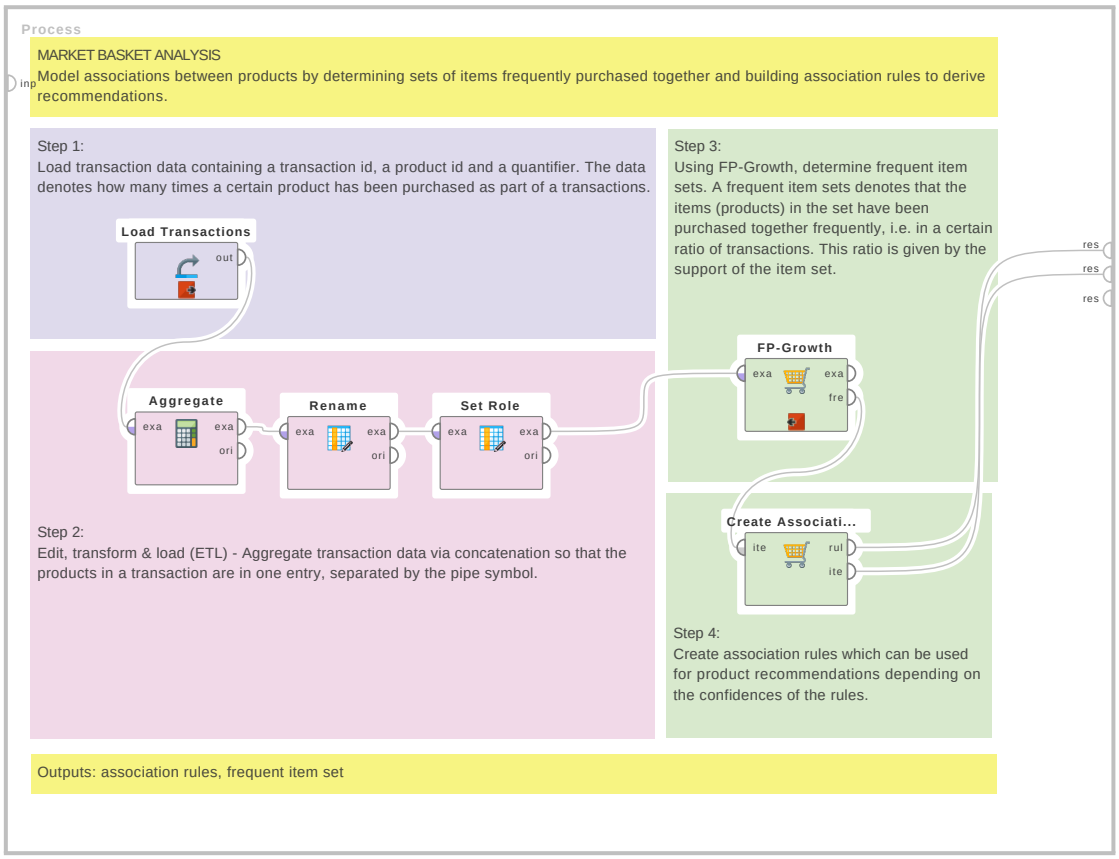



Figure 4.72: Tutorial process ‘Introduction to the FP-Growth Operator’.

### The input formats of the FP-Growth Operator

Data is loaded and transformed to three different input formats. A breakpoint is inserted before the FP-Growth Operators so that you can see the input data in each of these formats. The FP-Growth Operator is used and the resulting itemsets can be viewed in the Results View. The results are all the same because the input data is the same, despite the difference in formats.

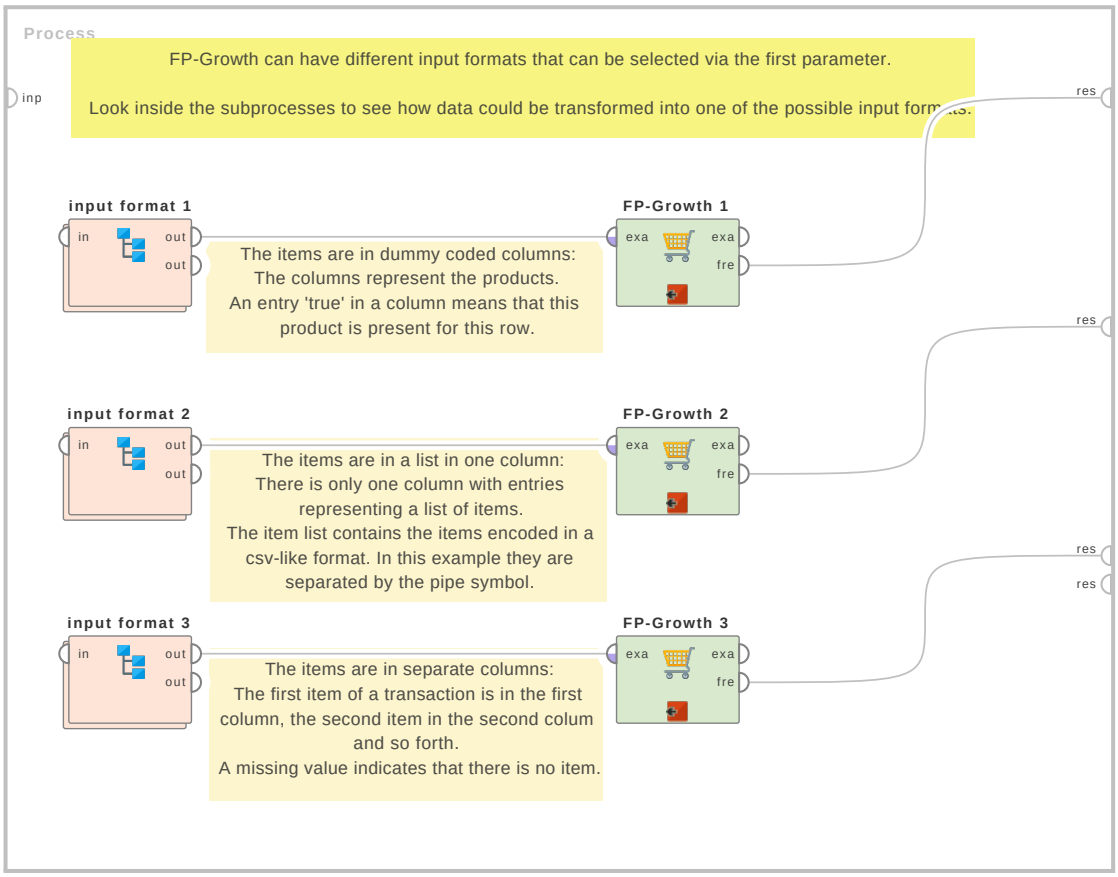



Figure 4.73: Tutorial process ‘The input formats of the FP-Growth Operator’.

# Generalized Sequential Patterns


This operator searches sequential patterns in a set of transactions using the GSP (Generalized Sequential Pattern) algorithm. GSP is a popular algorithm used for sequence mining.

## Description

This operator searches sequential patterns in a set of transactions. The ExampleSet must contain one attribute for the time and one attribute for the customer. Moreover, each transaction must be encoded as a single example. The time and customer attributes are specified through the *time attribute* and *customer id* parameters respectively. This pair of attributes is used for generating one sequence per customer containing every transaction ordered by the time of each transaction. The algorithm then searches sequential patterns in the form of: If a customer bought item ‘a’ and item ‘c’ in one transaction, he bought item ‘b’ in the next. This pattern is repre-

sented in this form: <a, c> then <b>. The minimal support describes how many customer must support such a pattern for regarding it as frequent. Infrequent patterns will be dropped. A customer supports such a pattern, if there are some parts of his sequence that includes that pattern. The above pattern would be supported by a customer, for example, with transactions: <s, g> then <a, s, c> then <b> then <f, h>. The minimum support criteria is specified through the *min support* parameter.

The *min gap*, *max gap* and *window size* parameters determine how transactions are handled. For example, if the above customer forgot to buy item 'c', and had to return 5 minutes later to buy it, then his transactions would look like: <s, g> then <a, s> then <c> then <b> then <f, h>. This would not support the pattern <a, c> then <b>. To avoid this problem, the window size determines, how long a subsequent transaction is treated as the same transaction. If the window size is larger than 5 minutes then <c> would be treated as being part of the second transaction and hence this customer would support the above pattern. The *max gap* parameter causes a customers sequence not to support a pattern, if the transactions containing this pattern are too widely separated in time. The *min gap* parameter does the same if they are too near.

This technique overcomes some crucial drawbacks of existing mining methods, for example:

- absence of time constraints: This drawback is overcome by the *min gap* and *max gap* parameters.
- rigid definition of a transaction: This drawback is overcome by the sliding time window.

Please note that all attributes (except customer and time attributes) of the given ExampleSet should be binominal, i.e. nominal attributes with only two possible values. If your ExampleSet does not satisfy this condition, you may use appropriate preprocessing operators to transform it into the required form. The discretization operators can be used for changing the value of numerical attributes to nominal attributes. Then the Nominal to Binominal operator can be used for transforming nominal attributes into binominal attributes.

Please note that the sequential patterns are mined for the positive entries in your ExampleSet, i.e. for those nominal values which are defined as positive in your ExampleSet. If your data does not specify the positive entries correctly, you may set them using the *positive value* parameter. This only works if all your attributes contain this value.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. Please make sure that all attributes (except customer and time attributes) of the ExampleSet are binominal.

## Output Ports

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**patterns *(pat)*** The GSP algorithm is applied on the given ExampleSet and the resultant set of sequential patterns is delivered through this port.

## Parameters

**customer id *(string)*** This parameter specifies the name of the attribute that will be used for identifying the customers.

**time attribute *(string)*** This parameter specifies the name of the numerical attribute that specifies the time of a transaction.

**min support *(real)*** Prune patterns that are supported by less than *min support* percentage of the customers.

**window size *(real)*** The time window within successive transactions will be additional handled as a single transaction.

**max gap *(real)*** The *max gap* parameter causes a customers sequence not to support a pattern, if the transactions containing this pattern are too widely separated in time.

**min gap *(real)*** The *min gap* parameter causes a customers sequence not to support a pattern, if the transactions containing this pattern are too near in time.

**positive value *(string)*** This parameter determines which value of the binominal attributes should be treated as positive. The attributes with this value in an example are considered to be part of that transaction.

## Tutorial Processes

### Introduction to the GSP operator

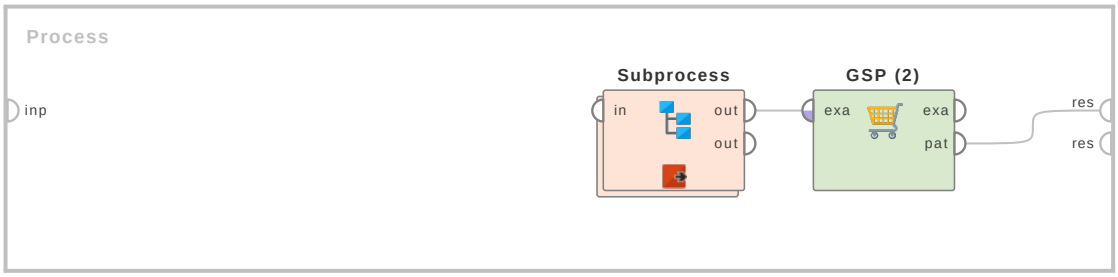



Figure 4.74: Tutorial process 'Introduction to the GSP operator'.

The ExampleSet expected by the GSP operator should meet the following criteria: It should have an attribute that can be used for identifying the customers.It should have a numerical attribute that represents the time of the transaction.All other attributes are used for representing items of transactions. These attributes should be binominal.

This Example Process starts with the Subprocess operator. A sequence of operators is applied in the subprocess to generate an ExampleSet that satisfies all the above mentioned conditions. A breakpoint is inserted after the Subprocess operator so that you can have a look at the ExampleSet. The Customer attribute represents the customers, this ExampleSet has five. The Time attribute represents the time of transaction. For simplicity the Time Attribute consists of 20 days. In real scenarios unix time should be used. There are 20 binominal attributes in the ExampleSet that represent items that the customer may buy in a transaction. In this ExampleSet, value 'true' for an item in an example means that this item was bought in this transaction (represented by the current example). The GSP operator is applied on this ExampleSet. The customer id and time attribute parameters are set to 'Customer' and 'Time' respectively. The positive value parameter is set to 'true'. The min support parameter is set 0.9. The resultant set of sequential patterns can be seen in the Results Workspace.

# Unify Item Sets


Compares sets of frequent item sets and removes common not unique sets.

## Description

This operator compares a number of FrequentItemSet sets and removes every not unique FrequentItemSet.

## Input Ports

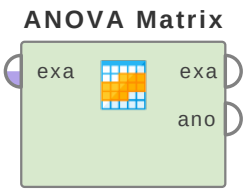
**frequent item sets *(fre)*** Expects at least two FrequentItemSets.

## Output Ports

**frequent item sets *(fre)*** Same number as input ports. Contains only itemsets that are unique across all input ports.

# 4.4 Correlations

## ANOVA Matrix




This operator performs an ANOVA significance test for all numerical attributes based on the groups defined by all the nominal attributes. ANOVA is a general technique that can be used to test the hypothesis that the means among two or more groups are equal, under the assumption that the sampled populations are normally distributed.

### Description

The ANalysis Of VAriance (ANOVA) is a statistical model in which the observed variance in a particular variable is partitioned into components attributable to different sources of variation. In its simplest form, ANOVA provides a statistical test of whether or not the means of several groups are all equal, and therefore generalizes a t-test to more than two groups. Doing multiple two-sample t-tests would result in an increased chance of committing a type I error. For this reason, ANOVA is useful in comparing two, three, or more means. 'False positive' or Type I error is defined as the probability that a decision to reject the null hypothesis will be made when it is in fact true and should not have been rejected. In the typical application of ANOVA, the null hypothesis is that all groups are simply random samples of the same population. This implies that all treatments have the same effect (perhaps none). Rejecting the null hypothesis implies that different treatments result in altered effects.

### Differentiation

- **Grouped ANOVA** The Grouped ANOVA operator performs ANOVA significance test for the user-specified anova attribute (numerical) based on the groups defined by user-specified attribute (nominal). See page 618 for details.

### Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. The ExampleSet should have both nominal and numerical attributes because this operator performs an ANOVA significance test for all numerical attributes based on the groups defined by all the nominal attributes.

### Output Ports

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**anova *(ano)*** The ANOVA significance test for all numerical attributes is performed based on the groups defined by all the nominal attributes. The resultant ANOVA matrix is returned from this port.

## Parameters

**significance level *(real)*** This parameter specifies the significance level for the ANOVA calculation.

**only distinct *(boolean)*** This parameter indicates if only rows with distinct values of the aggregation attribute should be used for the calculation of the aggregation function.

## Related Documents

- **Grouped ANOVA** (page 618)

## Tutorial Processes

### ANOVA matrix of the Golf data set

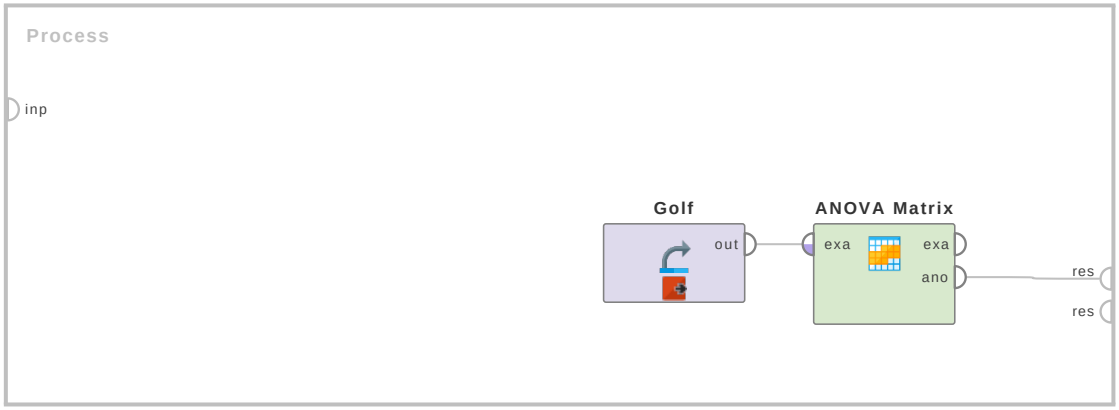
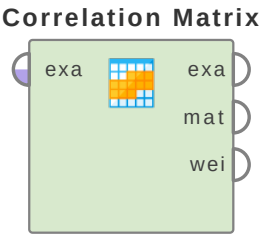



Figure 4.75: Tutorial process 'ANOVA matrix of the Golf data set'.

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the ExampleSet. You can see that the ExampleSet has both nominal and numerical attributes. The ANOVA Matrix operator is applied on this ExampleSet. This operator performs an ANOVA significance test for all numerical attributes based on the groups defined by all the nominal attributes. The resultant ANOVA matrix can be viewed in the Results Workspace.

# Correlation Matrix




This Operator determines correlation between all Attributes and it can produce a weights vector based on these correlations. Correlation is a statistical technique that can show whether and how strongly pairs of Attributes are related.

## Description

A correlation is a number between -1 and +1 that measures the degree of association between two Attributes (call them X and Y). A positive value for the correlation implies a positive association. In this case large values of X tend to be associated with large values of Y and small values of X tend to be associated with small values of Y. A negative value for the correlation implies a negative or inverse association. In this case large values of X tend to be associated with small values of Y and vice versa.

Suppose we have two Attributes X and Y, with means X' and Y' respectively and standard deviations S(X) and S(Y) respectively. The correlation is computed as summation from 1 to n of the product *(X(i)-X').(Y(i)-Y')* and then dividing this summation by the product *(n-1).S(X).S(Y)* where *n* is total number of Examples and *i* is the increment variable of summation. There can be other formulas and definitions but let us stick to this one for simplicity.

As discussed earlier a positive value for the correlation implies a positive association. Suppose that an X value was above average, and that the associated Y value was also above average. Then the product *(X(i)-X').(Y(i)-Y')* would be the product of two positive numbers which would be positive. If the X value and the Y value were both below average, then the product above would be of two negative numbers, which would also be positive. Therefore, a positive correlation is evidence of a general tendency that large values of X are associated with large values of Y and small values of X are associated with small values of Y.

As discussed earlier a negative value for the correlation implies a negative or inverse association. Suppose that an X value was above average, and that the associated Y value was instead below average. Then the product *(X(i)-X').(Y(i)-Y')* would be the product of a positive and a negative number which would make the product negative. If the X value was below average and the Y value was above average, then the product above would also be negative. Therefore, a negative correlation is evidence of a general tendency that large values of X are associated with small values of Y and small values of X are associated with large values of Y.

This Operator can be used for creating a correlation matrix that shows correlations of all the Attributes of the input ExampleSet. The Attribute weights vector; based on the correlations can also be returned by this Operator. Using this weights vector, highly correlated Attributes can be removed from the ExampleSet with the help of the Select by Weights Operator. Highly correlated Attributes can be more easily removed by simply using the Remove Correlated Attributes Operator. Correlated Attributes are usually removed because they are similar in behavior and only have little influence when calculating predictions. They may also hamper run time and memory usage.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet on which the correlation matrix will be calculated.

## Output Ports

**example set *(exa)*** The ExampleSet, that was given as input is passed through without changes.

**matrix *(mat)*** The correlations of all Attributes of the input ExampleSet are calculated and the resultant correlation matrix is returned from this port. The correlation for nominal Attributes is not well defined and results in a missing value. When Attributes contain missing values, only pairwise complete tuples are used for calculating the correlation.

**weights *(wei)*** The Attribute weights vector based on the correlations of the Attributes is delivered through this output port.

## Parameters

**attribute filter type** This parameter allows you to select the Attribute selection filter; the method you want to use for selecting Attributes. It has the following options:

- **all** This option selects all the Attributes of the ExampleSet, no Attributes are removed. This is the default option.
- **single** This option allows the selection of a single Attribute. The required Attribute is selected by the *attribute* parameter.
- **subset** This option allows the selection of multiple Attributes through a list (see parameter *attributes*). If the meta data of the ExampleSet is known all Attributes are present in the list and the required ones can easily be selected.
- **regular_expression** This option allows you to specify a regular expression for the Attribute selection. The regular expression filter is configured by the parameters *regular expression, use except expression and except expression*.
- **value_type** This option allows selection of all the Attributes of a particular type. It should be noted that types are hierarchical. For example real and integer types both belong to the numeric type. The value type filter is configured by the parameters *value type, use value type exception, except value type*.
- **block_type** This option allows the selection of all the Attributes of a particular block type. It should be noted that block types may be hierarchical. For example value_series_start and value_series_end block types both belong to the value_series block type. The block type filter is configured by the parameters *block type, use block type exception, except block type*.
- **no_missing_values** This option selects all Attributes of the ExampleSet which do not contain a missing value in any Example. Attributes that have even a single missing value are removed.
- **numeric_value_filter** All numeric Attributes whose Examples all match a given numeric condition are selected. The condition is specified by the *numeric condition* parameter. Please note that all nominal Attributes are also selected irrespective of the given numerical condition.

**attribute** The required Attribute can be selected from this option. The Attribute name can be selected from the drop down box of the parameter if the meta data is known.

**attributes** The required Attributes can be selected from this option. This opens a new window with two lists. All Attributes are present in the left list. They can be shifted to the right list, which is the list of selected Attributes that will make it to the output port.

**regular expression** Attributes whose names match this expression will be selected. The expression can be specified through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression** If enabled, an exception to the first regular expression can be specified. This exception is specified by the *except regular expression* parameter.

**except regular expression** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in *regular expression* parameter).

**value type** This option allows to select a type of Attribute. One of the following types can be chosen: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date, time.

**use value type exception** If enabled, an exception to the selected type can be specified. This exception is specified by the *except value type* parameter.

**except value type** The Attributes matching this type will be removed from the final output even if they matched the before selected type, specified by the *value type* parameter. One of the following types can be selected here: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date, time.

**block type** This option allows to select a block type of Attribute. One of the following types can be chosen: single_value, value_series, value_series_start, value_series_end, value_matrix, value_matrix_start, value_matrix_end, value_matrix_row_start.

**use block type exception** If enabled, an exception to the selected block type can be specified. This exception is specified by the *except block type* parameter.

**except block type** The Attributes matching this block type will be removed from the final output even if they matched the before selected type by the *block type* parameter. One of the following block types can be selected here: single_value, value_series, value_series_start, value_series_end, value_matrix, value_matrix_start, value_matrix_end, value_matrix_row_start.

**numeric condition** The numeric condition used by the numeric condition filter type. A numeric Attribute is kept if all Examples match the specified condition for this Attribute. For example the numeric condition '$> 6$' will keep all numeric Attributes having a value of greater than 6 in every Example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '$(> 0$ && $< 2)$ || $(>10$ && $< 12)$' are not allowed because they use both && and ||. Nominal Attributes are always kept, regardless of the specified numeric condition.

**include special attributes** Special Attributes are Attributes with special roles. These are: id, label, prediction, cluster, weight and batch. Also custom roles can be assigned to Attributes. By default all special Attributes are delivered to the output port irrespective of the conditions in the Select Attribute Operator. If this parameter is set to true, special Attributes are also tested against conditions specified in the Select Attribute Operator and only those Attributes are selected that match the conditions.

**invert selection** If this parameter is set to true the selection is reversed. In that case all Attributes matching the specified condition are removed and the other Attributes remain in the output ExampleSet. Special Attributes are kept independent of the *invert selection*

parameter as along as the *include special attributes* parameter is not set to true. If so the condition is also applied to the special Attributes and the selection is reversed if this parameter is checked.

**normalize weights *(boolean)*** This parameter indicates if the weights of the resultant Attribute weights vector should be normalized. If set to true, all weights are normalized such that the minimum weight is 0 and the maximum weight is 1.

**squared correlation *(boolean)*** This parameter indicates if the squared correlation should be calculated. If set to true, the correlation matrix shows squares of correlations instead of simple correlations.

## Tutorial Processes

### Correlation matrix of the Golf data set

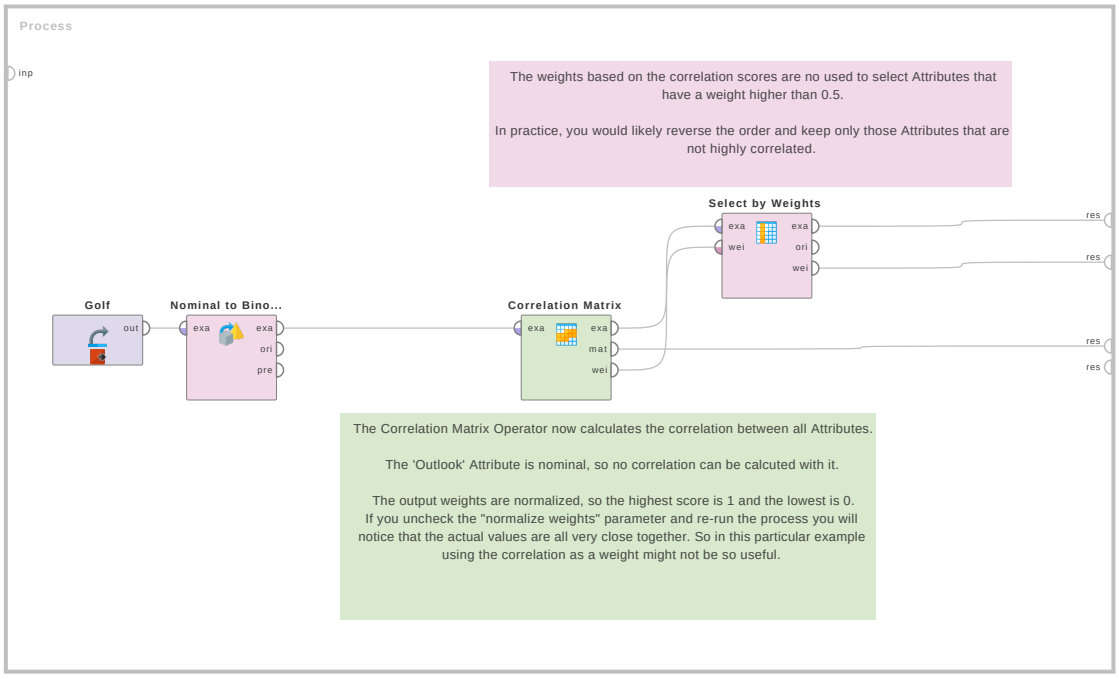


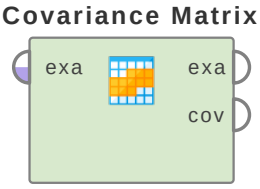

Figure 4.76: Tutorial process ‘Correlation matrix of the Golf data set’.

The ‘Golf’ data set is loaded using the Retrieve Operator. A breakpoint is inserted here so that you can view the ExampleSet. As you can see, the ExampleSet has 4 regular Attributes i.e. ‘Outlook’, ‘Temperature’, ‘Humidity’ and ‘Wind’ and the label Attribute ‘Play’.

All Attributes with only two nominal values are converted to binominal Attributes using Nominal to Binominal. Then the Correlation Matrix Operator is applied on the result. The weights vector generated by this Operator is provided to the Select by Weights Operator along with the data set. The parameters of the Select by Weights Operator are adjusted such that the Attributes with weights greater than 0.5 are selected and all other Attributes are removed. This is why the resultant ExampleSet only has the ‘Play’ and the ‘Temperature’ Attribute.

The correlation matrix, weights vector and the resultant ExampleSet can be viewed in the Results Workspace. For the correlation matrix you can see that Outlook is a nominal Attribute, so no correlation can be calculated with it. The correlation of an Attribute to its self is always one, so the diagonal entries are all 1.

# Covariance Matrix




This operator calculates the covariance between all attributes of the input ExampleSet and returns a covariance matrix giving a measure of how much two attributes change together.

## Description

Covariance is a measure of how much two attributes change together. If the greater values of one attribute mainly correspond with the greater values of the other attribute, and the same holds for the smaller values, i.e. the attributes tend to show similar behavior, the covariance is a positive number. In the opposite case, when the greater values of one attribute mainly correspond to the smaller values of the other, i.e. the attributes tend to show opposite behavior, the covariance is negative. The sign of the covariance therefore shows the tendency in the linear relationship between the variables. For two attributes *x* and *y* having means *E{x}* and *E{y}*, the covariance is defined as:

*Cov(x,y) = E{[ x - E(x) ][ y - E(y) ]}*

The covariance calculation begins with pairs of *x* and *y*, takes their differences from their mean values and multiplies these differences together. For instance, if for *x1* and *y1* this product is positive, for that pair of data points the values of *x* and *y* have varied together in the same direction from their means. If the product is negative, they have varied in opposite directions. The larger the magnitude of the product, the stronger the strength of the relationship. The covariance is defined as the mean value of this product, calculated using each pair of data points *x(i)* and *y(i)*. If the covariance is zero, then the cases in which the product was positive were offset by those in which it was negative, and there is no linear relationship between the two attributes.

The value of the covariance is interpreted as follows:

- Positive covariance: indicates that *higher* than average values of one attribute tend to be paired with higher than average values of the other attribute.
- Negative covariance: indicates that *higher* than average values of one attribute tend to be paired with lower than average values of the other attribute.
- Zero covariance: if the two attributes are independent, the covariance will be zero. However, a covariance of zero does not necessarily mean that the variables are independent. A nonlinear relationship can exist that still would result in a covariance value of zero.

Because the number representing covariance depends on the units of the data, it is difficult to compare covariances among data sets having different scales. A value that might represent a strong linear relationship for one data set might represent a very weak one in another.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**covariance *(cov)*** The covariances of all attributes of the input ExampleSet are calculated and the resultant covariance matrix is returned from this port.

## Tutorial Processes

### Covariance matrix of the Polynomial data set

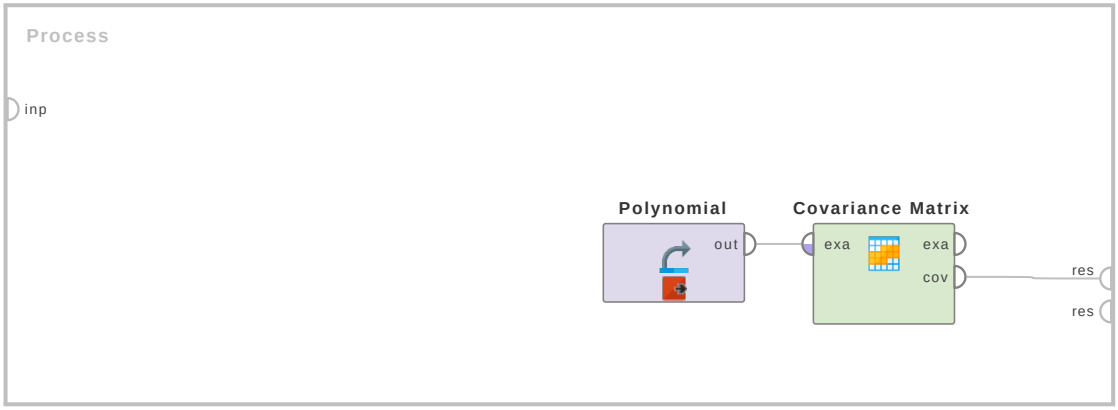
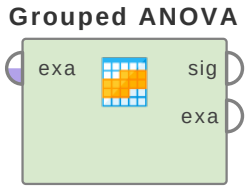



Figure 4.77: Tutorial process 'Covariance matrix of the Polynomial data set'.

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the ExampleSet. As you can see that the ExampleSet has 5 real attributes. The Covariance Matrix operator is applied on this ExampleSet. The resultant covariance matrix can be viewed in the Results Workspace.

# Grouped ANOVA




This operator performs an ANOVA significance test for the user-specified attribute (numerical) based on the groups defined by the user-specified attribute (nominal). ANOVA is a general technique that can be used to test the hypothesis that the means among two or more groups are equal, under the assumption that the sampled populations are normally distributed.

## Description

The Grouped ANOVA operator creates groups of the input ExampleSet based on the grouping attribute which is specified by the *group by attribute* parameter. For each of the groups the mean and variance of the anova attribute is calculated and an ANalysis Of VAriance (ANOVA) is performed. The anova attribute is specified by the *anova attribute* parameter. It is important to note that the grouping attribute should be nominal and the anova attribute should be numerical. The result of this operator is a significance test result for the specified significance level (specified by the *significance level* parameter) indicating if the values for the attribute are significantly different between the groups defined by the grouping attribute.

ANalysis Of VAriance (ANOVA) is a statistical model in which the observed variance in a particular variable is partitioned into components attributable to different sources of variation. In its simplest form, ANOVA provides a statistical test of whether or not the means of several groups are all equal, and therefore generalizes a t-test to more than two groups. Doing multiple two-sample t-tests would result in an increased chance of committing a Type I error. For this reason, ANOVA is useful in comparing two, three, or more means. 'False positive' or a Type I error is defined as the probability that a decision to reject the null hypothesis will be made when it is in fact true and should not have been rejected. In the typical application of ANOVA, the null hypothesis is that all groups are simply random samples of the same population. This implies that all treatments have the same effect (perhaps none). Rejecting the null hypothesis implies that different treatments result in altered effects.

## Differentiation

- **ANOVA Matrix** The ANOVA Matrix operator performs ANOVA significance test for all numerical attributes based on the groups defined by all the nominal attributes. See page 609 for details.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. The ExampleSet should have both nominal and numerical attributes because this operator performs an ANOVA significance test for a specified numerical attribute based on the groups defined by a specified nominal attribute.

## Output Ports

**significance *(sig)*** The ANOVA test is performed and the ANOVA significance test result is returned from this port.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**anova attribute *(string)*** The ANOVA is calculated for the attribute specified by this parameter based on the groups defined by the *group by attribute* parameter. It is compulsory that this attribute should be numerical.

**group by attribute *(string)*** Grouping is performed by the values of the attribute specified by this parameter. It is compulsory that this attribute should be nominal.

**significance level *(real)*** This parameter specifies the significance level for the ANOVA calculation.

**only distinct *(boolean)*** This parameter indicates if only rows with distinct values of the aggregation attribute should be used for the calculation of the aggregation function.

## Related Documents

- **ANOVA Matrix** (page 609)

## Tutorial Processes

### Grouped ANOVA of the Golf data set

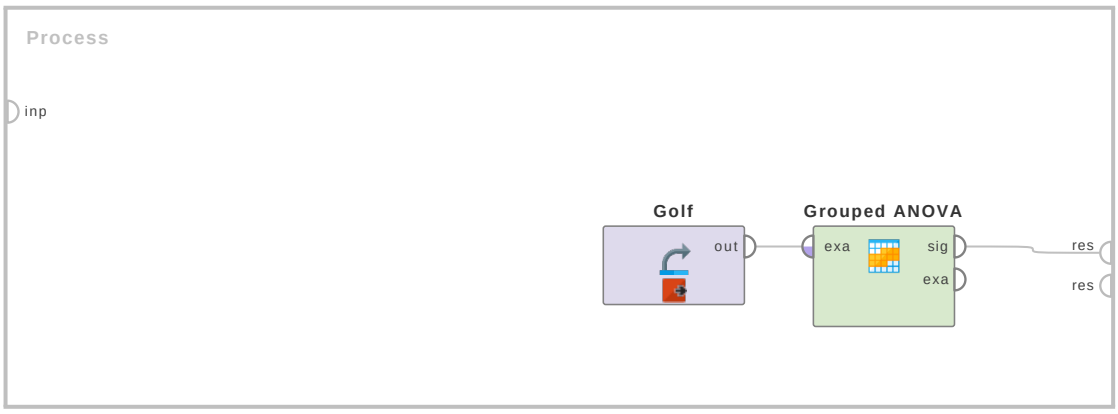
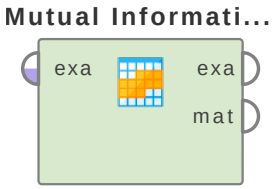



Figure 4.78: Tutorial process 'Grouped ANOVA of the Golf data set'.

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the ExampleSet. You can see that the ExampleSet has both nominal and numerical attributes. The Grouped ANOVA operator is applied on this ExampleSet. The anova attribute and group by attribute parameter are set to 'Humidity' and 'Play' respectively. This operator performs an ANOVA significance test for the 'Humidity' attribute based on the groups defined by the 'Play' attribute. The result of the ANOVA significance test can be viewed in the Results Workspace.

# Mutual Information Matrix




This operator calculates the mutual information between all attributes of the input ExampleSet and returns a mutual information matrix. Mutual information of two attributes is a quantity that measures the mutual dependence of the two attributes.

## Description

Mutual information is one of many quantities that measures how much one attribute tells us about another. It is a dimensionless quantity, and can be thought of as the reduction in uncertainty about one attribute given the knowledge of another. High mutual information indicates a large reduction in uncertainty; low mutual information indicates a small reduction; and zero mutual information between two attribute means the variables are independent.

This operator calculates the mutual information matrix between all attributes of the input ExampleSet. Please note that this simple implementation performs a data scan for each attribute combination and might therefore take some time for non-memory tables.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set *(exa)*** The ExampleSet that was given as input is passed after some modifications to the output through this port. Please note that this ExampleSet is not exactly the same as the input ExampleSet.

**matrix *(mat)*** The mutual information of all attributes of the input ExampleSet are calculated and the resultant matrix is returned from this port.

## Parameters

**number of bins *(integer)*** This parameter specifies the number of bins to be used for numerical attributes.

## Tutorial Processes

### Mutual information matrix of the Polynomial data set

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the ExampleSet. You can see that the ExampleSet has 5 real attributes. The Mutual Information Matrix operator is applied on this ExampleSet. The resultant matrix can be viewed in the Results Workspace.

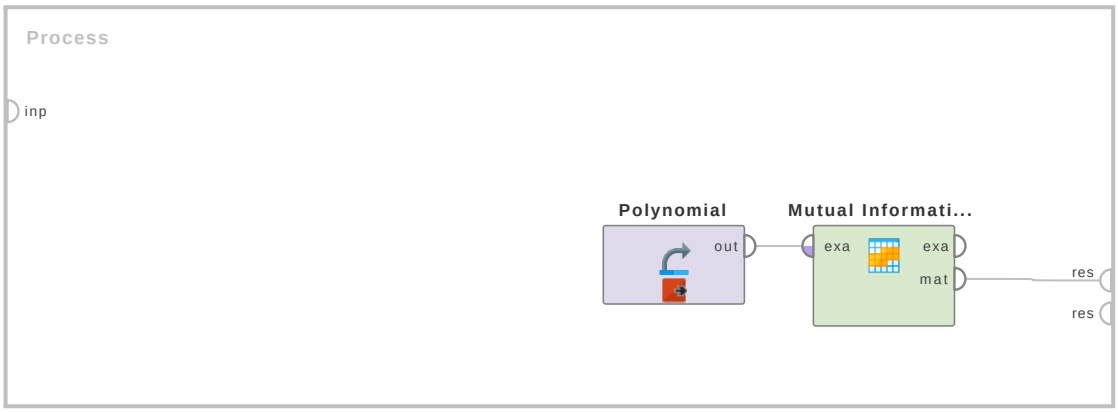
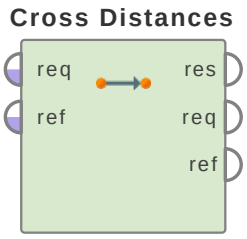



Figure 4.79: Tutorial process 'Mutual information matrix of the Polynomial data set'.

# 4.5 Similarities

## Cross Distances




This operator calculates the distance between each example of a 'request set' ExampleSet to each example of a 'reference set' ExampleSet. This operator is also capable of calculating similarity instead of distance.

### Description

The Cross Distances operator takes two ExampleSets as input i.e. the 'reference set' and 'request set' ExampleSets. It creates an ExampleSet that contains the distance between each example of the 'request set' ExampleSet to each example of the 'reference set' ExampleSet. Please note that both input ExampleSets should have the same attributes and in the same order. This operator will not work properly if the order of the attributes is different. This operator is also capable of calculating similarity instead of distance. If the *compute similarities* parameter is set to true, similarities are calculated instead of distances. Please note that both input ExampleSets should have *id* attributes. If *id* attributes are not present, this operator automatically creates *id* attributes for such ExampleSets. The measure to use for calculating the distances can be specified through the parameters. Four type of measures are provided: *mixed measures*, *nominal measures*, *numerical measures* and *Bregman divergences*.

If data is imported from two different sources that are supposed to represent the same data but which have columns in different orders, the Cross Distances operator will not behave as expected. It is possible to work round this by using the Generate Attributes operator to recreate attributes in both ExampleSets in the same order.

## Input Ports

**request set *(req)*** This input port expects an ExampleSet. This ExampleSet will be used as the 'request set'. Please note that both input ExampleSets ( 'request set' and 'reference set') should have the same attributes and in the same order. This operator will not work properly if the order of the attributes is different. Also note that both input ExampleSets should have *id* attributes. If *id* attributes are not present, this operator automatically creates *id* attributes for such ExampleSets.

**reference set *(ref)*** This input port expects an ExampleSet. This ExampleSet will be used as the 'reference set'. Please note that both input ExampleSets ( 'request set' and 'reference set') should have same attributes and in the same order. This operator will not work properly if the order of the attributes is different. Also note that both input ExampleSets should have *id* attributes. If *id* attributes are not present, this operator automatically creates *id* attributes for such ExampleSets.

## Output Ports

**result set *(res)*** An ExampleSet that contains the distance (or similarity, if the *compute similarities* parameter is set to true) between each example of the 'request set' ExampleSet to each example of the 'reference set' ExampleSet is delivered through this port.

**request set *(req)*** The 'request set' ExampleSet that was provided at the *request set* input port is delivered through this port. If the input ExampleSet had an *id* attribute then the ExampleSet is delivered without any modification. Otherwise an *id* attribute is automatically added to the input ExampleSet.

**reference set *(ref)*** The 'reference set' ExampleSet that was provided at the *reference set* input port is delivered through this port. If the input ExampleSet had an *id* attribute then the ExampleSet is delivered without any modification. Otherwise an *id* attribute is automatically added to the input ExampleSet.

## Parameters

**measure types *(selection)*** This parameter is used for selecting the type of measure to be used for calculating distances (or similarity).The following options are available: *mixed measures*, *nominal measures*, *numerical measures* and *Bregman divergences*.

**mixed measure *(selection)*** This parameter is available when the *measure type* parameter is set to 'mixed measures'. The only available option is the 'Mixed Euclidean Distance'

**nominal measure *(selection)*** This parameter is available when the *measure type* parameter is set to 'nominal measures'. This option cannot be applied if the input ExampleSet has numerical attributes. If the input ExampleSet has numerical attributes the 'numerical measure' option should be selected.

**numerical measure *(selection)*** This parameter is available when the *measure type* parameter is set to 'numerical measures'. This option cannot be applied if the input ExampleSet has nominal attributes. If the input ExampleSet has nominal attributes the 'nominal measure' option should be selected.

**divergence *(selection)*** This parameter is available when the *measure type* parameter is set to 'bregman divergences'.

**kernel type *(selection)*** This parameter is available only when the *numerical measure* parameter is set to 'Kernel Euclidean Distance'. The type of the kernel function is selected through this parameter. Following kernel types are supported:

- **dot** The dot kernel is defined by*k(x,y)=x*y* i.e.it is inner product of*x* and *y*.
- **radial** The radial kernel is defined by *exp(-g ||x-y||^2)* where *g* is the *gamma* that is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by *k(x,y)=(x*y+1)^d* where *d* is the degree of the polynomial and it is specified by the *kernel degree* parameter. The Polynomial kernels are well suited for problems where all the training data is normalized.
- **neural** The neural kernel is defined by a two layered neural net *tanh(a x*y+b)* where *a* is *alpha* and *b* is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is 1/N, where N is the data dimension. Note that not all choices of *a* and *b* lead to a valid kernel function.
- **sigmoid** This is the sigmoid kernel. Please note that the *sigmoid* kernel is not valid under some parameters.
- **anova** This is the anova kernel. It has adjustable parameters *gamma* and *degree*.
- **epachnenikov** The Epanechnikov kernel is this function *(3/4)(1-u2)* for *u* between -1 and 1 and zero for *u* outside that range. It has two adjustable parameters *kernel sigma1* and *kernel degree*.
- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters *kernel sigma1, kernel sigma2* and *kernel sigma3*.
- **multiquadric** The multiquadric kernel is defined by the square root of *||x-y||^2 + c^2*. It has adjustable parameters *kernel sigma1* and *kernel sigma shift*.

**kernel gamma *(real)*** This is the SVM kernel parameter gamma. This parameter is available when only the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1 *(real)*** This is the SVM kernel parameter sigma1. This parameter is available only when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2 *(real)*** This is the SVM kernel parameter sigma2. This parameter is available only when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3 *(real)*** This is the SVM kernel parameter sigma3. This parameter is available only when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel shift *(real)*** This is the SVM kernel parameter shift. This parameter is available only when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *multiquadric*.

**kernel degree *(real)*** This is the SVM kernel parameter degree. This parameter is available only when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a *(real)*** This is the SVM kernel parameter a. This parameter is available only when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

**kernel b *(real)*** This is the SVM kernel parameter b. This parameter is available only when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

**only top k *(boolean)*** This parameter indicates if only the *k* nearest to each request example should be calculated.

**k *(integer)*** This parameter is only available when the *only top k* parameter is set to true. It determines how many of the nearest examples should be shown in the result.

**search for *(selection)*** This parameter is only available when the *only top k* parameter is set to true. It determines if the nearest or the farthest distances should be selected.

**compute similarities *(boolean)*** If this parameter is set true, similarities are computed instead of distances. All measures will still be usable, but measures that are not originally distance or respective similarity measure are transformed to match optimization direction.

## Tutorial Processes

### Introduction to the Cross Distances operator

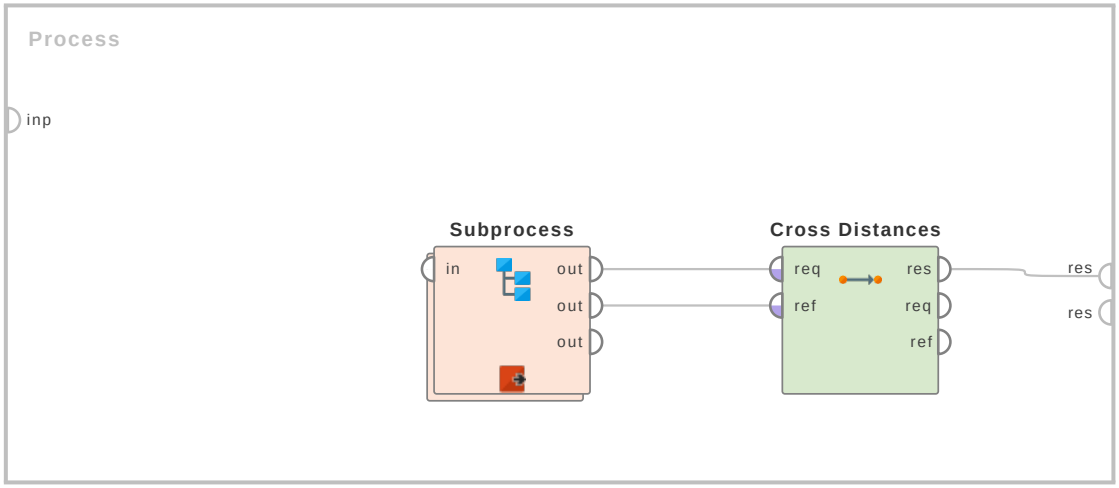
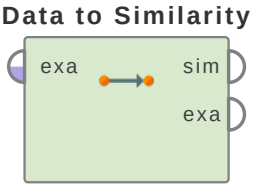



Figure 4.80: Tutorial process 'Introduction to the Cross Distances operator'.

This Example Process starts with a Subprocess operator. This subprocess generates the 'request set' ExampleSet and the 'reference set' ExampleSet. A breakpoint is inserted here so that you can have a look at the ExampleSets before application of the Cross Distances operator. You can see that the 'request set' has only 1 example with id 'id_1'. The 'reference set' has just two examples with ids 'id_1' and 'id_2'. Both ExampleSets have three attributes in the same order. It is very important that both ExampleSets should have the same attributes and in the same order otherwise the Cross Distances operator will not behave as expected. The Cross Distances operator is applied on these ExampleSets. The resultant ExampleSet that contains the distance

between each example of the ‘request set’ ExampleSet to each example of the ‘reference set’ ExampleSet is calculated by the Cross Distance operator. The resultant ExampleSet can be viewed in the Results Workspace.

# Data to Similarity




This operator measures the similarity of each example of the given ExampleSet with every other example of the same ExampleSet.

## Description

The Data to Similarity operator calculates the similarity among examples of an ExampleSet. Same comparisons are not repeated again e.g. if example *x* is compared with example *y* to compute similarity then example *y* will not be compared again with example *x* to compute similarity because the result will be the same. Thus if there are *n* examples in the ExampleSet, this operator does not return *n^2* similarity comparisons. Instead it returns *(n)(n-1)/2* similarity comparisons. This operator provides many different measures for similarity computation. The measure to use for calculating the similarity can be specified through the parameters. Four types of measures are provided: *mixed measures*, *nominal measures*, *numerical measures* and *Bregman divergences*.

The behavior of this operator can be considered close to a certain scenario of the Cross Distances operator, if the same ExampleSet is provided at both inputs of the Cross Distances operator and the *compute similarities* parameter is also set to true. In this case the Cross Distances operator behaves similar to the Data to Similarity operator. There are a few differences though e.g. in this scenario examples are also compared with themselves and secondly the signs (i.e.+ive or -ive) of the results are also different.

## Differentiation

- **Data to Similarity Data** The Data to Similarity Data operator calculates the similarity among all examples of an ExampleSet. Even examples are compared to themselves. Thus if there are *n* examples in the ExampleSet, this operator returns *n^2* similarity comparisons. The Data to Similarity Data operator returns an ExampleSet which is merely a view, so there should be no memory problems. See page 629 for details.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**similarity *(sim)*** A similarity measure object that contains the calculated similarity between each example of the given ExampleSet with every other example of the same ExampleSet is delivered through this port.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**measure types *(selection)*** This parameter is used for selecting the type of measure to be used for calculating similarity. following options are available: *mixed measures*, *nominal measures*, *numerical measures* and *Bregman divergences*.

**mixed measure *(selection)*** This parameter is available if the *measure type* parameter is set to 'mixed measures'. The only available option is the 'Mixed Euclidean Distance'

**nominal measure *(selection)*** This parameter is available if the *measure type* parameter is set to 'nominal measures'. This option cannot be applied if the input ExampleSet has numerical attributes. In this case the 'numerical measure' option should be selected.

**numerical measure *(selection)*** This parameter is available if the *measure type* parameter is set to 'numerical measures'. This option cannot be applied if the input ExampleSet has nominal attributes. In this case the 'nominal measure' option should be selected.

**divergence *(selection)*** This parameter is available if the *measure type* parameter is set to 'bregman divergences'.

**kernel type *(selection)*** This parameter is only available if the *numerical measure* parameter is set to 'Kernel Euclidean Distance'. The type of the kernel function is selected through this parameter. Following kernel types are supported:

- **dot** The dot kernel is defined by*k(x,y)=x*y* i.e.it is the inner product of*x* and *y*.
- **radial** The radial kernel is defined by *exp(-g ||x-y||^2)* where *g* is the *gamma* that is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by *k(x,y)=(x*y+1)^d* where *d* is the degree of the polynomial and it is specified by the *kernel degree* parameter. The Polynomial kernels are well suited for problems where all the training data is normalized.
- **neural** The neural kernel is defined by a two layered neural net *tanh(a x*y+b)* where *a* is *alpha* and *b* is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is 1/N, where N is the data dimension. Note that not all choices of *a* and *b* lead to a valid kernel function.
- **sigmoid** This is the sigmoid kernel. Please note that the *sigmoid* kernel is not valid under some parameters.
- **anova** This is the anova kernel. It has the adjustable parameters *gamma* and *degree*.
- **epachnenikov** The Epanechnikov kernel is this function *(3/4)(1-u2)* for *u* between -1 and 1 and zero for *u* outside that range. It has the two adjustable parameters *kernel sigma1* and *kernel degree*.
- **gaussian_combination** This is the gaussian combination kernel. It has the adjustable parameters *kernel sigma1, kernel sigma2* and *kernel sigma3*.
- **multiquadric** The multiquadric kernel is defined by the square root of *||x-y||^2 + c^2*. It has the adjustable parameters *kernel sigma1* and *kernel sigma shift*.

**kernel gamma *(real)*** This is the SVM kernel parameter gamma. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1** ***(real)*** This is the SVM kernel parameter sigma1. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2** ***(real)*** This is the SVM kernel parameter sigma2. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3** ***(real)*** This is the SVM kernel parameter sigma3. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel shift** ***(real)*** This is the SVM kernel parameter shift. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *multiquadric*.

**kernel degree** ***(real)*** This is the SVM kernel parameter degree. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a** ***(real)*** This is the SVM kernel parameter a. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

**kernel b** ***(real)*** This is the SVM kernel parameter b. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

## Related Documents

- **Data to Similarity Data** (page 629)

## Tutorial Processes

### Introduction to the Data to Similarity operator

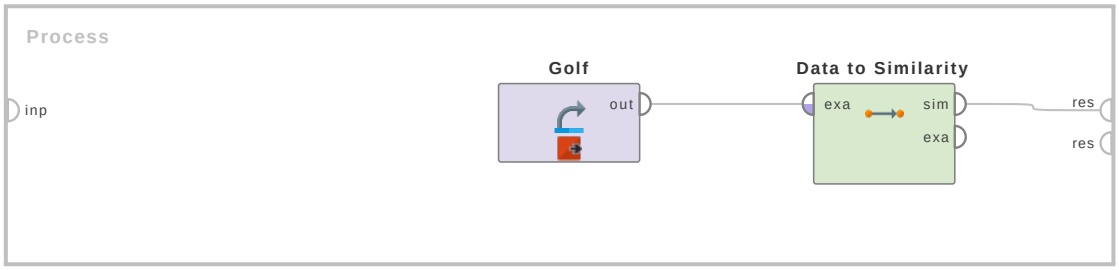
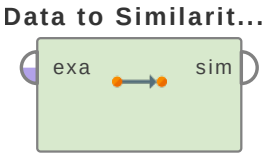



Figure 4.81: Tutorial process 'Introduction to the Data to Similarity operator'.

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look the ExampleSet. You can see that the ExampleSet has 14 examples. The Data to Similarity operator is applied on it to compute the similarity of examples. As there are 14 examples in the given ExampleSet, there will be 91 (i.e. (14)(14-1)/2) similarity comparisons in the resultant similarity measure object which can be seen in the Results Workspace.

# Data to Similarity Data




This operator measures the similarity of each example of the given ExampleSet with every other example of the same ExampleSet and returns a similarity ExampleSet.

## Description

The Data to Similarity Data operator calculates the similarity among all examples of an ExampleSet. Examples are even compared to themselves. Thus if there are *n* examples in the ExampleSet, this operator returns *n^2* similarity comparisons. This operator provides many different measures for similarity computation. The measure to use for calculating the similarity can be specified through the parameters. Four types of measures are provided: *mixed measures*, *nominal measures*, *numerical measures* and *Bregman divergences*. Please note that the data set created by this operator is merely a view, so there should be no memory problems.

The behavior of this operator can be considered close to a certain scenario of the Cross Distances operator, if the same ExampleSet is provided at both inputs of the Cross Distances operator and the *compute similarities* parameter is also set to false. In this case the Cross Distances operator behaves similar to the Data to Similarity Data operator. Besides sorting order, there is no major difference between these two scenarios.

## Differentiation

- **Data to Similarity** The Data to Similarity operator calculates the similarity among examples of an ExampleSet. Same comparisons are not repeated again e.g. if example *x* is compared with example *y* to compute similarity then example *y* will not be compared again with example *x* to compute similarity because the result will be the same. Thus if there are *n* examples in the ExampleSet, this operator does not return *n^2* similarity comparisons. Instead it returns *(n)(n-1)/2* similarity comparisons. Moreover, this operator returns a similarity measure object instead of an ExampleSet. See page 626 for details.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**similarity example set *(sim)*** A similarity ExampleSet that contains the calculated similarities is delivered through this port.

## Parameters

**measure types *(selection)*** This parameter is used for selecting the type of measure to be used for calculating similarity. following options are available: *mixed measures*, *nominal measures*, *numerical measures* and *Bregman divergences*.

**mixed measure** ***(selection)*** This parameter is available if the *measure type* parameter is set to 'mixed measures'. The only available option is the 'Mixed Euclidean Distance'

**nominal measure** ***(selection)*** This parameter is available if the *measure type* parameter is set to 'nominal measures'. This option cannot be applied if the input ExampleSet has numerical attributes. In this case the 'numerical measure' option should be selected.

**numerical measure** ***(selection)*** This parameter is available if the *measure type* parameter is set to 'numerical measures'. This option cannot be applied if the input ExampleSet has nominal attributes. In this case the 'nominal measure' option should be selected.

**divergence** ***(selection)*** This parameter is available if the *measure type* parameter is set to 'bregman divergences'.

**kernel type** ***(selection)*** This parameter is only available if the *numerical measure* parameter is set to 'Kernel Euclidean Distance'. The type of the kernel function is selected through this parameter. Following kernel types are supported:

- **dot** The dot kernel is defined by*k(x,y)=x*y* i.e.it is the inner product of*x* and *y*.
- **radial** The radial kernel is defined by *exp(-g ||x-y||^2)* where *g* is the *gamma* that is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- **polynomial** The polynomial kernel is defined by *k(x,y)=(x*y+1)^d* where *d* is the degree of the polynomial and it is specified by the *kernel degree* parameter. The Polynomial kernels are well suited for problems where all the training data is normalized.
- **neural** The neural kernel is defined by a two layered neural net *tanh(a x*y+b)* where *a* is *alpha* and *b* is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is 1/N, where N is the data dimension. Note that not all choices of *a* and *b* lead to a valid kernel function.
- **sigmoid** This is the sigmoid kernel. Please note that the *sigmoid* kernel is not valid under some parameters.
- **anova** This is the anova kernel. It has the adjustable parameters *gamma* and *degree*.
- **epachnenikov** The Epanechnikov kernel is this function *(3/4)(1-u2)* for *u* between -1 and 1 and zero for *u* outside that range. It has the two adjustable parameters *kernel sigma1* and *kernel degree*.
- **gaussian_combination** This is the gaussian combination kernel. It has the adjustable parameters *kernel sigma1, kernel sigma2* and *kernel sigma3*.
- **multiquadric** The multiquadric kernel is defined by the square root of *||x-y||^2 + c^2*. It has the adjustable parameters *kernel sigma1* and *kernel sigma shift*.

**kernel gamma** ***(real)*** This is the SVM kernel parameter gamma. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1** ***(real)*** This is the SVM kernel parameter sigma1. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2** ***(real)*** This is the SVM kernel parameter sigma2. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3** ***(real)*** This is the SVM kernel parameter sigma3. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel shift** ***(real)*** This is the SVM kernel parameter shift. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *multiquadric*.

**kernel degree** ***(real)*** This is the SVM kernel parameter degree. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a** ***(real)*** This is the SVM kernel parameter a. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

**kernel b** ***(real)*** This is the SVM kernel parameter b. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

## Related Documents

- **Data to Similarity** (page 626)

## Tutorial Processes

### Introduction to the Data to Similarity Data operator

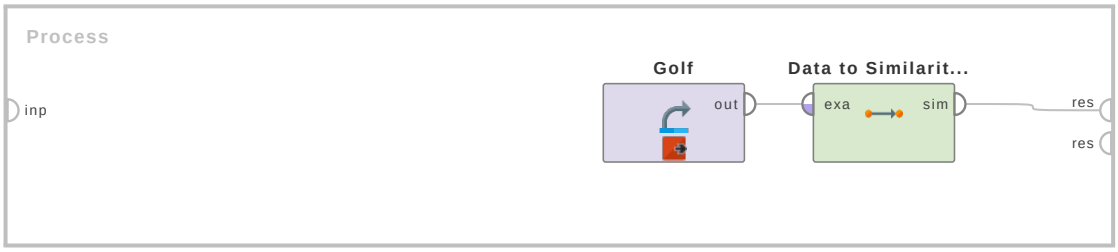
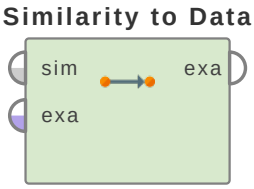



Figure 4.82: Tutorial process 'Introduction to the Data to Similarity Data operator'.

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet which has 14 examples. The Data to Similarity Data operator is applied on it to compute the similarity of examples. As there are 14 examples in the given ExampleSet, there will be 196 (i.e. 14 x 14) similarity comparisons in the resultant similarity ExampleSet which can be seen in the Results Workspace.

# Similarity to Data




This operator calculates an ExampleSet from the given similarity measure.

## Description

The Similarity to Data operator calculates an ExampleSet from the given SimilarityMeasure Object. The ExampleSet can be in form of a long table or a matrix. This behavior can be controlled by the *table type* parameter. A similarity measure object contains the calculated similarity between each example of an ExampleSet with every other example of the same ExampleSet. Operators like the Data to Similarity operator can generate a similarity measure object.

## Input Ports

**similarity *(sim)*** This input port expects a similarity measure object. A similarity measure object contains the calculated similarity between each example of an ExampleSet with every other example of the same ExampleSet. The Data to Similarity operator can generate a similarity measure object.

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Data to Similarity operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set *(exa)*** An ExampleSet is calculated from the given similarity measure and it is returned from this port.

## Parameters

**table type *(selection)*** This parameter indicates if the resulting table should have a matrix format or a long table format.

## Tutorial Processes

### Introduction to the Similarity to Data operator

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look the ExampleSet. You can see that the ExampleSet has 14 examples. The Data to Similarity operator is applied on it to compute the similarity of examples. As there are 14 examples in the given ExampleSet, there will be 91 (i.e. (14)(14-1)/2) similarity comparisons in the resultant similarity measure object. A breakpoint is inserted here so that you can have a look at this SimilarityMeasure Object. The Similarity to Data operator is applied on this SimilarityMeasure Object to calculate an ExampleSet. The table type parameter is set to 'matrix', therefore the resultant ExampleSet is in the form of a matrix. It can be seen in the Results Workspace.

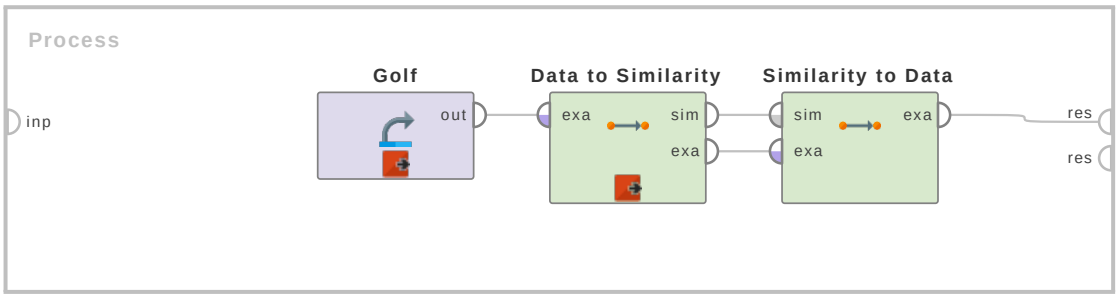
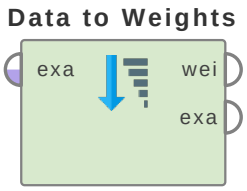



Figure 4.83: Tutorial process 'Introduction to the Similarity to Data operator'.

# 4.6 Feature Weights

## Data to Weights




This operator simply generates an attribute weights vector with weight 1.0 for each input attribute.

### Description

The Data to Weights operator creates a new *attribute weights IOObject* from the given ExampleSet. The result is a vector of attribute weights containing the weight 1.0 for each attribute of the input ExampleSet.

### Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

### Output Ports

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

### Parameters

**normalize weights *(boolean)*** This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.

**sort weights *(boolean)*** This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the *sort direction* parameter.

**sort direction *(selection)*** This parameter is only available when the *sort weights* parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

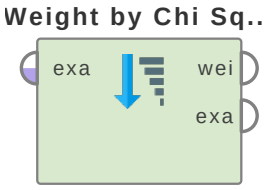
## Tutorial Processes

### Generating a weight vector with weight 1.0 for all the attributes


Figure 4.84: Tutorial process 'Generating a weight vector with weight 1.0 for all the attributes'.

The 'Golf' data set is loaded using the Retrieve operator. The Data to Weights operator is applied on it to generate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, the sort weights parameter is set to true and the sort direction parameter is set to 'ascending'. Run the process and see the results of this process in the Results Workspace. You can see that all attributes have been assigned to weight 1.0.

# Weight by Chi Squared Statistic




This operator calculates the relevance of the attributes by computing for each attribute of the input ExampleSet the value of the chi-squared statistic with respect to the class attribute.

## Description

The Weight by Chi Squared Statistic operator calculates the weight of attributes with respect to the class attribute by using the chi-squared statistic. The higher the weight of an attribute, the more relevant it is considered. Please note that the chi-squared statistic can only be calculated for nominal labels. Thus this operator can be applied only on ExampleSets with nominal label.

The chi-square statistic is a nonparametric statistical technique used to determine if a distribution of observed frequencies differs from the theoretical expected frequencies. Chi-square statistics use nominal data, thus instead of using means and variances, this test uses frequencies. The value of the chi-square statistic is given by

*X2 = Sigma [ (O-E)2/ E ]*

where *X2* is the chi-square statistic, *O* is the observed frequency and *E* is the expected frequency. Generally the chi-squared statistic summarizes the discrepancies between the expected number of times each outcome occurs (assuming that the model is true) and the observed number of times each outcome occurs, by summing the squares of the discrepancies, normalized by the expected numbers, over all the categories.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.

**example set *(exa)*** ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**normalize weights *(boolean)*** This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.

**sort weights *(boolean)*** This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the *sort direction* parameter.

**sort direction *(selection)*** This parameter is available only when the *sort weights* parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

**number of bins *(integer)*** This parameter specifies the number of bins used for discretization of numerical attributes before the chi-squared test can be performed.

## Tutorial Processes

### Calculating the weights of the attributes of the Golf data set

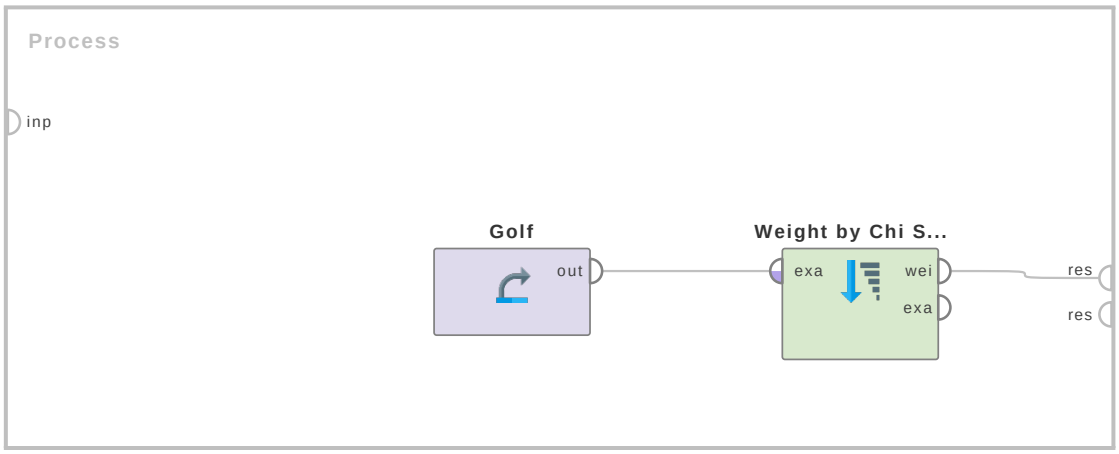
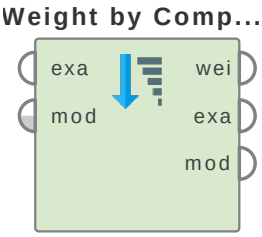



Figure 4.85: Tutorial process 'Calculating the weights of the attributes of the Golf data set'.

The 'Golf' data set is loaded using the Retrieve operator. The Weight by Chi Squared Statistic operator is applied on it to calculate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, thus all the weights will be normalized in range 0 to 1. The sort weights parameter is set to true and the sort direction parameter is set to 'ascending', thus the results will be in ascending order of the weights. You can verify this by viewing the results of this process in the Results Workspace.

# Weight by Component Model




This operator creates attribute weights of the ExampleSet by using a component created by operators like the PCA, GHA or ICA. If the model given to this operator is PCA then this operator behaves exactly as the Weight by PCA operator.

## Description

The Weight by Component Model operator always comes after operators like the PCA, GHA or ICA. The ExampleSet and Preprocessing model generated by these operators is connected to the *ExampleSet* and *Model* ports of the Weight by Component Model operator. The Weight by Component Model operator then generates attribute weights of the original ExampleSet using a component created by the previous operator (i.e. PCA, GHA, ICA etc). The component is specified by the *component number* parameter. If the *normalize weights* parameter is not set to true exact values of the selected component are used as attribute weights. The *normalize weights* parameter is usually set to true to spread the weights between 0 and 1.

The attribute weights reflect the relevance of the attributes with respect to the class attribute. The higher the weight of an attribute, the more relevant it is considered.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the PCA operator in the attached Example Process.

**model *(mod)*** This input port expects a model. Usually the Preprocessing model generated by the operators like PCA, GHA or ICA is provided here.

## Output Ports

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**model *(mod)*** The model that was given as input is passed without changing to the output through this port.

## Parameters

**normalize weights *(boolean)*** This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.

**sort weights *(boolean)*** This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the *sort direction* parameter.

**sort direction *(selection)*** This parameter is only available when the *sort weights* parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

**component number *(integer)*** This parameter specifies the number of the component that should be used as attribute weights.

## Tutorial Processes

### Calculating the attribute weights of the Sonar data set by PCA

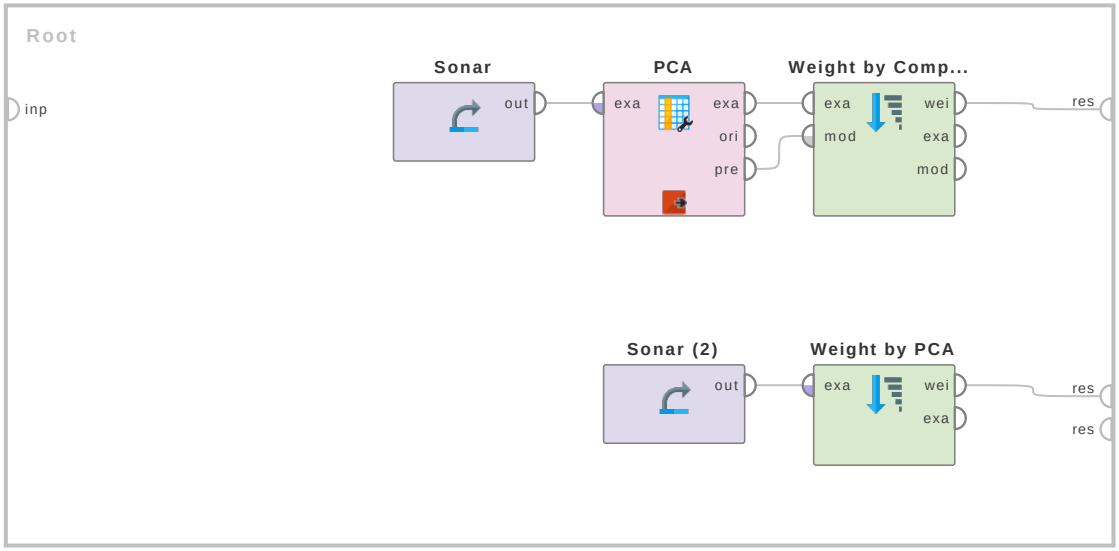
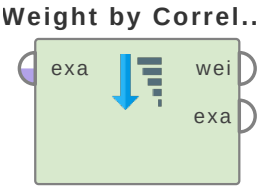



Figure 4.86: Tutorial process 'Calculating the attribute weights of the Sonar data set by PCA'.

The 'Sonar' data set is loaded using the Retrieve operator. The PCA operator is applied on it. The dimensionality reduction parameter is set to 'none'. A breakpoint is inserted here so that you can have a look at the components created by the PCA operator. Have a look at the EigenVectors generated by the PCA operator especially 'PC1' because it will be used as weights by using the Weight by Component Model operator. The Weight by Component Model operator is applied next. The ExampleSet and Model ports of the PCA operator are connected to the corresponding ports of the Weight by Component Model operator. The normalize weights and sort weights parameters are set to false, thus all the weights will be exactly the same as the selected component. The component number parameter is set to 1, thus 'PC1' will be used as attribute weights. The weights can be seen in the Results Workspace. You can see that these weights are exactly the same as the values of 'PC1'.

In the second operator chain the Weight by PCA operator is applied on the 'Sonar' data set. The parameters of the Weight by PCA operator are set exactly the same as the parameters of the Weight by Component Model operator. As it can be seen in the Results Workspace, exactly same weights are generated here.

# Weight by Correlation




This operator calculates the relevance of the attributes by computing the value of correlation for each attribute of the input ExampleSet with respect to the label attribute. This weighting scheme is based upon correlation and it returns the absolute or squared value of correlation as attribute weight.

## Description

The Weight by Correlation operator calculates the weight of attributes with respect to the label attribute by using correlation. The higher the weight of an attribute, the more relevant it is considered. Please note that the Weight by Correlation operator can be applied only on ExampleSets with numerical or binominal label. It cannot be applied on Polynominal attributes because the polynominal classes provide no information about their ordering, therefore the weights are more or less random depending on the internal numerical representation of the classes. Binominal labels work because of the representation as 0 and 1, as do numerical ones.

A correlation is a number between -1 and +1 that measures the degree of association between two attributes (call them X and Y). A positive value for the correlation implies a positive association. In this case large values of X tend to be associated with large values of Y and small values of X tend to be associated with small values of Y. A negative value for the correlation implies a negative or inverse association. In this case large values of X tend to be associated with small values of Y and vice versa.

Suppose we have two attributes X and Y, with means X' and Y' and standard deviations S(X) and S(Y) respectively. The correlation is computed as summation from 1 to n of the product *(X(i)-X').(Y(i)-Y')* and then dividing this summation by the product *(n-1).S(X).S(Y)* where *n* is the total number of examples and *i* is the increment variable of summation. There can be other formulas and definitions but let us stick to this one for simplicity.

As discussed earlier a positive value for the correlation implies a positive association. Suppose that an X value was above average, and that the associated Y value was also above average. Then the product *(X(i)-X').(Y(i)-Y')* would be the product of two positive numbers which would be positive. If the X value and the Y value were both below average, then the product above would be of two negative numbers, which would also be positive. Therefore, a positive correlation is evidence of a general tendency that large values of X are associated with large values of Y and small values of X are associated with small values of Y.

As discussed earlier a negative value for the correlation implies a negative or inverse association. Suppose that an X value was above average, and that the associated Y value was instead below average. Then the product *(X(i)-X').(Y(i)-Y')* would be the product of a positive and a negative number which would make the product negative. If the X value was below average and the Y value was above average, then the product above would also be negative. Therefore, a negative correlation is evidence of a general tendency that large values of X are associated with small values of Y and small values of X are associated with large values of Y.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**normalize weights *(boolean)*** This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.

**sort weights *(boolean)*** This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the *sort direction* parameter.

**sort direction *(selection)*** This parameter is only available when the *sort weights* parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

**squared correlation *(boolean)*** This parameter indicates if the squared correlation should be calculated instead of simple correlation. If set to true, the attribute weights are calculated as squares of correlations instead of simple correlations.

## Tutorial Processes

### Calculating the attribute weights of the Polynomial data set

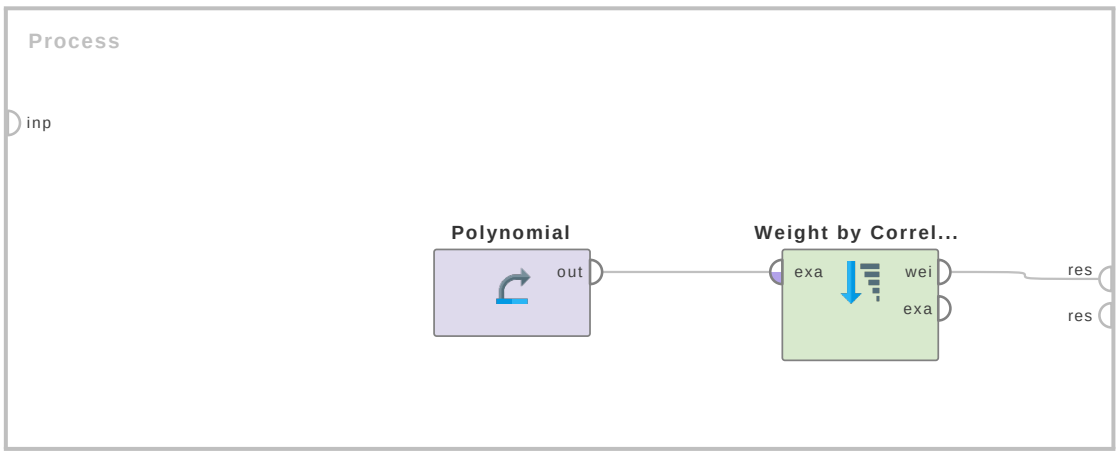
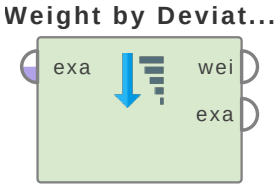



Figure 4.87: Tutorial process 'Calculating the attribute weights of the Polynomial data set'.

The 'Polynomial' data set is loaded using the Retrieve operator. The Weight by Correlation operator is applied on it to calculate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, thus all the weights will be normalized in range 0 to 1. The sort weights parameter is set to true and the sort direction parameter

is set to ‘ascending’, thus the results will be in ascending order of the weights. You can verify this by viewing the results of this process in the Results Workspace. Now set the squared correlation parameter to true and run the process again. You will see that these weights are the squares of the previous weights.

# Weight by Deviation




This operator calculates the relevance of attributes of the given ExampleSet based on the (normalized) standard deviation of the attributes.

## Description

The Weight by Deviation operator calculates the weight of attributes with respect to the label attribute based on the (normalized) standard deviation of the attributes. The higher the weight of an attribute, the more relevant it is considered. The standard deviations can be normalized by average, minimum, or maximum of the attribute. Please note that this operator can be only applied on ExampleSets with numerical label.

Standard deviation shows how much variation or dispersion exists from the average (mean, or expected value). A low standard deviation indicates that the data points tend to be very close to the mean, whereas high standard deviation indicates that the data points are spread out over a large range of values. The standard deviation is a measure of how spread out numbers are. The formula is simple: it is the square root of the Variance.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**normalize weights *(boolean)*** This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in the range from 0 to 1.

**sort weights *(boolean)*** This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the *sort direction* parameter.

**sort direction *(selection)*** This parameter is only available when the *sort weights* parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

**normalize *(selection)*** This parameter indicates if the standard deviation should be divided by the minimum, maximum, or average of the attribute.

## Tutorial Processes

### Calculating the attribute weights of the Polynomial data set

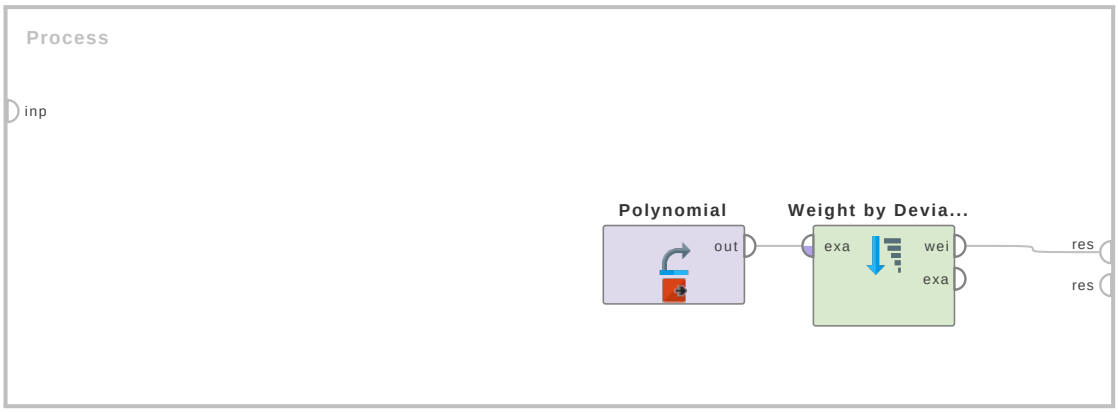
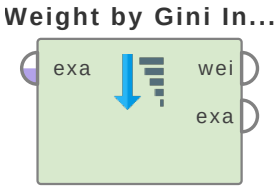



Figure 4.88: Tutorial process 'Calculating the attribute weights of the Polynomial data set'.

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can also see the standard deviation of all attributes in the 'Statistics' column in the Meta Data View. The Weight by Deviation operator is applied on this ExampleSet to calculate the weights of the attributes. The normalize weights parameter is set to false, thus the weights will not be normalized. The sort weights parameter is set to true and the sort direction parameter is set to 'ascending', thus the results will be in ascending order of the weights. You can verify this by viewing the results of this process in the Results Workspace. You can also see that these weights are exactly the same as the standard deviations of the attributes.

# Weight by Gini Index




This operator calculates the relevance of the attributes of the given ExampleSet based on the Gini impurity index.

## Description

The Weight by Gini Index operator calculates the weight of attributes with respect to the label attribute by computing the Gini index of the class distribution, if the given ExampleSet would have been split according to the attribute. Gini Index is a measure of impurity of an ExampleSet. The higher the weight of an attribute, the more relevant it is considered. Please note that this operator can be only applied on ExampleSets with nominal label.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.

**example set *(exa)*** ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**normalize weights *(boolean)*** This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.

**sort weights *(boolean)*** This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the *sort direction* parameter.

**sort direction *(selection)*** This parameter is only available when the *sort weights* parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

## Tutorial Processes

### Calculating the attribute weights of the Golf data set

The 'Golf' data set is loaded using the Retrieve operator. The Weight by Gini Index operator is applied on it to calculate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, thus all the weights will be normalized

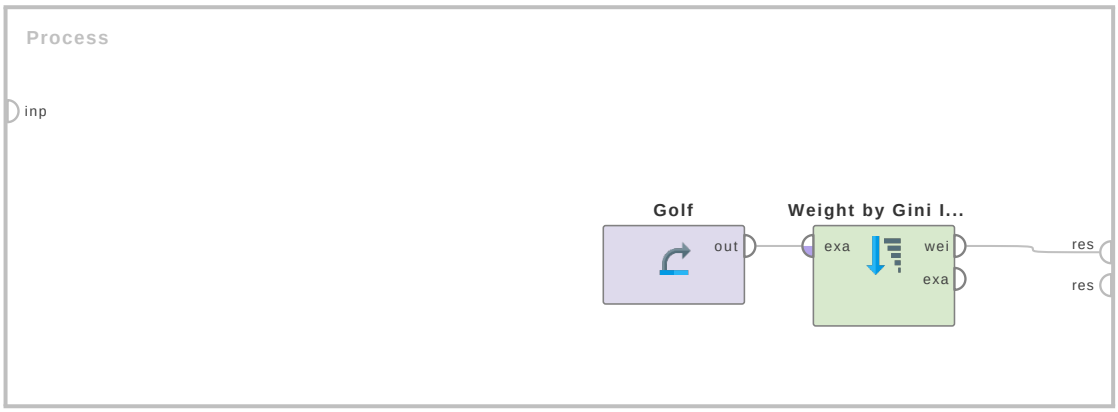
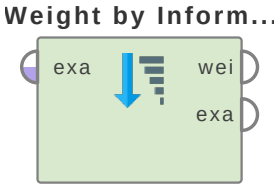



Figure 4.89: Tutorial process ‘Calculating the attribute weights of the Golf data set’.

in range 0 to 1. The sort weights parameter is set to true and the sort direction parameter is set to ‘ascending’, thus the results will be in ascending order of the weights. You can verify this by viewing the results of this process in the Results Workspace.

# Weight by Information Gain




This operator calculates the relevance of the attributes based on information gain and assigns weights to them accordingly.

## Description

The Weight by Information Gain operator calculates the weight of attributes with respect to the class attribute by using the information gain. The higher the weight of an attribute, the more relevant it is considered. Please note that this operator can be applied only on ExampleSets with nominal label.

Although information gain is usually a good measure for deciding the relevance of an attribute, it is not perfect. A notable problem occurs when information gain is applied to attributes that can take on a large number of distinct values. For example, suppose some data that describes the customers of a business. When information gain is used to decide which of the attributes are the most relevant, the customer's credit card number may have high information gain. This attribute has a high information gain, because it uniquely identifies each customer, but we may not want to assign high weights to such attributes.

Information gain ratio is sometimes used instead. This method biases against considering attributes with a large number of distinct values. However, attributes with very low information values then appear to receive an unfair advantage. The Weight by Information Gain Ratio operator uses information gain ratio for generating attribute weights.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**normalize weights *(boolean)*** This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.

**sort weights *(boolean)*** This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the *sort direction* parameter.

**sort direction *(selection)*** This parameter is only available when the *sort weights* parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

## Tutorial Processes

### Calculating the weights of the attributes of the Golf data set

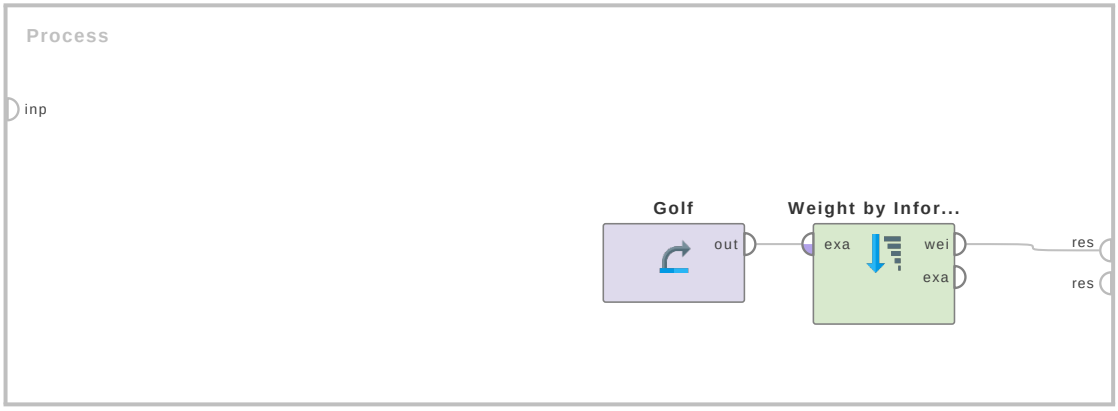
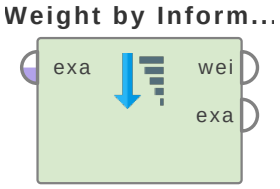



Figure 4.90: Tutorial process 'Calculating the weights of the attributes of the Golf data set'.

The 'Golf' data set is loaded using the Retrieve operator. The Weight by Information Gain operator is applied on it to calculate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, thus all the weights will be normalized in range 0 to 1. The sort weights parameter is set to true and the sort direction parameter is set to 'ascending', thus the results will be in ascending order of the weights. You can verify this by viewing the results of this process in the Results Workspace.

# Weight by Information Gain Ratio




This operator calculates the relevance of the attributes based on the information gain ratio and assigns weights to them accordingly.

## Description

The Weight by Information Gain Ratio operator calculates the weight of attributes with respect to the label attribute by using the information gain ratio. The higher the weight of an attribute, the more relevant it is considered. Please note that this operator can be only applied on ExampleSets with nominal label.

Information gain ratio is used because it solves the drawback of information gain. Although information gain is usually a good measure for deciding the relevance of an attribute, it is not perfect. A notable problem occurs when information gain is applied to attributes that can take on a large number of distinct values. For example, suppose some data that describes the customers of a business. When information gain is used to decide which of the attributes are the most relevant, the customer's credit card number may have high information gain. This attribute has a high information gain, because it uniquely identifies each customer, but we may not want to assign high weights to such attributes. The Weight by Information Gain operator uses information gain for generating attribute weights.

Information gain ratio is sometimes used instead of information gain. Information gain ratio biases against considering attributes with a large number of distinct values. However, attributes with very low information values then appear to receive an unfair advantage.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.

**example set *(exa)*** ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**normalize weights *(boolean)*** This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.

**sort weights *(boolean)*** This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the *sort direction* parameter.

**sort direction *(selection)*** This parameter is only available when the *sort weights* parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

## Tutorial Processes

### Calculating the weights of the attributes of the Golf data set

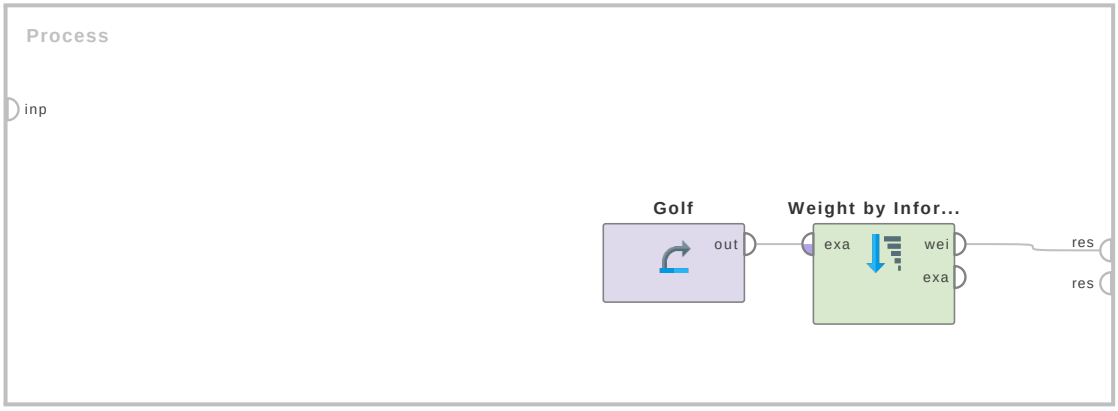
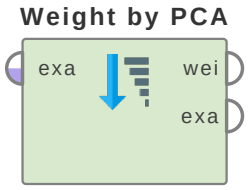



Figure 4.91: Tutorial process 'Calculating the weights of the attributes of the Golf data set'.

The 'Golf' data set is loaded using the Retrieve operator. The Weight by Information Gain Ratio operator is applied on it to calculate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, thus all the weights will be normalized in range 0 to 1. The sort weights parameter is set to true and the sort direction parameter is set to 'ascending', thus the results will be in ascending order of the weights. You can verify this by viewing the results of this process in the Results Workspace.

# Weight by PCA




This operator creates attribute weights of the ExampleSet by using a component created by the PCA. This operator behaves exactly the same way as if a PCA model is given to the Weight by Component Model operator.

## Description

The Weight by PCA operator generates attribute weights of the given ExampleSet using a component created by the PCA. The component is specified by the *component number* parameter. If the *normalize weights* parameter is not set to true, exact values of the selected component are used as attribute weights. The *normalize weights* parameter is usually set to true to spread the weights between 0 and 1. The attribute weights reflect the relevance of the attributes with respect to the class attribute. The higher the weight of an attribute, the more relevant it is considered.

Principal Component Analysis (PCA) is a mathematical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated attributes into a set of values of uncorrelated attributes called principal components. The number of principal components is less than or equal to the number of original attributes. This transformation is defined in such a way that the first principal component's variance is as high as possible (accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it should be orthogonal to (uncorrelated with) the preceding components.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

## Output Ports

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**normalize weights *(boolean)*** This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in the range from 0 to 1.

**sort weights *(boolean)*** This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the *sort direction* parameter.

**sort direction *(selection)*** This parameter is only available when the *sort weights* parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

**component number *(integer)*** This parameter specifies the number of the component that should be used as attribute weights.

## Tutorial Processes

### Calculating the attribute weights of the Sonar data set by PCA

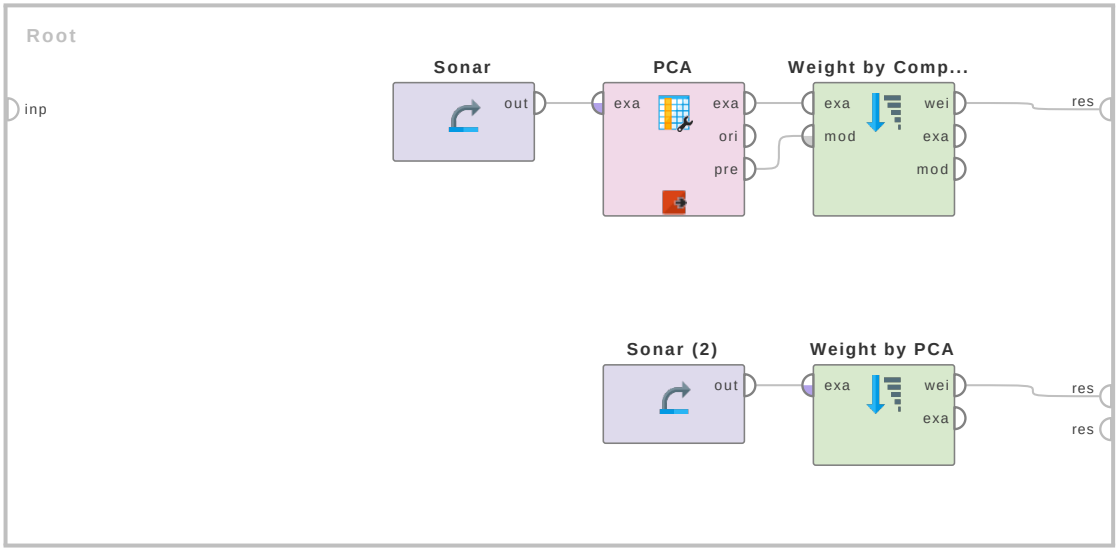
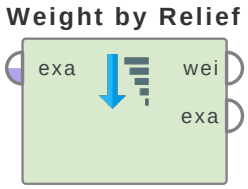



Figure 4.92: Tutorial process ‘Calculating the attribute weights of the Sonar data set by PCA’.

The ‘Sonar’ data set is loaded using the Retrieve operator. The PCA operator is applied on it. The dimensionality reduction parameter is set to ‘none’. A breakpoint is inserted here so that you can have a look at the components created by the PCA operator. Have a look at the EigenVectors generated by the PCA operator especially ‘PC1’ because it will be used as weights by using the Weight by Component Model operator. The Weight by Component Model operator is applied next. The ExampleSet and Model ports of the PCA operator are connected to the corresponding ports of the Weight by Component Model operator. The normalize weights and sort weights parameters are set to false, thus all the weights will be exactly the same as the selected component. The component number parameter is set to 1, thus ‘PC1’ will be used as attribute weights. The weights can be seen in the Results Workspace. You can see that these weights are exactly the same as the values of ‘PC1’.

In the second operator chain the Weight by PCA operator is applied on the ‘Sonar’ data set to perform exactly the same task. The parameters of the Weight by PCA operator are set exactly the same as the parameters of the Weight by Component Model operator. As it can be seen in the Results Workspace, exactly same weights are generated here.

# Weight by Relief




This operator calculates the relevance of the attributes by Relief. The key idea of Relief is to estimate the quality of features according to how well their values distinguish between the instances of the same and different classes that are near each other.

## Description

Relief is considered one of the most successful algorithms for assessing the quality of features due to its simplicity and effectiveness. The key idea of Relief is to estimate the quality of features according to how well their values distinguish between the instances of the same and different classes that are near each other. Relief measures the relevance of features by sampling examples and comparing the value of the current feature for the nearest example of the same and of a different class. This version also works for multiple classes and regression data sets. The resulting weights are normalized into the interval between 0 and 1 if the *normalize weights* parameter is set to true.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**normalize weights *(boolean)*** This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.

**sort weights *(boolean)*** This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the *sort direction* parameter.

**sort direction *(selection)*** This parameter is only available when the *sort weights* parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

**number of neighbors *(integer)*** This parameter specifies the number of nearest neighbors for relevance calculation.

**sample ratio *(real)*** This parameter specifies the ratio of examples to be used for determining the weights.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomizing examples of a subset. Using the same value of the *local random seed* will produce the same sample. Changing the value of this parameter changes the way examples are randomized, thus the sample will have a different set of examples.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Calculating the attribute weights of the Polynomial data set

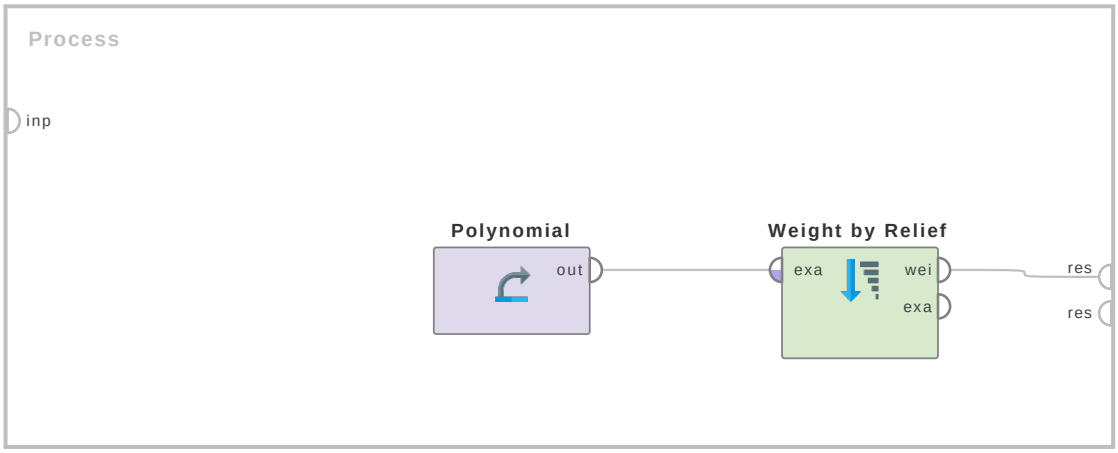
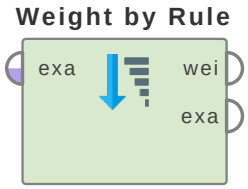



Figure 4.93: Tutorial process 'Calculating the attribute weights of the Polynomial data set'.

The 'Polynomial' data set is loaded using the Retrieve operator. The Weight by Relief operator is applied on it to calculate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, thus all the weights will be normalized in range 0 to 1. The sort weights parameter is set to true and the sort direction parameter is set to 'ascending', thus the results will be in ascending order of the weights. You can verify this by viewing the results of this process in the Results Workspace.

# Weight by Rule




This operator calculates the relevance of the attributes of the given ExampleSet by constructing a single rule for each attribute and calculating the errors.

## Description

The Weight by Rule operator calculates the weight of attributes with respect to the label attribute by constructing a single rule for each attribute and calculating the errors. The higher the weight of an attribute, the more relevant it is considered. Please note that this operator can be only applied on ExampleSets with nominal label.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**normalize weights *(boolean)*** This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.

**sort weights *(boolean)*** This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the *sort direction* parameter.

**sort direction *(selection)*** This parameter is only available when the *sort weights* parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

## Tutorial Processes

### Calculating the attribute weights of the Golf data set

The 'Golf' data set is loaded using the Retrieve operator. The Weight by Rule operator is applied on it to calculate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, thus all the weights will be normalized in range 0 to 1. The sort weights parameter is set to true and the sort direction parameter is set to 'ascending',

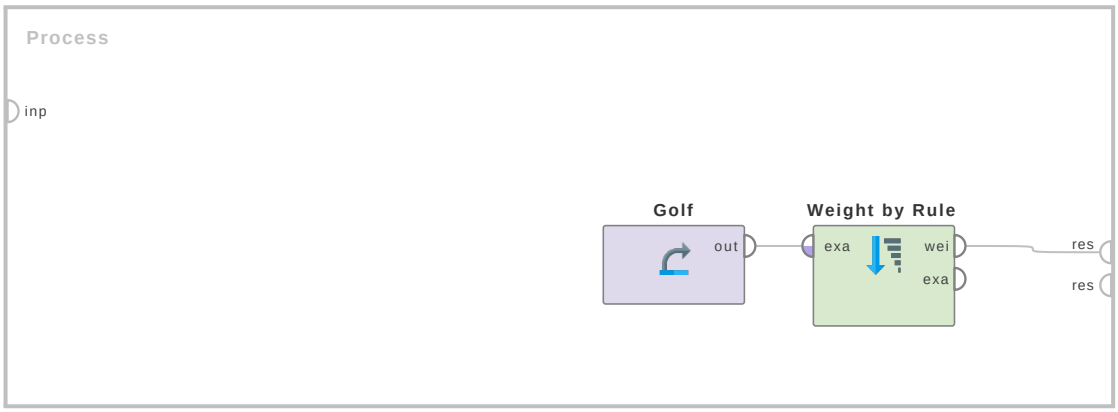
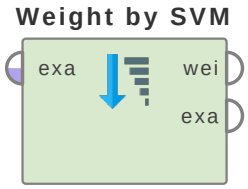



Figure 4.94: Tutorial process 'Calculating the attribute weights of the Golf data set'.

thus the results will be in ascending order of the weights. You can verify this by viewing the results of this process in the Results Workspace.

# Weight by SVM




This operator calculates the relevance of the attributes by computing for each attribute of the input ExampleSet the weight with respect to the class attribute. The coefficients of a hyperplane calculated by an SVM (Support Vector Machine) are set as attribute weights.

## Description

The Weight by SVM operator uses the coefficients of the normal vector of a linear SVM as attribute weights. In contrast to most of the SVM based operators available in RapidMiner, this operator works for multiple classes too. Please note that the attribute values still have to be numerical. This operator can be applied only on ExampleSets with numerical label. Please use appropriate preprocessing operators (type conversion operators) in order to ensure this. For more information about SVM please study the documentation of the SVM operator.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**normalize weights *(boolean)*** This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in the range from 0 to 1.

**sort weights *(boolean)*** This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of sorting is specified using the *sort direction* parameter.

**sort direction *(selection)*** This parameter is only available when the *sort weights* parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

**C *(real)*** This parameter specifies the SVM complexity weighting factor.

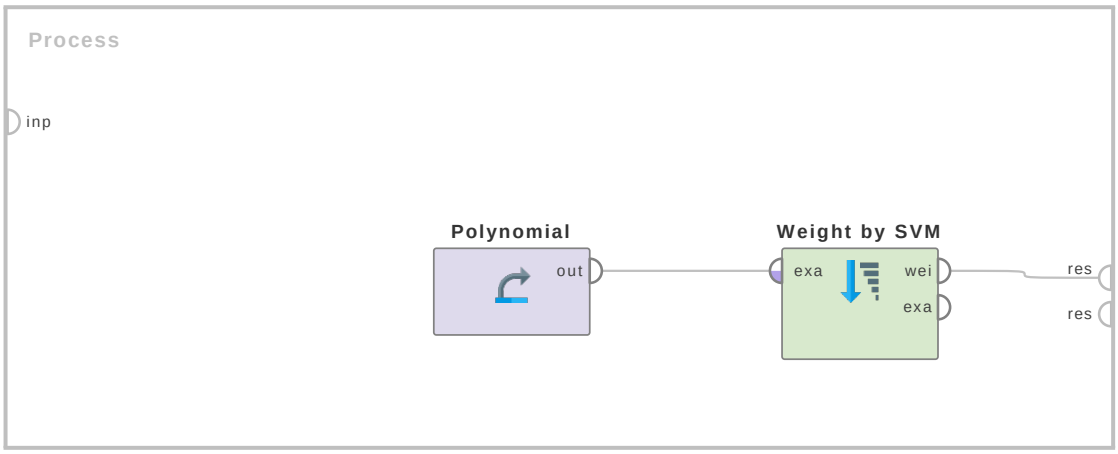


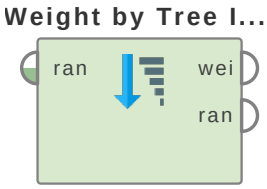

Figure 4.95: Tutorial process 'Calculating the weights of the attributes of the Polynomial data set'.

## Tutorial Processes

### Calculating the weights of the attributes of the Polynomial data set

The 'Polynomial' data set is loaded using the Retrieve operator. The Weight by SVM operator is applied on it to calculate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, thus all the weights will be normalized in the range 0 to 1. The sort weights parameter is set to true and the sort direction parameter is set to 'ascending', thus the results will be in ascending order of the weights. You can verify this by viewing the results of this process in the Results Workspace.

# Weight by Tree Importance




This operator calculates the weight of the attributes by analyzing the split points of a Random Forest model. The attributes with higher weight are considered more relevant and important.

## Description

This weighting schema will use a given random forest to extract the implicit importance of the used attributes. Therefore each node of each tree is visited and the benefit created by the respective split is retrieved. This benefit is summed per attribute, that had been used for the split. The mean benefit over all trees is used as importance.

This algorithm is implemented following the idea from "A comparison of random forest and its gini importance with standard chemometric methods for the feature selection and classification of spectral data" by Menze, Bjoen H et all (2009). It has been extended by additional criterias for computing the benefit created from a certain split. The original paper only mentioned Gini Index, this operator additionally supports the more reliable criterions Information Gain and Information Gain Ratio.

## Input Ports

**random forest *(ran)*** The input port expects a Random Forest model which is a voting model of random trees. It is output of the Random Forest operator in the attached Example Process.

## Output Ports

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.

**random forest *(ran)*** The Random Forest model that was given as input is passed without changing to the output through this port. This is usually used to reuse the same model in further operators or to view the model in the Results Workspace.

## Parameters

**criterion *(selection)*** This parameter specifies the criterion to be used for weighting the attributes. It can have one of the following values: information gain, gain ratio, gini index or accuracy.

**normalize weights *(boolean)*** This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in a range from 0 to 1.

## Tutorial Processes

### Calculating the attribute weights of the Golf data set using Random Forest model

The 'Golf' data set is loaded using the Retrieve operator. The Random Forest operator is applied on it to generate a random forest model. A breakpoint is inserted here so that you can have a

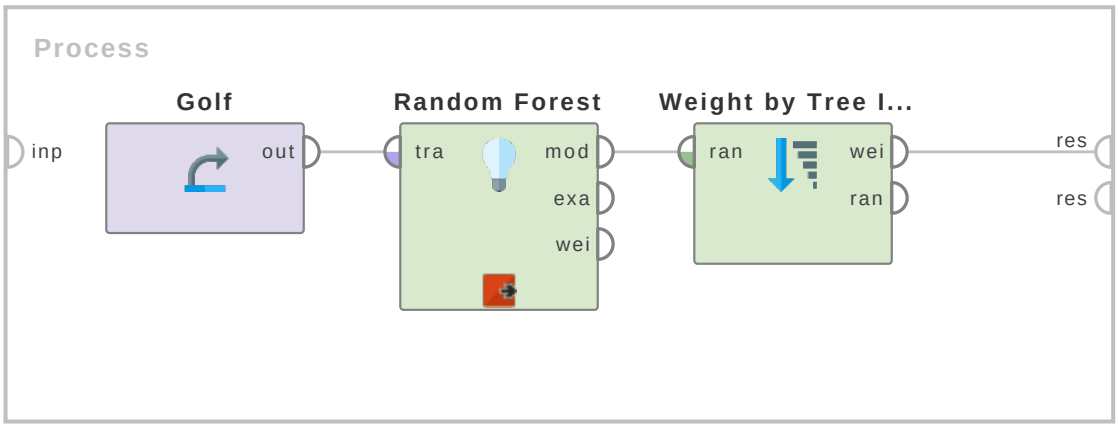
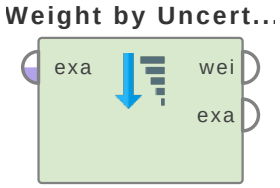



Figure 4.96: Tutorial process 'Calculating the attribute weights of the Golf data set using Random Forest model'.

look at the generated model. The resultant model is provided as input to the Weight by Tree Importance operator to calculate the weights of the attributes of the 'Golf' data set. All parameters are used with default values. The normalize weights parameter is set to true, thus all the weights will be normalized in a range from 0 to 1. You can verify this by viewing the results of this process in the Results Workspace.

# Weight by Uncertainty




This operator calculates the relevance of attributes of the given ExampleSet by measuring the symmetrical uncertainty with respect to the class.

## Description

The Weight by Uncertainty operator calculates the weight of attributes with respect to the label attribute by measuring the symmetrical uncertainty with respect to the class. The higher the weight of an attribute, the more relevant it is considered. Please note that this operator can be only applied on ExampleSets with nominal label. The relevance is calculated by the following formula:

*relevance = 2 * (P(Class) - P(Class | Attribute)) / P(Class) + P(Attribute)*

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**normalize weights *(boolean)*** This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in a range from 0 to 1.

**sort weights *(boolean)*** This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the *sort direction* parameter.

**sort direction *(selection)*** This parameter is only available when the *sort weights* parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

**normalize *(selection)*** This parameter indicates if the standard deviation should be divided by the minimum, maximum, or average of the attribute.

**number of bins *(integer)*** This parameter specifies the number of bins to be used.

## Tutorial Processes

### Calculating the attribute weights of the Golf data set

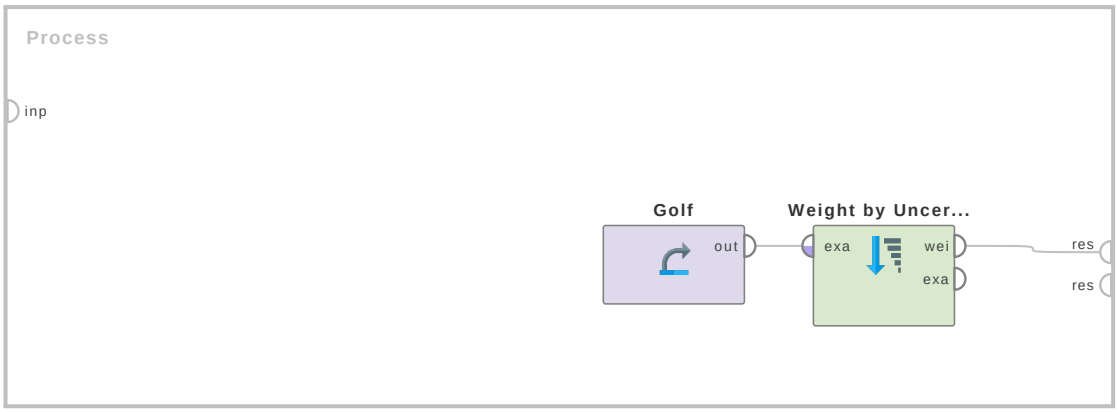
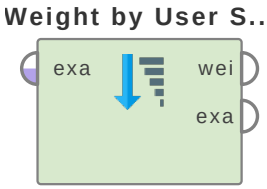



Figure 4.97: Tutorial process ‘Calculating the attribute weights of the Golf data set’.

The ‘Golf’ data set is loaded using the Retrieve operator. The Weight by Uncertainty operator is applied on it to calculate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, thus all the weights will be normalized in a range from 0 to 1. The sort weights parameter is set to true and the sort direction parameter is set to ‘ascending’, thus the results will be in ascending order of the weights. You can verify this by viewing the results of this process in the Results Workspace.

# Weight by User Specification




This operator assigns user-defined weights to the specified attributes. The attributes can be selected using regular expressions.

## Description

The Weight by User Specification operator assigns user-defined weights to the selected attributes of the given ExampleSet. The higher the weight of an attribute, the more relevant it is considered. Unlike many other weighting operators, this operator can be applied on ExampleSets with both nominal or numerical label.

The *name regex to weights* parameter is used for selecting the attributes and assigning weights to them. The attributes are selected through regular expressions. Multiple regular expressions can be used for different attribute selections. Please note that the weights defined in the regular expression list are set in the order as they are defined in the list, i.e. weights can overwrite weights set before.

If the *distribute weights* parameter is set to true, then the weight specified in the *name regex to weights* parameter is divided equally into all the attributes that match the regular expression. The *default weight* parameter specifies weight of all those attributes that do not match any regular expression. Please Study the attached Example Process for more information.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

## Output Ports

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.

**example set *(exa)*** ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**normalize weights *(boolean)*** This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.

**sort weights *(boolean)*** This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the *sort direction* parameter.

**sort direction *(selection)*** This parameter is only available when the *sort weights* parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

**name regex to weights *(list)*** This parameter is used for selecting the attributes and assigning weights to them. The attributes are selected through regular expressions. Multiple regular expressions can be used for different attribute selections. Please note that the weights defined in the regular expression list are set in the order as they are defined in the list, i.e. weights can overwrite weights set before.

**distribute weights *(boolean)*** If this parameter is set to true, the weight specified in the *name regex to weights* parameter is split and distributed equally among the attributes matching the corresponding regular expressions.

**default weight *(real)*** This parameter specifies the weight of all those attributes that do not match any regular expression.

## Tutorial Processes

### Manually setting the attribute weights of the Golf data set

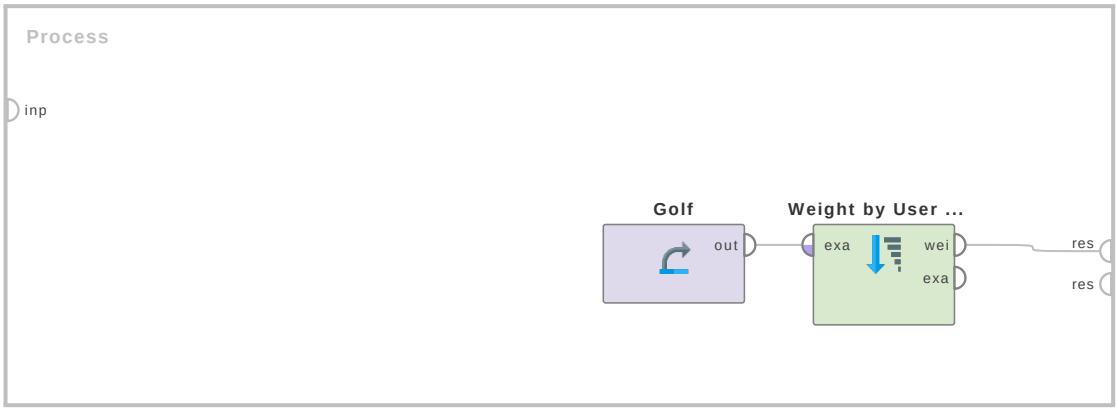
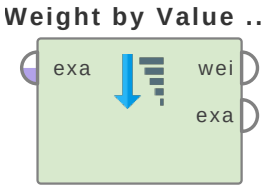



Figure 4.98: Tutorial process 'Manually setting the attribute weights of the Golf data set'.

The 'Golf' data set is loaded using the Retrieve operator. The Weight by User Specification operator is applied on it to assign the attribute weights. The normalize weights parameter is set to false, thus the weights will not be normalized. The sort weights parameter is set to true and the sort direction parameter is set to 'ascending', thus the results will be in ascending order of the weights. The name regex to weights parameter is used for assigning weights through regular expressions. Only one regular expression is defined in this process. This regular expression selects all those attributes that have the alphabet 'i' in their names. The matching attributes (i.e. Humidity and Wind) are assigned weight 4.0. All those attributes that do not match any regular expression (i.e.. Temperature and Outlook) are assigned the default weight which is defined by the default weight parameter. In this process it is set to 1.0. Run the process and you will see that the attributes that matched the regular expression get corresponding weight (i.e. 4.0) and the remaining attributes get default weight (i.e. 1.0). Now set the distribute weights parameter to true and run the process again. Now the weight 4.0 will be equally split into the Wind and Humidity attributes, thus their weight will be set to 2.0. You can verify this by viewing results of the process in the Results Workspace.

# Weight by Value Average




This operator uses a corpus of examples to characterize a single class by setting feature weights.

## Description

This operator uses a corpus of examples to characterize a single class by setting feature weights. Characteristic features receive higher weights than less characteristic features. The weight for a feature is determined by calculating the average value of this feature for all examples of the target class.

This operator assumes that the feature values characterize the importance of this feature for an example (e.g. TFIDF or others). Therefore, this operator is mainly used on textual data based on TFIDF weighting schemes. To extract such feature values from text collections you can use the Text plugin.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet.

## Output Ports

**weights *(wei)*** This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

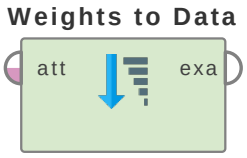
**normalize weights *(boolean)*** Activates the normalization of all weights.

**sort weights *(boolean)*** This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the *sort direction* parameter.

**sort direction *(selection)*** This parameter is only available when the *sort weights* parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

**class to characterize *(string)*** The target class for which to find characteristic feature weights.

# Weights to Data




This operator takes attribute weights as input and delivers an ExampleSet with attribute names and corresponding weights as output.

## Description

The Weights to Data operator takes attribute weights as input and delivers an ExampleSet with attribute names and corresponding weights as output. The resultant ExampleSet has two attributes 'Attribute' and 'Weight'. The 'Attribute' attribute stores the names of attributes and the 'Weight' attribute stores the weight of the corresponding attribute. This ExampleSet has *n* number of examples; where *n* is the number of attributes in the input weights vector. There are numerous operators that provide attribute weights at 'Modeling/Attribute Weighting' in the Operators Window.

## Input Ports

**attribute weights *(att)*** This input port expects weights of attributes. It is the output of the Weight by Chi Squared Statistic operator in the attached Example Process.

## Output Ports

**example set *(exa)*** The ExampleSet that contains attribute names and corresponding weights is delivered through this port.

## Tutorial Processes

### Calculating the weights of the attributes of the Golf data set and storing them in an ExampleSet

The 'Golf' data set is loaded using the Retrieve operator. The Weight by Chi Squared Statistic operator is applied on it to calculate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, thus all the weights will be normalized in range 0 to 1. The sort weights parameter is set to true and the sort direction parameter is set to 'ascending', thus the results will be in ascending order of the weights. A breakpoint is inserted here to show the weights produced by the Weight by Chi Squared Statistic operator. These weights are provided as input to the Weights to Data operator which stores these weights in form of an ExampleSet. The ExampleSet can be seen in the Results Workspace.

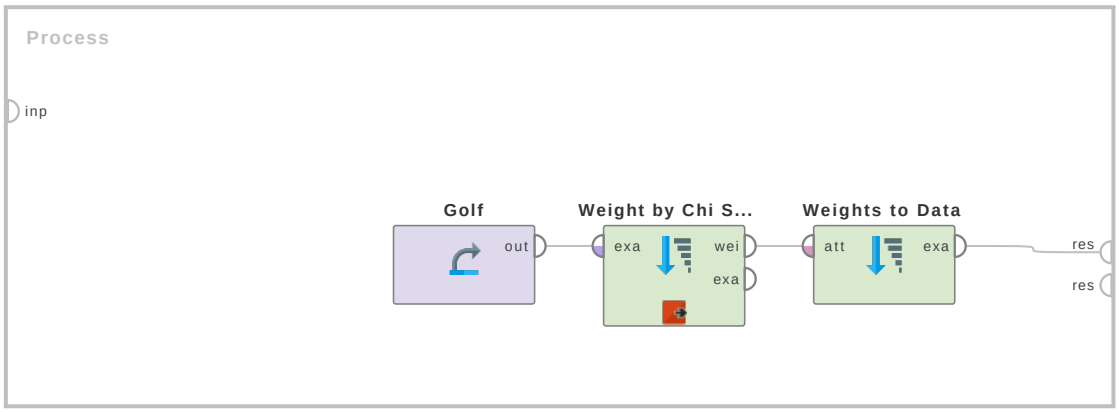
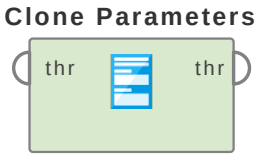



Figure 4.99: Tutorial process 'Calculating the weights of the attributes of the Golf data set and storing them in an ExampleSet'.

# 4.7 Optimization

## 4.7.1 Parameters

### Clone Parameters




This operator copies parameter values of the specified source parameters into parameter values of the specified target parameters. This operator is a more generic form of the Set Parameters operator. This operator is mostly used for applying an optimal set of parameters of one operator to another similar operator.

#### Description

The Clone Parameters operator does not require an input whereas the Set Parameters operator takes a set of parameters as input. The Clone Parameters operator can be used for assigning multiple source parameters to multiple target parameters. The source and target parameters can be specified through the *name map* parameter. This menu has two columns: 'source' and 'target'. The 'source' column is used for specifying the names of the desired parameters whose values will be copied into the target parameters. The 'target' column specifies the names of the target parameters. Both these columns expect values in 'operator.parameter' format where 'operator' is the name of the operator and 'parameter' is the desired parameter. Multiple parameters can be specified by adding more rows in the *name map* parameter.

This operator is a very generic form of the Set Parameters operator. It differs from the Set Parameters operator because it does not require a ParameterSet as input. It simply reads a parameter value from a source and uses it to set the parameter value of a target parameter. This operator is more generic than the Set Parameters operator and could completely replace it. It is most useful, if you need a parameter which is optimized more than once within the optimization loop - the Set Parameters operator cannot be used here. The fact that this operator does not require an explicit ParameterSet as input and that it does not require the source and target parameters to be exactly the same, makes this operator a lot more generic and powerful than

the Set Parameters operator. For more information about the Set Parameters operator please study its documentation

## Differentiation

- **Set Parameters** The Clone Parameters operator is a very generic form of the Set Parameters operator. It differs from the Set Parameters operator because it does not require a ParameterSet as input. It simply reads a parameter value from a source and uses it to set the parameter value of a target parameter. This operator is more generic than the Set Parameters operator and could completely replace it. See page 679 for details.

## Input Ports

**through *(thr)*** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of this operator is available at the first *through* output port.

## Output Ports

**through *(thr)*** The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port. This operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of this operator is delivered at the first *through* output port.

## Parameters

**name map *(list)*** The Clone Parameters operator assigns parameter values of the source to those of the target of the specified operators which can be specified through the *name map* parameter. This menu has two columns: 'source' and 'target'. The 'source' column is used for specifying the names of the desired parameters whose values will be copied into the target parameters. The 'target' column specifies the names of the target parameters. Both these columns expect values in 'operator.parameter' format where 'operator' is the name of the operator and 'parameter' is the desired parameter. Multiple parameters can be specified by adding more rows in the *name map* parameter.

## Related Documents

- **Set Parameters** (page 679)

## Tutorial Processes

### Introduction to the Clone Parameters operator

This Example Process highlights the fact that the Clone Parameters operator is a very generic operator which can be used for assigning any parameter value to any other parameter in the

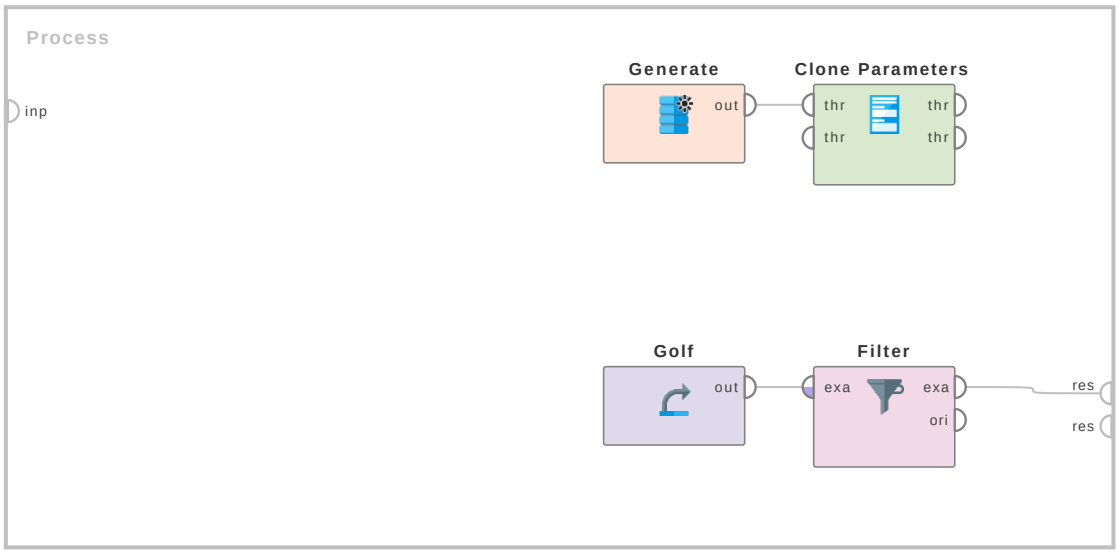
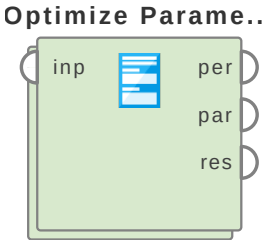



Figure 4.100: Tutorial process 'Introduction to the Clone Parameters operator'.

process. The task performed in this process does not serve any useful purpose. For understanding a useful scenario of this operator please study the Example Process of the Set Parameters operator.

The Generate Data operator is used for generating an ExampleSet with 10 examples (i.e. number examples parameter = 10). Then the Filter Example Range operator is applied on the 'Golf' ExampleSet. Its first example and last example parameters are set to 1. Thus only the first row of the 'Golf' data set should make it to the output. But have a look at the parameters of the Clone Parameters operator. It assigns the value of the number examples parameter of the Generate Data operator to the last example parameter of the Filter Example Range operator. Thus the last example parameter is set to 10 (previously 1). Therefore instead of only 1 row, the first 10 rows of the 'Golf' data set make it to the output. The resultant ExampleSet can be seen in the Results Workspace. You can see after the execution of the process that the value of the last example parameter has also been changed from 1 to 10. This is how the Clone Parameters operator can be used to dynamically copy parameter values.

# Optimize Parameters (Evolutionary)




This operator finds the optimal values of the selected parameters of the operators in its subprocess. It uses an evolutionary computation approach.

## Description

This operator finds the optimal values for a set of parameters using an evolutionary approach which is often more appropriate than a grid search (as in the Optimize Parameters (Grid) operator) or a greedy search (as in the Optimize Parameters (Quadratic) operator) and leads to better results. This is a nested operator i.e. it has a subprocess. It executes its subprocess for a multiple number of times to find optimal values for the specified parameters.

This operator delivers the optimal parameter values through the *parameter* port which can also be written into a file with the Write Parameters operator. This parameter set can be read in another process using the Read Parameters operator. The performance vector for optimal values of parameters is delivered through the *performance* port. Any additional results of the subprocess are delivered through the *result* ports.

Other parameter optimization schemes are also available in RapidMiner. The Optimize Parameters (Evolutionary) operator might be useful if the best ranges and dependencies are not known at all. Another operator which works similar to this parameter optimization operator is the Loop Parameters operator. In contrast to the optimization operators, this operator simply iterates through all parameter combinations. This might be especially useful for plotting purposes.

## Differentiation

- **Optimize Parameters (Grid)** The Optimize Parameters (Grid) operator executes its subprocess for all combinations of the selected values of the parameters and then delivers the optimal parameter values. See page **??** for details.

## Input Ports

**input *(inp)*** This operator can have multiple inputs. When one input is connected, another *input* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *input* port of this operator is available at the first *input* port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at the subprocess level.

## Output Ports

**performance *(per)*** This port delivers the Performance Vector for the optimal values of the selected parameters. A Performance Vector is a list of performance criteria values.

**parameter *(par)*** This port delivers the optimal values of the selected parameters. This optimal parameter set can be written into a file with the Write Parameters operator. The written parameter set can be read in another process using the Read Parameters operator.

**result *(res)*** Any additional results of the subprocess are delivered through the *result* ports. This operator can have multiple outputs. When one *result* port is connected, another *result* port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first *result* port of the subprocess is delivered at the first *result* port of the operator. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports.

## Parameters

**edit parameter settings *(menu)*** The parameters are selected through the *edit parameter settings* menu. You can select the parameters and their possible values through this menu. This menu has an *Operators* window which lists all the operators in the subprocess of this operator. When you click on any operator in the *Operators* window, all parameters of that operator are listed in the *Parameters* window. You can select any parameter through the arrow keys of the menu. The selected parameters are listed in the *Selected Parameters* window. Only those parameters should be selected for which you want to find optimal values. This operator finds optimal values of the parameters in the specified range. The range of every selected parameter should be specified. When you click on any selected parameter (parameter in the *Selected Parameters* window) the *Grid/Range* option is enabled. This option allows you to specify the range of values of the selected parameters. The *Min* and *Max* fields are for specifying the lower and upper bounds of the range respectively. The *steps* and *scale* options are disabled for this operator. Note that only numerical parameters are displayed, since this operator does not support non numerical parameters.

**error handling *(selection)*** This parameter allows you to select the method for handling errors occurring during the execution of the inner process. It has the following options:

- **fail_on_error** In case an error occurs, the execution of the process will fail with an error message.
- **ignore_error** In case an error occurs, the error will be ignored and the execution of the process will continue with the next iteration.

**max generations *(integer)*** This parameter specifies the number of generations after which the algorithm should be terminated.

**use early stopping *(boolean)*** This parameter enables early stopping. If not set to true, always the maximum number of generations are performed.

**generations without improval *(integer)*** This parameter is only available when the *use early stopping* parameter is set to true. This parameter specifies the stop criterion for early stopping i.e. it stops after *n* generations without improvement in the performance. *n* is specified by this parameter.

**specify population size *(boolean)*** This parameter specifies the size of the population. If it is not set to true, one individual per example of the given ExampleSet is used.

**population size *(integer)*** This parameter is only available when the *specify population size* parameter is set to true. This parameter specifies the population size i.e. the number of individuals per generation.

**keep best *(boolean)*** This parameter specifies if the best individual should survive. This is also called elitist selection. Retaining the best individuals in a generation unchanged in the next generation, is called elitism or elitist selection.

**mutation type *(selection)*** This parameter specifies the type of the mutation operator.

**selection type *(selection)*** This parameter specifies the selection scheme of this evolutionary algorithms.

**tournament fraction *(real)*** This parameter is only available when the *selection type* parameter is set to 'tournament'. It specifies the fraction of the current population which should be used as tournament members.

**crossover prob *(real)*** The probability for an individual to be selected for crossover is specified by this parameter.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

**show convergence plot *(boolean)*** This parameter indicates if a dialog with a convergence plot should be drawn.

## Related Documents

- **Optimize Parameters (Grid)** (page **??**)
- **Optimize Parameters (Quadratic)** (page 677)

## Tutorial Processes

### Finding optimal values of parameters of the SVM operator through the Optimize Parameters (Evolutionary) operator

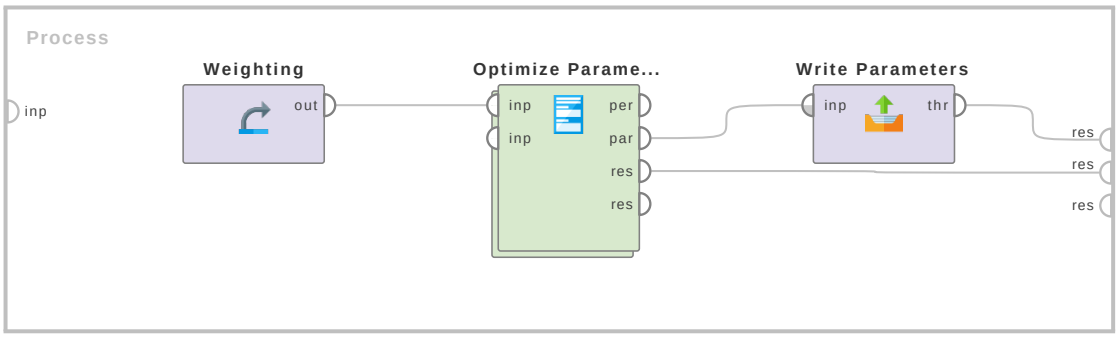



Figure 4.101: Tutorial process 'Finding optimal values of parameters of the SVM operator through the Optimize Parameters (Evolutionary) operator'.

The 'Weighting' data set is loaded using the Retrieve operator. The Optimize Parameters (Evolutionary) operator is applied on it. Have a look at the Edit Parameter Settings parameter of the Optimize Parameters (Evolutionary) operator. You can see in the Selected Parameters window that the C and gamma parameters of the SVM operator are selected. Click on the SVM.C parameter in the Selected Parameters window, you will see that the range of the C parameter is set from 0.001 to 100000. Now, click on the SVM.gamma parameter in the Selected Parameters window, you will see that the range of the gamma parameter is set from 0.001 to 1.5. In every iteration of the subprocess, the value of the C and/or gamma parameters of the SVM(LibSVM) operator is changed in search of optimal values.

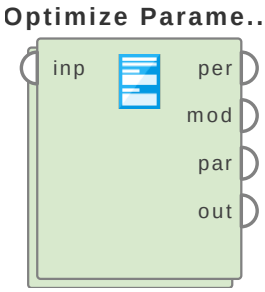
Have a look at the subprocess of the Optimize Parameters (Evolutionary) operator. First the data is split into two equal partitions using the Split Data operator. The SVM (LibSVM) operator is applied on one partition. The resultant classification model is applied using two Apply Model operators on both the partitions. The statistical performance of the SVM model on both testing and training partitions is measured using the Performance (Classification) operators. At the end the Log operator is used to store the required results.

The log parameter of the Log operator stores five things. The iterations of the Optimize Parameters (Evolutionary) operator are counted by the apply-count of the SVM operator. This is stored in a column named 'Count'.The value of the classification error parameter of the Performance (Classification) operator that was applied on the Training partition is stored in a column named 'Training Error'. The value of the classification error parameter of the Performance (Classification) operator that was applied on the Testing partition is stored in a column named 'Testing Error'. The value of the C parameter of the SVM (LibSVM) operator is stored in a column named 'SVM C'.The value of the gamma parameter of the SVM (LibSVM) operator is stored in a column named 'SVM gamma'. Also note that the stored information will be written into a file as specified in the filename parameter.

At the end of the process, the Write Parameters operator is used for writing the optimal parameter set in a file. This file can be read using the Read Parameters operator to use these parameter values in another process.

Run the process and turn to the Results Workspace. You can see that the optimal parameter set has the following values: SVM.C = 56462 and SVM.gamma = 0.115 approximately. Now have a look at the values saved by the Log operator to verify these values. Switch to Table View to see the stored values in tabular form. You can see that the minimum Testing Error is 0.064 (in 20th iteration). The values of the C and gamma parameters for this iteration are the same as given in the optimal parameter set.

# Optimize Parameters (Grid)




This Operator finds the optimal values of the selected parameters for the Operators in its subprocess.

## Description

The Optimize Parameters (Grid) Operator is a nested Operator. It executes the subprocess for all combinations of selected values of the parameters and then delivers the optimal parameter values through the *parameter set* port. The performance vector for optimal values of parameters is delivered through the *performance* port and the associated model (if any) through the *model* port. Any additional results of the best run are delivered through the *output* ports. Which parameters are optimal is based on the performance value delivered to the inner performance port.

The entire configuration of this Operator is done through the *edit parameter settings* parameter. Complete description of this parameter can be found in the parameters section.

This Operator returns an optimal parameter set which can also be written to a file with the Write Parameters Operator. This parameter set can be read in another process using the Read Parameters Operator and then be applied using the Set Parameters Operator.

The inner performance port can be used to log the performance of the inner subprocess. A log is created automatically to capture the number of the run, the parameter settings and the main criterion or all criteria of the delivered performance vector, depending on the parameter *log all criteria*. This can be disabled by deselecting *log performance*. The inner performance port is also used to determine the best model by comparing the fitness of the performance of the different iterations.

Please note that this Operator has two modes: synchronized and non-synchronized. They depend on the setting of the *synchronize* parameter. In the latter, all parameter combinations are generated and the subprocess is executed for each combination. In the synchronized mode, no combinations are created but the parameter values are treated as a list of combinations. For the iteration over a single parameter there is no difference between both modes. Please note that the number of parameter possibilities must be the same for all parameters in the synchronized mode. As an Example, having two boolean parameters A and B (both with true/false as possible parameter settings) will produce four combinations in non-synchronized mode (t/t, f/t, t/f, f/f) and two combinations in synchronized mode (t/t, f/f).

If the *synchronize* parameter is not set to true, selecting a large number of parameters and/or large number of steps (or possible values of parameters) results in a huge number of combinations. For example, if you select 3 parameters and 25 steps for each parameter then the total number of combinations would be above 17576 (i.e. 26 x 26 x 26). The subprocess is executed for all possible combinations. Running a subprocess for such a huge number of iterations will take a lot of time. So always carefully limit the parameters and their steps.

## Differentiation

Other parameter optimization schemes are also available. The Optimize Parameters (Evolutionary) Operator might be useful if the best ranges and dependencies are not known at all. Another Operator which works similar to this parameter optimization Operator is the Loop Parameters Operator. In contrast to the optimization Operators, this Operator simply iterates through all parameter combinations. This might be especially useful for plotting and logging purposes.

- **Optimize Parameters (Evolutionary)**

  The Optimize Parameters (Evolutionary) Operator finds the optimal values for a set of parameters using an evolutionary approach which is often more appropriate than a grid search (as in the Optimize Parameters (Grid) Operator) or a greedy search (as in the Optimize Parameters (Quadratic) Operator) and leads to better results. The Optimize Parameters (Evolutionary) Operator might be useful if the best ranges and dependencies are not known at all.

  See page 669 for details.

- **Optimize Parameters (Quadratic)**

  The Optimize Parameters (Quadratic) Operator finds the optimal values using a quadratic interaction model. First it runs the same iterations as this operator. From the collected parameter set/performance pairs it tries to calculate a new parameter set that might lie in between the given grid lines. The result will either be the best performance from the original runs or the from the newly calculated parameter set.

  See page 677 for details.

## Tutorial Processes

### Finding optimal values of parameters of the SVM Operator

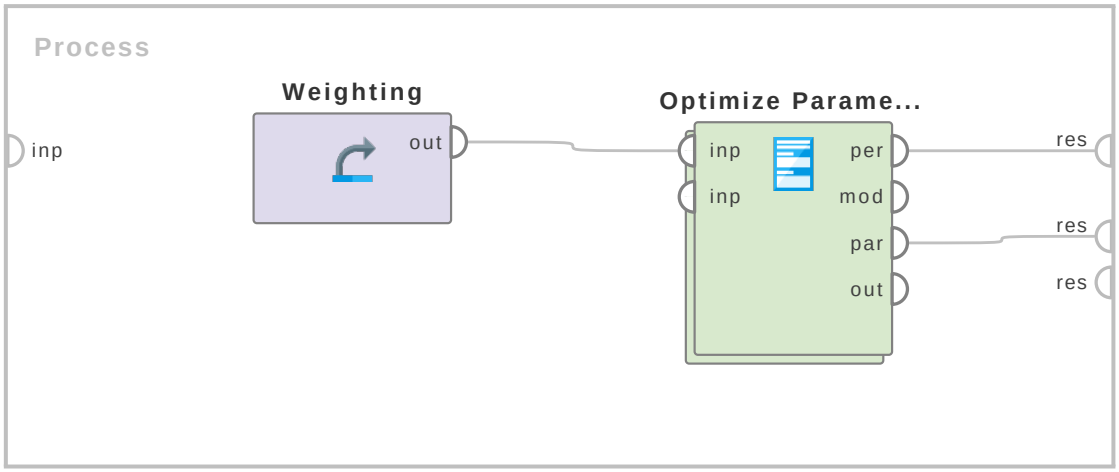



Figure 4.102: Tutorial process 'Finding optimal values of parameters of the SVM Operator'.

The 'Weighting' data set is loaded using the Retrieve Operator. The Optimize Parameters (Grid) Operator is applied on it. Have a look at the Edit Parameter Settings parameter of the

Optimize Parameters (Grid) Operator. You can see in the Selected Parameters window that the C and gamma parameters of the SVM Operator are selected. Click on the SVM.C parameter in the Selected Parameters window, you will see that the range of the C parameter is set from 0.001 to 100000. 11 values are selected (in 10 steps) logarithmically. Now, click on the SVM.gamma parameter in the Selected Parameters window, you will see that the range of the gamma parameter is set from 0.001 to 1.5. 11 values are selected (in 10 steps) logarithmically. There are 11 possible values of 2 parameters, thus there are 121 ( i.e. 11 x 11) combinations. The subprocess will be executed for all combinations of these values, thus it will iterate 121 times. In every iteration, the values of the C and/or gamma parameters of the SVM(LibSVM) Operator are changed. The value of the C parameter is 0.001 in the first iteration. The value is increased logarithmically until it reaches 100000 in the last iteration. Similarly, the value of the gamma parameter is 0.001 in the first iteration. The value is increased logarithmically until it reaches 1.5 in the last iteration.

Have a look at the subprocess of the Optimize Parameters (Grid) Operator. First the data is split into two equal partitions using the Split Data Operator. The SVM (LibSVM) Operator is applied on one partition. The resultant classification model is applied using a Apply Model Operator on the second partition. The statistical performance of the SVM model on the testing partition is measured using the Performance (Classification) Operators. The nested Operator also logs the performance and parameters for each iteration.

Run the process and turn to the Results View. You can see that the optimal parameter set has the following values: SVM.C = 398.107 and SVM.gamma = 0.001. Now have a look at the values logged by the Optimize Parameter (Grid) Operator to verify these values. You can see that the minimum Testing Error is 0.02 (in 8th iteration). The values of the C and gamma parameters for this iteration are the same as given in the optimal parameter set.

## Parameters

**edit parameter settings *(menu)*** The parameters are selected through the *edit parameter settings* menu. You can select the parameters and their possible values through this menu. This menu has an *Operators* window which lists all the operators in the subprocess of this Operator. When you click on any Operator in the *Operators* window, all parameters of that Operator are listed in the *Parameters* window. You can select any parameter through the arrow keys of the menu. The selected parameters are listed in the *Selected Parameters* window. Only those parameters should be selected for which you want to iterate the subprocess. This Operator iterates through parameter values in the specified range. The range of every selected parameter should be specified. When you click on any selected parameter (parameter in *Selected Parameters* window), the *Grid/Range* and *Value List* option is enabled. These options allow you to specify the range of values of the selected parameters. The *Min* and *Max* fields are for specifying the lower and upper bounds of the range respectively. As all values within this range cannot be checked, the *steps* field allows you to specify the number of values to be checked from the specified range. Finally the *scale* option allows you to select the pattern of these values. You can also specify the values in form of a list.

**error handling *(selection)*** This parameter allows you to select the method for handling errors occurring during the execution of the inner process. It has the following options:

- **fail_on_error** In case an error occurs, the execution of the process will fail with an error message.
- **ignore_error** In case an error occurs, the error will be ignored and the execution of the process will continue with the next iteration.

**log performance** ***(boolean)*** This parameter will only be visible if the inner performance port is connected. If it is connected, the main criterion of the performance vector will be automatically logged with the parameter set if this parameter is set to true.

**log all criteria** ***(boolean)*** This parameter allows for more logging. If set to true, all performance criteria will be logged.

**synchronize** ***(boolean)*** This Operator has two modes: synchronized and non-synchronized. They depend on the setting of this parameter. If it is set to false, all parameter combinations are generated and the inner Operators are applied for each combination. If it is set to true, no combinations are created but the parameter values are treated as a list of combinations. For the iteration over a single parameter there is no difference between both modes. Please note that the number of parameter possibilities must be the same for all parameters in the synchronized mode.

**enable parallel execution** ***(boolean)*** This parameter enables the parallel execution of the subprocess. Please disable the parallel execution if you run into memory problems.

## Input Ports

**input** ***(inp)*** This Operator can have multiple inputs. When one input is connected, another *input* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *input* port of this Operator is available at the first *input* port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at the subprocess level.

## Output Ports

**performance** ***(per)*** This port delivers the Performance Vector for the optimal values of the selected parameters. A Performance Vector is a list of performance criteria values.

**model** ***(mod)*** This port delivers the Model for the optimal values of the selected parameters.

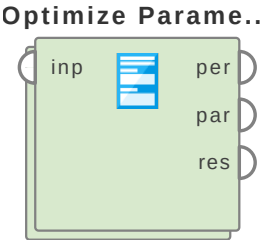
**parameters** ***(par)*** This port delivers the optimal values of the selected parameters. This optimal parameter set can also be written to a file with the Write Parameters operator. The written parameter set can be read in another process using the Read Parameters operator.

**output** ***(out)*** Any results of the subprocess are delivered through the *output* ports. This Operator can have multiple outputs. When one *output* port is connected, another *output* port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first *output* port of the subprocess is delivered at the first *output*port of the Operator. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports.

## Related Documents

- **Optimize Parameters (Evolutionary)** (page 669)
- **Optimize Parameters (Quadratic)** (page 677)

# Optimize Parameters (Quadratic)




This operator finds the optimal values for parameters using a quadratic interaction model.

## Description

This operator finds the optimal values for a set of parameters using a quadratic interaction model. The parameter *parameters* is a list of key value pairs where the keys are of the form OperatorName.parameter_name and the value is a comma separated list of values (as for the GridParameterOptimization operator).

The operator returns an optimal *ParameterSet* which can as well be written to a file with a *ParameterSetLoader*. This parameter set can be read in another process using an *ParameterSetLoader*.

The file format of the parameter set file is straightforward and can also easily be generated by external applications. Each line is of the form operator_name.parameter_name = value .

## Differentiation

- **Optimize Parameters (Evolutionary)** The Optimize Parameters (Evolutionary) operator finds the optimal values for a set of parameters using an evolutionary approach which is often more appropriate than a grid search (as in the Optimize Parameters (Grid) operator) or a greedy search (as in the Optimize Parameters (Quadratic) operator) and leads to better results. The Optimize Parameters (Evolutionary) operator might be useful if the best ranges and dependencies are not known at all. See page 669 for details.

- **Optimize Parameters (Grid)** The Optimize Parameters (Grid) operator executes its subprocess for all combinations of the selected values of the parameters and then delivers the optimal parameter values. See page **??** for details.

## Input Ports

**input *(inp)*** This operator can have multiple inputs. When one input is connected, another *input* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *input* port of this operator is available at the first *input* port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at the subprocess level.

## Output Ports

**performance *(per)*** This port delivers the Performance Vector for the optimal values of the selected parameters. A Performance Vector is a list of performance criteria values.

**parameters *(par)*** This port delivers the optimal values of the selected parameters. This optimal parameter set can also be written to a file with the Write Parameters operator. The written parameter set can be read in another process using the Read Parameters operator.

**result *(res)*** Any additional results of the subprocess are delivered through the *result* ports. This operator can have multiple outputs. When one *result* port is connected, another *result* port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first *result* port of the subprocess is delivered at the first *result* port of the operator. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports.

## Parameters

**configure operator** Configure this operator by means of a Wizard.

**parameters *(list)*** The parameters

**error handling *(selection)*** This parameter allows you to select the method for handling errors occurring during the execution of the inner process. It has the following options:

- **fail_on_error** In case an error occurs, the execution of the process will fail with an error message.
- **ignore_error** In case an error occurs, the error will be ignored and the execution of the process will continue with the next iteration.

**if exceeds region *(selection)*** What to do if region is exceeded.

- **ignore**
- **clip**
- **fail**

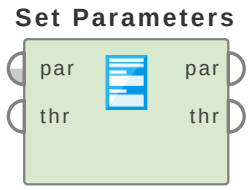
**if exceeds range *(selection)*** What to do if range is exceeded.

- **ignore**
- **clip**
- **fail**

## Related Documents

- **Optimize Parameters (Evolutionary)** (page 669)
- **Optimize Parameters (Grid)** (page **??**)

# Set Parameters




This operator applies a set of parameters to the specified operators. This operator is mostly used for applying an optimal set of parameters of one operator to another similar operator.

## Description

The Set Parameters operator takes a set of parameters as input. Operators like Optimize Parameters (Grid) or Read Parameters can be used as a source of parameter set. The Set Parameters operator takes this parameter set and assigns these parameter values to the parameters of the specified operator which can be specified through the *name map* parameter. This menu has two columns: 'set operator name' and 'operator name'. The 'set operator name' column is used for specifying the name of the operator (or the name of the parameter set) that generated this parameter set and the 'operator name' column specifies the name of the target operator i.e. the operator that will use this parameter set.

This operator can be very useful, e.g. in the following scenario. If one wants to find the best parameters for a certain learning scheme, one is usually also interested in the model generated with these optimal parameters. Finding the best parameter set is easily possible by operators like the Optimize Parameters (Grid) operator. But generating a model with these optimal parameters is not possible because such parameter optimization operators do not return the IOObjects produced within, but only a parameter set and a performance vector. This is, because the parameter optimization operators know nothing about models, but only about the performance vectors produced within and producing performance vectors does not necessarily require a model. To solve this problem, one can use the Set Parameters operator. Usually, a process with the Set Parameters operator contains at least two operators of the same type, typically a learner. One learner may be an inner operator of a parameter optimization scheme and may be named 'Learner', whereas a second learner of the same type named 'OptimalLearner' comes after the parameter optimization scheme and should use the optimal parameter set returned by the optimization scheme. The Set Parameters operator takes the parameter set returned by the optimization scheme as input. The name 'Learner', is specified in the 'set operator name' column and the name 'OptimalLearner' is specified in the 'operator name' column of the *name map* parameter. Each parameter in this list maps the name of an operator that was used during the optimization (in our case this is 'Learner') to an operator that should now use these parameters (in our case this is 'OptimalLearner'). An important thing to note here is the sequence of operators. The 'Learner', operator should be first in sequence, followed by the Set Parameters operator and finally the 'OptimalLearner' operator.

## Differentiation

- **Clone Parameters** The Clone Parameters operator is a very generic form of the Set Parameters operator. It differs from the Set Parameters operator because it does not require a ParameterSet as input. It simply reads a parameter value from a source and uses it to set the parameter value of a target parameter. This operator is more generic than the Set Parameters operator and could completely replace it. See page 666 for details.

## Input Ports

**parameter set *(par)*** This input port expects a parameter set. It is the output of the Optimize Parameters (Grid) operator in the attached Example Process. The output of other operators like the Read Parameters operator can also be used as a source of a parameter set.

## Parameters

**name map *(list)*** The Set Parameters operator takes a parameter set and assigns these parameter values to the parameters of the specified operator which can be specified through the *name map* parameter. This menu has two columns i.e. 'set operator name' and 'operator name'. The 'set operator name' column is used for specifying the operator that generated this parameter set (or the name of the parameter set) and the 'operator name' column specifies the name of the target operator i.e. the operator that will use this parameter.

## Related Documents

- **Clone Parameters** (page 666)

## Tutorial Processes

### Building a model using an optimal parameter set

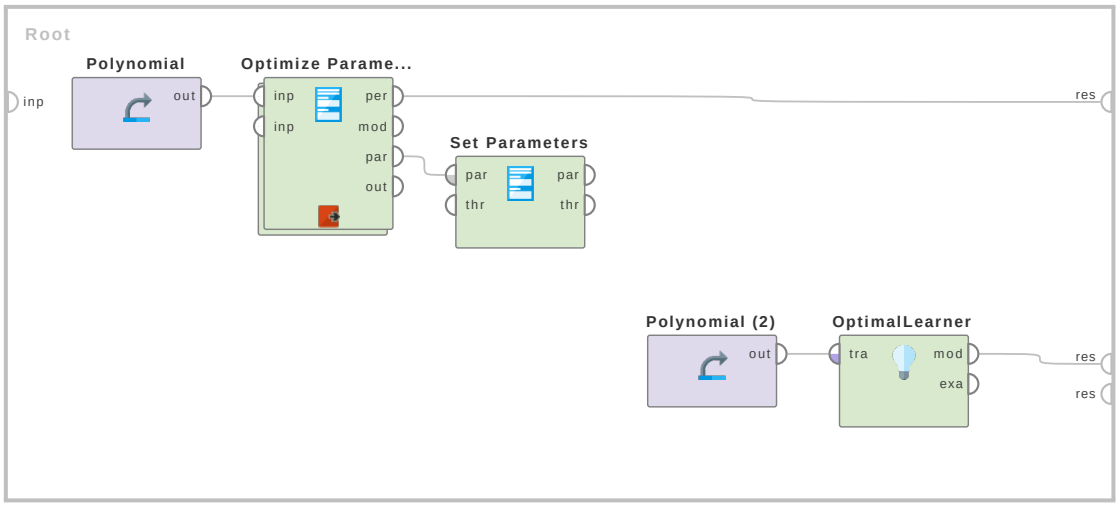


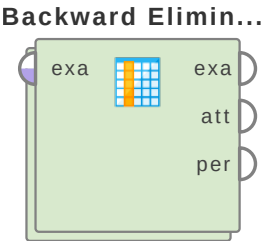

Figure 4.103: Tutorial process 'Building a model using an optimal parameter set'.

The 'Polynomial' data set is loaded using the Retrieve operator. The Optimize Parameters (Grid) operator is applied on it to produce an optimal set of parameters. The X-Validation operator is applied in the subprocess of the Optimize Parameters (Grid) operator. The X-Validation operator uses an SVM (LibSVM) operator for training a model. This SVM (LibSVM) operator is named 'Learner'. After execution of the Optimize Parameters (Grid) operator, it returns a parameter set with optimal values of the C and degree parameters of the SVM (LibSVM) operator. This parameter set is provided to the Set Parameters operator. The Set Parameters operator provides these optimal parameters to the SVM (LibSVM) operator in the main process (outside the

Optimize Parameter (Grid) operator). This SVM (LibSVM) operator is named 'OptimalLearner' and it comes after the Set Parameters operator in the execution sequence. Have a look at the name map parameter of the Set Parameters operator. The name 'Learner', is specified in the 'set operator name' column and the name 'OptimalLearner' is specified in the 'operator name' column of the name map parameter. Each parameter in this list maps the name of an operator that was used during the optimization (in our case this is 'Learner') to an operator that should now use these parameters (in our case this is 'OptimalLearner'). The 'OptimalLearner' uses the optimal parameters of 'Learner' and applies the model on the 'Polynomial' data set. Have a look at the parameters of the 'OptimalLearner' before the execution of the process. The degree and C parameters are set to 1 and 0.0 respectively. These values are changed to 3 and 250 after execution of the process because these are the optimal values for these parameters generated by the Optimize Parameters (Grid) operator and then these parameters were used by 'OptimalLearner'. These parameters can also be seen during the process execution (at the breakpoint after the Optimize Parameters (Grid) operator).

# 4.7.2 Feature Selection

## Backward Elimination




This operator selects the most relevant attributes of the given ExampleSet through an efficient implementation of the backward elimination scheme.

### Description

The Backward Elimination operator is a nested operator i.e. it has a subprocess. The subprocess of the Backward Elimination operator must always return a performance vector. For more information regarding subprocesses please study the Subprocess operator.

The Backward Elimination operator starts with the full set of attributes and, in each round, it removes each remaining attribute of the given ExampleSet. For each removed attribute, the performance is estimated using the inner operators, e.g. a cross-validation. Only the attribute giving the least decrease of performance is finally removed from the selection. Then a new round is started with the modified selection. This implementation avoids any additional memory consumption besides the memory used originally for storing the data and the memory which might be needed for applying the inner operators. The *stopping behavior* parameter specifies when the iteration should be aborted. There are three different options:

- with decrease: The iteration runs as long as there is any increase in performance.
- with decrease of more than: The iteration runs as long as the decrease is less than the specified threshold, either relative or absolute. The *maximal relative decrease* parameter is used for specifying the maximal relative decrease if the *use relative decrease* parameter is set to true. Otherwise, the *maximal absolute decrease* parameter is used for specifying the maximal absolute decrease.
- with significant decrease: The iteration stops as soon as the decrease is significant to the level specified by the *alpha* parameter.

The *speculative rounds* parameter defines how many rounds will be performed in a row, after the first time the stopping criterion is fulfilled. If the performance increases again during the speculative rounds, the elimination will be continued. Otherwise all additionally eliminated attributes will be restored, as if no speculative rounds had executed. This might help avoiding getting stuck in local optima.

Feature selection i.e. the question for the most relevant features for classification or regression problems, is one of the main data mining tasks. A wide range of search methods have been integrated into RapidMiner including evolutionary algorithms. For all search methods we need a performance measurement which indicates how well a search point (a feature subset) will probably perform on the given data set.

### Differentiation

- **Forward Selection** The Forward Selection operator starts with an empty selection of attributes and, in each round, it adds each unused attribute of the given ExampleSet. For

each added attribute, the performance is estimated using the inner operators, e.g. a cross-validation. Only the attribute giving the highest increase of performance is added to the selection. Then a new round is started with the modified selection. See page 685 for details.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. This ExampleSet is available at the first port of the nested chain (inside the subprocess) for processing in the subprocess.

## Output Ports

**example set *(exa)*** The feature selection algorithm is applied on the input ExampleSet. The resultant ExampleSet with reduced attributes is delivered through this port.

**attribute weights *(att)*** The attribute weights are delivered through this port.

**performance *(per)*** This port delivers the Performance Vector for the selected attributes. A Performance Vector is a list of performance criteria values.

## Parameters

**maximal number of eliminations *(integer)*** This parameter specifies the maximal number of backward eliminations.

**speculative rounds *(integer)*** This parameter specifies the number of times, the stopping criterion might be consecutively ignored before the elimination is actually stopped. A number higher than one might help avoiding getting stuck in local optima.

**stopping behavior *(selection)*** The *stopping behavior* parameter specifies when the iteration should be aborted. There are three different options:

- **with_decrease** The iteration runs as long as there is any increase in performance.
- **with_decrease_of_more_than** The iteration runs as long as the decrease is less than the specified threshold, either relative or absolute. The *maximal relative decrease* parameter is used for specifying the maximal relative decrease if the *use relative decrease* parameter is set to true. Otherwise, the *maximal absolute decrease* parameter is used for specifying the maximal absolute decrease.
- **with_significant_decrease** The iteration stops as soon as the decrease is significant to the level specified by the *alpha* parameter.

**use relative decrease *(boolean)*** This parameter is only available when the *stopping behavior* parameter is set to 'with decrease of more than'. If the *use relative decrease* parameter is set to true the *maximal relative decrease* parameter will be used otherwise the *maximal absolute decrease* parameter.

**maximal absolute decrease *(real)*** This parameter is only available when the *stopping behavior* parameter is set to 'with decrease of more than' and the *use relative decrease* parameter is set to false. If the absolute performance decrease to the last step exceeds this threshold, the elimination will be stopped.

**maximal relative decrease *(real)*** This parameter is only available when the *stopping behavior* parameter is set to 'with decrease of more than' and the *use relative decrease* parameter is set to true. If the relative performance decrease to the last step exceeds this threshold, the elimination will be stopped.

**alpha *(real)*** This parameter is only available when the *stopping behavior* parameter is set to 'with significant decrease'. This parameter specifies the probability threshold which determines if differences are considered as significant.

## Related Documents

- **Forward Selection** (page 685)

## Tutorial Processes

### Feature reduction of the Polynomial data set

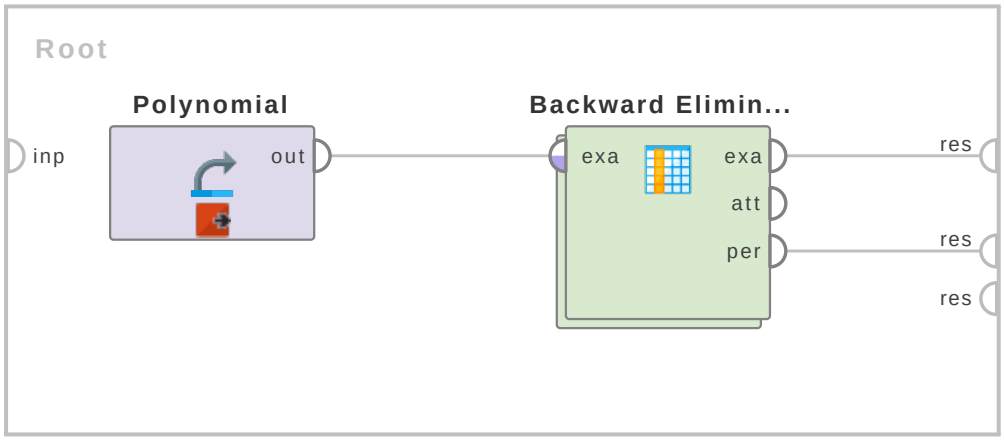
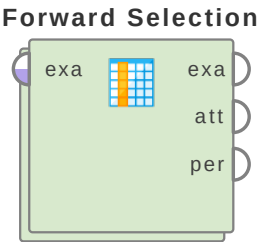



Figure 4.104: Tutorial process 'Feature reduction of the Polynomial data set'.

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes other then the label attribute. The Backward Elimination operator is applied on the ExampleSet which is a nested operator i.e. it has a subprocess. It is necessary for the subprocess to deliver a performance vector. This performance vector is used by the underlying feature reduction algorithm. Have a look at the subprocess of this operator. The X-Validation operator is used there which itself is a nested operator. Have a look at the subprocesses of the X-Validation operator. The K-NN operator is used in the 'Training' subprocess to train a model. The trained model is applied using the Apply Model operator in the 'Testing' subprocess. The performance is measured through the Performance operator and the resultant performance vector is used by the underlying algorithm. Run the process and switch to the Results Workspace. You can see that the ExampleSet that had 5 attributes has now been reduced to 3 attributes.

# Forward Selection




This operator selects the most relevant attributes of the given ExampleSet through a highly efficient implementation of the forward selection scheme.

## Description

The Forward Selection operator is a nested operator i.e. it has a subprocess. The subprocess of the Forward Selection operator must always return a performance vector. For more information regarding subprocesses please study the Subprocess operator.

The Forward Selection operator starts with an empty selection of attributes and, in each round, it adds each unused attribute of the given ExampleSet. For each added attribute, the performance is estimated using the inner operators, e.g. a cross-validation. Only the attribute giving the highest increase of performance is added to the selection. Then a new round is started with the modified selection. This implementation avoids any additional memory consumption besides the memory used originally for storing the data and the memory which might be needed for applying the inner operators. The *stopping behavior* parameter specifies when the iteration should be aborted. There are three different options:

- without increase : The iteration runs as long as there is any increase in performance.
- without increase of at least: The iteration runs as long as the increase is at least as high as specified, either relative or absolute. The *minimal relative increase* parameter is used for specifying the minimal relative increase if the *use relative increase* parameter is set to true. Otherwise, the *minimal absolute increase* parameter is used for specifying the minimal absolute increase.
- without significant increase: The iteration stops as soon as the increase is not significant to the level specified by the *alpha* parameter.

The *speculative rounds* parameter defines how many rounds will be performed in a row, after the first time the stopping criterion is fulfilled. If the performance increases again during the speculative rounds, the selection will be continued. Otherwise all additionally selected attributes will be removed, as if no speculative rounds had executed. This might help avoiding getting stuck in local optima.

Feature selection i.e. the question for the most relevant features for classification or regression problems, is one of the main data mining tasks. A wide range of search methods have been integrated into RapidMiner including evolutionary algorithms. For all search methods we need a performance measurement which indicates how well a search point (a feature subset) will probably perform on the given data set.

## Differentiation

- **Backward Elimination** The Backward Elimination operator starts with the full set of attributes and, in each round, it removes each remaining attribute of the given ExampleSet. For each removed attribute, the performance is estimated using the inner operators, e.g.

a cross-validation. Only the attribute giving the least decrease of performance is finally removed from the selection. Then a new round is started with the modified selection. See page 682 for details.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. This ExampleSet is available at the first port of the nested chain (inside the subprocess) for processing in the subprocess.

## Output Ports

**example set *(exa)*** The feature selection algorithm is applied on the input ExampleSet. The resultant ExampleSet with reduced attributes is delivered through this port.

**attribute weights *(att)*** The attribute weights are delivered through this port.

**performance *(per)*** This port delivers the Performance Vector for the selected attributes. A Performance Vector is a list of performance criteria values.

## Parameters

**maximal number of attributes *(integer)*** This parameter specifies the maximal number of attributes to be selected through Forward Selections.

**speculative rounds *(integer)*** This parameter specifies the number of times, the stopping criterion might be consecutively ignored before the elimination is actually stopped. A number higher than one might help avoiding getting stuck in local optima.

**stopping behavior *(selection)*** The *stopping behavior* parameter specifies when the iteration should be aborted. There are three different options:

- **without_increase** The iteration runs as long as there is any increase in performance.
- **without_increase_of_at_least** The iteration runs as long as the increase is at least as high as specified, either relative or absolute. The *minimal relative increase* parameter is used for specifying the minimal relative increase if the *use relative increase* parameter is set to true. Otherwise, the *minimal absolute increase* parameter is used for specifying the minimal absolute increase.
- **without_significant_increase** The iteration stops as soon as the increase is not significant to the level specified by the *alpha* parameter.

**use relative increase *(boolean)*** This parameter is only available when the *stopping behavior* parameter is set to 'without increase of at least'. If the *use relative increase* parameter is set to true the *minimal relative increase* parameter will be used otherwise the *minimal absolute increase* parameter will be used.

**minimal absolute increase *(real)*** This parameter is only available when the *stopping behavior* parameter is set to 'without increase of at least' and the *use relative increase* parameter is set to false. If the absolute performance increase to the last step drops below this threshold, the selection will be stopped.

**minimal relative increase *(real)*** This parameter is only available when the *stopping behavior* parameter is set to 'without increase of at least' and the *use relative increase* parameter is set to true. If the relative performance increase to the last step drops below this threshold, the selection will be stopped.

**alpha *(real)*** This parameter is only available when the *stopping behavior* parameter is set to 'without significant increase'. This parameter specifies the probability threshold which determines if differences are considered as significant.

## Related Documents

- **Backward Elimination** (page 682)

## Tutorial Processes

### Feature reduction of the Polynomial data set through Forward Selection

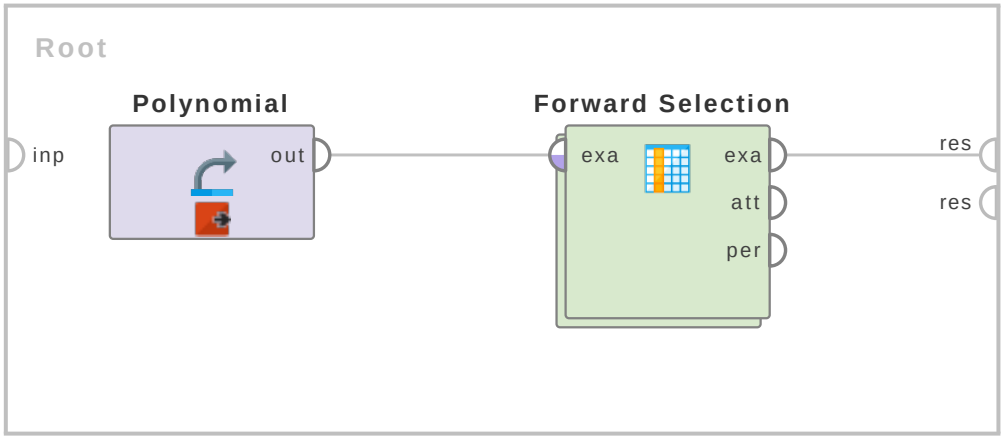
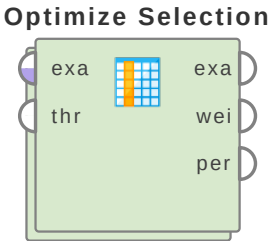



Figure 4.105: Tutorial process 'Feature reduction of the Polynomial data set through Forward Selection'.

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes other then the label attribute. The Forward Selection operator is applied on the ExampleSet which is a nested operator i.e. it has a subprocess. It is necessary for the subprocess to deliver a performance vector. This performance vector is used by the underlying feature reduction algorithm. Have a look at the subprocess of this operator. The X-Validation operator is used which itself is a nested operator. Have a look at the subprocesses of the X-Validation operator. The K-NN operator is used in the 'Training' subprocess to train a model. The trained model is applied using the Apply Model operator in the 'Testing' subprocess. The performance is measured through the Performance operator and the resultant performance vector is used by the underlying algorithm. Run the process and switch to the Results Workspace. You can see that the ExampleSet that had 5 attributes has now been reduced to 3 attributes.

# Optimize Selection




This operator selects the most relevant attributes of the given ExampleSet. Two deterministic greedy feature selection algorithms 'forward selection' and 'backward elimination' are used for feature selection.

## Description

Feature selection i.e. the question for the most relevant features for classification or regression problems, is one of the main data mining tasks. A wide range of search methods have been integrated into RapidMiner including evolutionary algorithms. For all search methods we need a performance measurement which indicates how well a search point (a feature subset) will probably perform on the given data set.

A deterministic algorithm is an algorithm which, in informal terms, behaves predictably. Given a particular input, it will always produce the same output, and the underlying machine will always pass through the same sequence of states.

A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. On some problems, a greedy strategy may not produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution.

This operator realizes the two deterministic greedy feature selection algorithms 'forward selection' and 'backward elimination'. However, we have added some enhancements to the standard algorithms which are described below:

### Forward Selection

1. Create an initial population with $n$ individuals where $n$ is the number of attributes in the input ExampleSet. Each individual will use exactly one of the features.
2. Evaluate the attribute sets and select only the best $k$.
3. For each of the $k$ attribute sets do: If there are $j$ unused attributes, make $j$ copies of the attribute set and add exactly one of the previously unused attributes to the attribute set.
4. As long as the performance improved in the last $p$ iterations go to step 2

### Backward Elimination

1. Start with an attribute set which uses all features.
2. Evaluate all attribute sets and select the best $k$.
3. For each of the $k$ attribute sets do: If there are $j$ attributes used, make $j$ copies of the attribute set and remove exactly one of the previously used attributes from the attribute set.
4. As long as the performance improved in the last $p$ iterations go to step 2

The parameter *k* can be specified by the *keep best* parameter, the parameter *p* can be specified by the *generations without improval* parameter. These parameters have default values 1 which means that the standard selection algorithms are used. Using other values increases the runtime but might help to avoid local extrema in the search for the global optimum.

Another unusual parameter is the *maximum number of generations* parameter. This parameter bounds the number of iterations to this maximum of feature selections / de-selections. In combination with the *generations without improval* parameter, this allows several different selection schemes (which are described for forward selection, backward elimination works analogous):

#### maximum number of generations = m and generations without improval = p:

Selects maximal *m* features. The selection stops if no performance improvement was measured in the last *p* generations.

#### maximum number of generations = -1 and generations without improval = p:

Tries to selects new features until no performance improvement was measured in the last *p* generations.

#### maximum number of generations = m and generations without improval = -1:

Selects maximal *m* features. The selection stops is not stopped until all combinations with maximal *m* were tried. However, the result might contain fewer features than these.

#### maximum number of generations = -1 and generations without improval = -1:

Test all combinations of attributes (brute force, this might take a very long time and should only be applied to small attribute sets).

## Differentiation

- **Optimize Selection (Evolutionary)** This is also an attribute set reduction operator but it uses a genetic algorithm for this purpose. See page 695 for details.

## Input Ports

**example set in *(exa)*** This input port expects an ExampleSet. This ExampleSet is available at the first port of the nested chain (inside the subprocess) for processing in the subprocess.

**through *(thr)*** This operator can have multiple *through* ports. When one input is connected with the *through* port, another *through* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *through* port of this operator is available at the first *through* port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at the subprocess level.

## Output Ports

**example set out *(exa)*** The feature selection algorithm is applied on the input ExampleSet. The resultant ExampleSet with reduced attributes is delivered through this port.

**weights *(wei)*** The attribute weights are delivered through this port.

**performance *(per)*** This port delivers the Performance Vector for the selected attributes. A Performance Vector is a list of performance criteria values.

## Parameters

**selection direction *(selection)*** This parameter specifies which of the 'forward selection' and 'backward elimination' algorithms should be used.

**limit generations without improval *(boolean)*** This parameter indicates if the optimization should be aborted if this number of generations showed no improvement. If unchecked, always the maximal number of generations will be used.

**generations without improval *(integer)*** This parameter is only available when the *limit generations without improval* parameter is set to true. This parameter specifies the stop criterion for early stopping i.e. it stops after *n* generations without improvement in the performance. *n* is specified by this parameter.

**limit number of generations *(boolean)*** This parameter indicates if the number of generations should be limited to a specific number.

**keep best *(integer)*** The best *n* individuals are kept in each generation where *n* is the value of this parameter.

**maximum number of generations *(integer)*** This parameter is only available when the *limit number of generations* parameter is set to true. This parameter specifies the number of generations after which the algorithm should be terminated.

**normalize weights *(boolean)*** This parameter indicates if the final weights should be normalized. If set to true, the final weights are normalized such that the maximum weight is 1 and the minimum weight is 0.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed* and is only available if the *use local random seed* parameter is set to true.

**show stop dialog *(boolean)*** This parameter determines if a dialog with a *stop* button should be displayed which stops the search for the best feature space. If the search for best feature space is stopped, the best individual found till then will be returned.

**user result individual selection *(boolean)*** If this parameter is set to true, it allows the user to select the final result individual from the last population.

**show population plotter *(boolean)*** This parameter determines if the current population should be displayed in the performance space.

**plot generations *(integer)*** This parameter is only available when the *show population plotter* parameter is set to true. The population plotter is updated in these generations.

**constraint draw range *(boolean)*** This parameter is only available when the *show population plotter* parameter is set to true. This parameter determines if the draw range of the population plotter should be constrained between 0 and 1.

**draw dominated points *(boolean)*** This parameter is only available when the *show population plotter* parameter is set to true. This parameter determines if only points which are not Pareto dominated should be drawn on the population plotter.

**population criteria data file *(filename)*** This parameter specifies the path to the file in which the criteria data of the final population should be saved.

**maximal fitness *(real)*** This parameter specifies the maximal fitness. The optimization will stop if the fitness reaches this value.

## Related Documents

- **Optimize Selection (Evolutionary)** (page 695)

## Tutorial Processes

### Feature reduction of the Polynomial data set

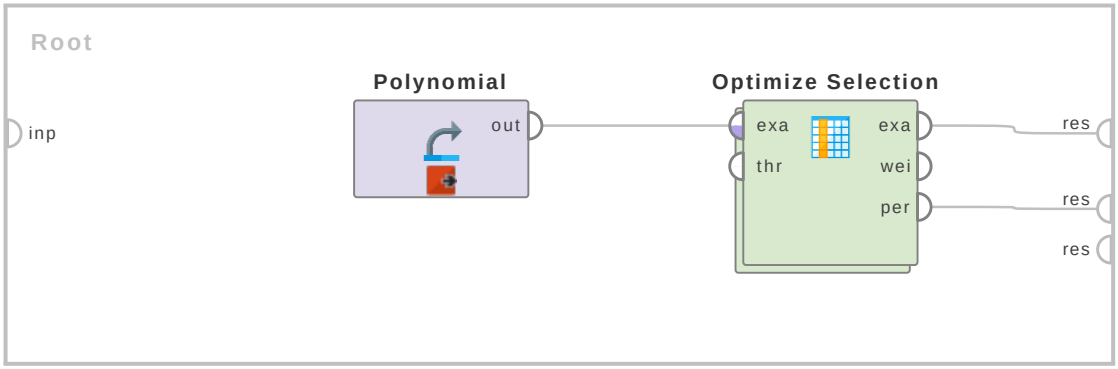
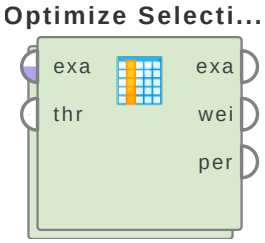



Figure 4.106: Tutorial process 'Feature reduction of the Polynomial data set'.

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes other then the label attribute. The Optimize Selection operator is applied on the ExampleSet which is a nested operator i.e. it has a subprocess. It is necessary for the subprocess to deliver a performance vector. This performance vector is used by the underlying feature reduction algorithm. Have a look at the subprocess of this operator. The Split Validation operator has been used there which itself is a nested operator. Have a look at the subprocesses of the Split Validation operator. The SVM operator is used in the 'Training' subprocess to train a model. The trained model is applied using the Apply Model operator in the 'Testing' subprocess. The performance is measured through the Performance operator and the resultant performance vector is used by the underlying algorithm. Run the process and switch to the Results Workspace. You can see that the ExampleSet that had 5 attributes has now been reduced to 2 attributes.

# Optimize Selection (Brute Force)




This operator selects the most relevant attributes of the given ExampleSet by trying all possible combinations of attribute selections.

## Description

The Optimize Selection (Brute Force) operator is a nested operator i.e. it has a subprocess. This subprocess must always return a performance vector. This operator selects the feature set with the best performance vector. You need to have basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

Feature selection i.e. the question for the most relevant features for classification or regression problems, is one of the main data mining tasks. A wide range of search methods have been integrated into RapidMiner including evolutionary algorithms. For all search methods we need a performance measurement which indicates how well a search point (a feature subset) will probably perform on the given data set.

This feature selection operator selects the best attribute set by trying all possible combinations of attribute selections. It returns the ExampleSet containing the subset of attributes which produced the best performance. As this operator works on the power-set of the attribute set, it has exponential runtime.

## Differentiation

- **Optimize Selection (Evolutionary)** This is also an attribute set reduction operator but it uses a genetic algorithm for this purpose. See page 695 for details.

## Input Ports

**example set in *(exa)*** This input port expects an ExampleSet. This ExampleSet is available at the first port of the nested chain (inside the subprocess) for processing in the subprocess.

**through *(thr)*** This operator can have multiple *through* ports. When one input is connected with the *through* port, another *through* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *through* port of this operator is available at the first *through* port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at the subprocess level.

## Output Ports

**example set out *(exa)*** The feature selection algorithm is applied on the input ExampleSet. The resultant ExampleSet with reduced attributes is delivered through this port.

**weights *(wei)*** The attribute weights are delivered through this port.

**performance *(per)*** This port delivers the Performance Vector for the selected attributes. A Performance Vector is a list of performance criteria values.

## Parameters

**use exact number of attributes *(boolean)*** This parameter determines if only combinations containing exact numbers of attributes should be tested. The exact number is specified by the *exact number of attributes* parameter.

**exact number of attributes *(integer)*** This parameter is only available when the *use exact number of attributes* parameter is set to true. Only combinations containing this numbers of attributes would be generated and tested.

**restrict maximum *(boolean)*** If set to true, the maximum number of attributes whose combinations will be generated and tested can be restricted. Otherwise all combinations of all attributes are generated and tested. This parameter is only available when the *use exact number of attributes* parameter is set to true.

**min number of attributes *(integer)*** This parameter determines the minimum number of features used for the combinations to be generated and tested.

**max number of attributes *(integer)*** This parameter determines the maximum number of features used for the combinations to be generated and tested. This parameter is only available when the *restrict maximum* parameter is set to true.

**normalize weights *(boolean)*** This parameter indicates if the final weights should be normalized. If set to true, the final weights are normalized such that the maximum weight is 1 and the minimum weight is 0.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed* and is only available if the *use local random seed* parameter is set to true.

**show stop dialog *(boolean)*** This parameter determines if a dialog with a *stop* button should be displayed which stops the search for the best feature space. If the search for best feature space is stopped, the best individual found till then will be returned.

**user result individual selection *(boolean)*** If this parameter is set to true, it allows the user to select the final result individual from the last population.

**show population plotter *(boolean)*** This parameter determines if the current population should be displayed in the performance space.

**plot generations *(integer)*** This parameter is only available when the *show population plotter* parameter is set to true. The population plotter is updated in these generations.

**constraint draw range *(boolean)*** This parameter is only available when the *show population plotter* parameter is set to true. This parameter determines if the draw range of the population plotter should be constrained between 0 and 1.

**draw dominated points *(boolean)*** This parameter is only available when the *show population plotter* parameter is set to true. This parameter determines if only points which are not Pareto dominated should be drawn on the population plotter.

**population criteria data file *(filename)*** This parameter specifies the path to the file in which the criteria data of the final population should be saved.

**maximal fitness *(real)*** This parameter specifies the maximal fitness. The optimization will stop if the fitness reaches this value.

## Related Documents

- **Optimize Selection (Evolutionary)** (page 695)

## Tutorial Processes

### Feature reduction of the Polynomial data set

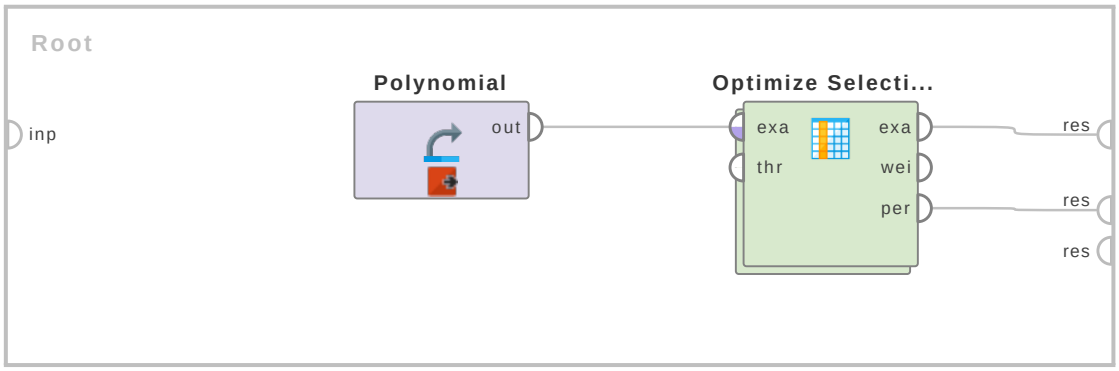
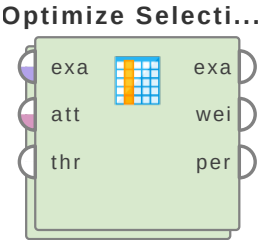



Figure 4.107: Tutorial process 'Feature reduction of the Polynomial data set'.

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes other then the label attribute. The Optimize Selection (Brute Force) operator is applied on the ExampleSet which is a nested operator i.e. it has a subprocess. It is necessary for the subprocess to deliver a performance vector. This performance vector is used by the underlying feature reduction algorithm. Have a look at the subprocess of this operator. The Split Validation operator has been used there which itself is a nested operator. Have a look at the subprocesses of the Split Validation operator. The SVM operator is used in the 'Training' subprocess to train a model. The trained model is applied using the Apply Model operator in the 'Testing' subprocess. The performance is measured through the Performance operator and the resultant performance vector is used by the underlying algorithm. Run the process and switch to the Results Workspace. You can see that the ExampleSet that had 5 attributes has now been reduced to 3 attributes.

# Optimize Selection (Evolutionary)




This operator selects the most relevant attributes of the given ExampleSet. A Genetic Algorithm is used for feature selection.

## Description

Feature selection i.e. the question for the most relevant features for classification or regression problems, is one of the main data mining tasks. A wide range of search methods have been integrated into RapidMiner including evolutionary algorithms. For all search methods we need a performance measurement which indicates how well a search point (a feature subset) will probably perform on the given data set.

A genetic algorithm (GA) is a search heuristic that mimics the process of natural evolution. This heuristic is routinely used to generate useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover.

In genetic algorithm for feature selection 'mutation' means switching features on and off and 'crossover' means interchanging used features. Selection is done by the specified selection scheme which is selected by the *selection scheme* parameter. A genetic algorithm works as follows:

Generate an initial population consisting of *p* individuals. Each attribute is switched on with probability *p_i*. The numbers *p* and *p_i* can be adjusted by the *population size* and *p initialize* parameters respectively.

For all individuals in the population

1. Perform mutation, i.e. set used attributes to unused with probability *p_m* and vice versa. The probability *p_m* can be adjusted by the *p mutation* parameter.

2. Choose two individuals from the population and perform crossover with probability *p_c*. The probability *p_c* can be adjusted by the *p crossover* parameter. The type of crossover can be selected by the *crossover type* parameter.

3. Perform selection, map all individuals according to their fitness and draw *p* individuals at random according to their probability where *p* is the population size which can be adjusted by the *population size* parameter.

4. As long as the fitness improves, go to step number 2.

If the ExampleSet contains value series attributes with block numbers, the whole block will be switched on and off. Exact, minimum or maximum number of attributes in combinations to be tested can be specified by the appropriate parameters. Many other options are also available for this operator. Please study the parameters section for more information.

## Input Ports

**example set in *(exa)*** This input port expects an ExampleSet. This ExampleSet is available at the first port of the nested chain (inside the subprocess) for processing in the subprocess.

**attribute weights in *(att)*** This port expects attribute weights. It is not compulsory to use this port.

**through *(thr)*** This operator can have multiple *through* ports. When one input is connected with the *through* port, another *through* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *through* port of this operator is available at the first *through* port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected right number of ports at subprocess level.

## Output Ports

**example set out *(exa)*** The genetic algorithm is applied on the input ExampleSet. The resultant ExampleSet with reduced attributes is delivered through this port.

**weights *(wei)*** The attribute weights are delivered through this port.

**performance *(per)*** This port delivers the Performance Vector for the selected attributes. A Performance Vector is a list of performance criteria values.

## Parameters

**use exact number of attributes *(boolean)*** This parameter determines if only combinations containing exact numbers of attributes should be tested. The exact number is specified by the *exact number of attributes* parameter.

**exact number of attributes *(integer)*** This parameter is only available when the *use exact number of attributes* parameter is set to true. Only combinations containing this numbers of attributes would be generated and tested.

**restrict maximum *(boolean)*** If set to true, the maximum number of attributes whose combinations will be generated and tested can be restricted. Otherwise all combinations of all attributes are generated and tested. This parameter is only available when the *use exact number of attributes* parameter is set to true.

**min of attributes *(integer)*** This parameter determines the minimum number of features used for the combinations to be generated and tested.

**max number of attributes *(integer)*** This parameter determines the maximum number of features used for the combinations to be generated and tested. This parameter is only available when the *restrict maximum* parameter is set to true.

**population size *(integer)*** This parameter specifies the population size i.e. the number of individuals per generation.

**maximum number of generations *(integer)*** This parameter specifies the number of generations after which the algorithm should be terminated.

**use early stopping *(boolean)*** This parameter enables early stopping. If not set to true, always the maximum number of generations are performed.

**generations without improval *(integer)*** This parameter is only available when the *use early stopping* parameter is set to true. This parameter specifies the stop criterion for early stopping i.e. it stops after *n* generations without improvement in the performance. *n* is specified by this parameter.

**normalize weights *(boolean)*** This parameter indicates if the final weights should be normalized. If set to true, the final weights are normalized such that the maximum weight is 1 and the minimum weight is 0.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

**show stop dialog *(boolean)*** This parameter determines if a dialog with a *stop* button should be displayed which stops the search for the best feature space. If the search for best feature space is stopped, the best individual found till then will be returned.

**user result individual selection *(boolean)*** If this parameter is set to true, it allows the user to select the final result individual from the last population.

**show population plotter *(boolean)*** This parameter determines if the current population should be displayed in performance space.

**plot generations *(integer)*** This parameter is only available when the *show population plotter* parameter is set to true. The population plotter is updated in these generations.

**constraint draw range *(boolean)*** This parameter is only available when the *show population plotter* parameter is set to true. This parameter determines if the draw range of the population plotter should be constrained between 0 and 1.

**draw dominated points *(boolean)*** This parameter is only available when the *show population plotter* parameter is set to true. This parameter determines if only points which are not Pareto dominated should be drawn on the population plotter.

**population criteria data file *(filename)*** This parameter specifies the path to the file in which the criteria data of the final population should be saved.

**maximal fitness *(real)*** This parameter specifies the maximal fitness. The optimization will stop if the fitness reaches this value.

**selection scheme *(selection)*** This parameter specifies the selection scheme of this evolutionary algorithms.

**tournament size *(real)*** This parameter is only available when the *selection scheme* parameter is set to 'tournament'. It specifies the fraction of the current population which should be used as tournament members.

**start temperature *(real)*** This parameter is only available when the *selection scheme* parameter is set to 'Boltzmann'. It specifies the scaling temperature.

**dynamic selection pressure *(boolean)*** This parameter is only available when the *selection scheme* parameter is set to 'Boltzmann' or 'tournament'. If set to true the selection pressure is increased to maximum during the complete optimization run.

**keep best individual *(boolean)*** If set to true, the best individual of each generations is guaranteed to be selected for the next generation.

**save intermediate weights *(boolean)*** This parameter determines if the intermediate best results should be saved.

**intermediate weights generations *(integer)*** This parameter is only available when the *save intermediate weights* parameter is set to true. The intermediate best results would be saved every *k* generations where *k* is specified by this parameter.

**intermediate weights file *(filename)*** This parameter specifies the file into which the intermediate weights should be saved.

**p initialize *(real)*** The initial probability for an attribute to be switched on is specified by this parameter.

**p mutation *(real)*** The probability for an attribute to be changed is specified by this parameter. If set to -1, the probability will be set to *1/n* where *n* is the total number of attributes.

**p crossover *(real)*** The probability for an individual to be selected for crossover is specified by this parameter.

**crossover type *(selection)*** The type of the crossover can be selected by this parameter.

## Tutorial Processes

### Feature reduction of the Polynomial data set

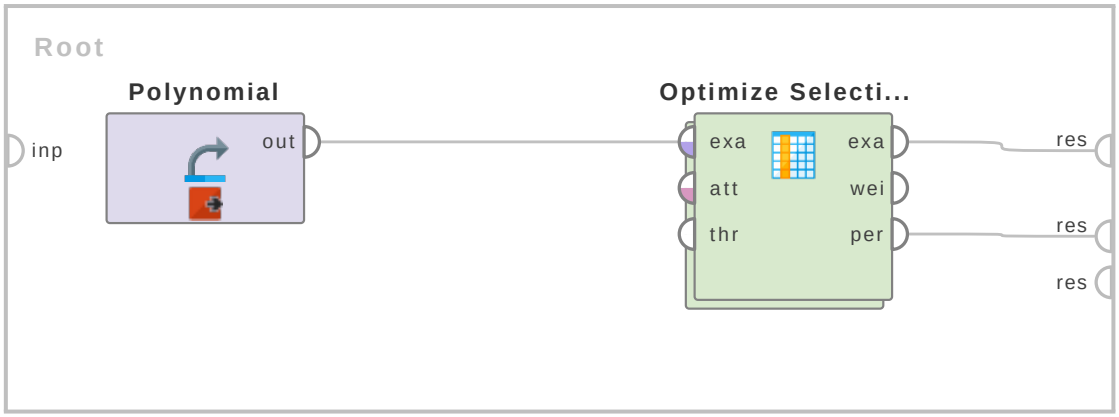


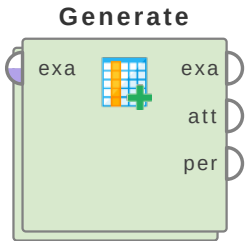

Figure 4.108: Tutorial process 'Feature reduction of the Polynomial data set'.

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes other then the label attribute. The Optimize Selection (Evolutionary) operator is applied on the ExampleSet it is a nested operator i.e. it has a subprocess. It is necessary for the subprocess to deliver a performance vector. This performance vector is used by the underlying Genetic Algorithm. Have a look at the subprocess of this operator. The Split Validation operator has been used there which itself is a nested operator. Have a look at the subprocesses of the Split Validation operator. The SVM operator is used in the 'Training' subprocess to train a model. The

trained model is applied using the Apply Model operator in the 'Testing' subprocess. The performance is measured through the Performance operator and the resultant performance vector is used by the underlying algorithm. Run the process and switch to the Results Workspace. You can see that the ExampleSet that had 5 attributes has now been reduced to 3 attributes.

# 4.7.3 Feature Generation

## Optimize by Generation (GGA)




This operator may select some attributes from the original attribute set and it may also generate new attributes from the original attribute set. GGA (Generating Genetic Algorithm) does not change the original number of attributes unless adding or removing (or both) attributes prove to have a better fitness.

### Description

Sometimes the selection of features alone is not sufficient. In these cases other transformations of the feature space must be performed. The generation of new attributes from the given attributes extends the feature space. Maybe a hypothesis can be easily found in the extended feature space. This operator can be considered to be a blend of attribute selection and attribute generation procedures. It may select some attributes from the original set of attributes and it may also generate new attributes from the original attributes.

A genetic algorithm (GA) is a search heuristic that mimics the process of natural evolution. This heuristic is routinely used to generate useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover. For studying the basic algorithm of a genetic algorithm please study the description of the Optimize Selection (Evolutionary) operator.

In contrast to the simple Genetic Algorithm, the Generating Genetic Algorithm generates new attributes and thus can change the length of an individual. Therefore specialized mutation and crossover operators are being applied. Generators are chosen at random from a list of generators specified by boolean parameters. This operator is a nested operator i.e. it has a subprocess. The subprocess must return a performance vector. You need to have basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

### Input Ports

**example set in *(exa)*** This input port expects an ExampleSet. This ExampleSet is available at the first port of the nested chain (inside the subprocess) for processing in the subprocess.

### Output Ports

**example set out *(exa)*** The genetic algorithm is applied on the input ExampleSet. The resultant ExampleSet is delivered through this port.

**attribute weights out *(att)*** The attribute weights are delivered through this port.

**performance out *(per)*** This port delivers the Performance Vector for the selected attributes. A Performance Vector is a list of performance criteria values.

## Parameters

**max number of new attributes *(integer)*** This parameter specifies the maximum number of attributes to generate for an individual in one generation.

**limit max total number of attributes *(boolean)*** This parameter indicates if the total number of attributes in all generations should be limited. If set to true, the maximum number is specified by the *max total number of attributes* parameter.

**max total number of attributes *(integer)*** This parameter is only available when the *limit max total number of attributes* parameter is set to true. This parameter specifies the maximum total number of attributes in all generations.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

**show stop dialog *(boolean)*** This parameter determines if a dialog with a *stop* button should be displayed which stops the search for the best feature space. If the search for best feature space is stopped, the best individual found till then will be returned.

**maximal fitness *(real)*** This parameter specifies the maximal fitness. The optimization will stop if the fitness reaches this value.

**population size *(integer)*** This parameter specifies the population size i.e. the number of individuals per generation.

**maximum number of generations *(integer)*** This parameter specifies the number of generations after which the algorithm should be terminated.

**use plus *(boolean)*** This parameter indicates if the summation function should be applied for a generation of new attributes.

**use diff *(boolean)*** This parameter indicates if the difference function should be applied for a generation of new attributes.

**use mult *(boolean)*** This parameter indicates if the multiplication function should be applied for a generation of new attributes.

**use div *(boolean)*** This parameter indicates if the division function should be applied for a generation of new attributes.

**reciprocal value *(boolean)*** This parameter indicates if the reciprocal function should be applied for a generation of new attributes.

**use early stopping *(boolean)*** This parameter enables early stopping. If not set to true, always the maximum number of generations are performed.

**generations without improval *(integer)*** This parameter is only available when the *use early stopping* parameter is set to true. This parameter specifies the stop criterion for early stopping i.e. it stops after *n* generations without improvement in the performance. *n* is specified by this parameter.

**tournament size *(real)*** This parameter specifies the fraction of the current population which should be used as tournament members.

**start temperature *(real)*** This parameter specifies the scaling temperature.

**dynamic selection pressure *(boolean)*** If this parameter is set to true, the selection pressure is increased to maximum during the complete optimization run.

**keep best individual *(boolean)*** If set to true, the best individual of each generation is guaranteed to be selected for the next generation.

**p initialize *(real)*** The initial probability for an attribute to be switched on is specified by this parameter.

**p crossover *(real)*** The probability for an individual to be selected for crossover is specified by this parameter.

**crossover type *(selection)*** The type of the crossover can be selected by this parameter.

**p generate *(real)*** This parameter specifies the probability for an individual to be selected for a generation.

**use heuristic mutation probability *(boolean)*** If this parameter is set to true, the probability for mutations will be chosen as $1/n$ where $n$ is the number of attributes. Otherwise the probability for mutations should be specified through the *p mutation* parameter

**p mutation *(real)*** The probability for an attribute to be changed is specified by this parameter. If set to -1, the probability will be set to $1/n$ where $n$ is the total number of attributes.

## Tutorial Processes

### Applying GGA on the Polynomial data set

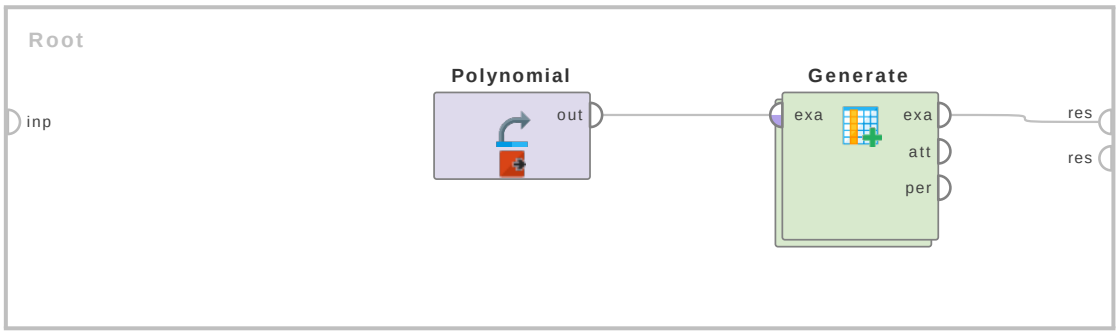
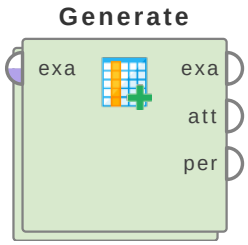



Figure 4.109: Tutorial process 'Applying GGA on the Polynomial data set'.

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes other then the label attribute. The Optimize by Generation (GGA) operator is applied on the ExampleSet. Optimize by Generation (GGA) is a nested operator i.e. it has a subprocess. It is necessary for the subprocess to deliver a performance vector. This performance vector is used by the underlying Genetic Algorithm. Have a look at the subprocess of this operator. The Split

Validation operator is used there which itself is a nested operator. Have a look at the subprocesses of the Split Validation operator. The Linear Regression operator is used in the 'Training' subprocess to train a model. The trained model is applied using the Apply Model operator in the 'Testing' subprocess. The performance is measured through the Performance (Regression) operator and the resultant performance vector is used by the underlying algorithm. Run the process and switch to the Results Workspace. You can see that the ExampleSet that had 5 attributes, now has 4 attributes. The resultant ExampleSet with reduced attributes can be seen in the Results Workspace.

# Optimize by Generation (YAGGA)




This operator may select some attributes from the original attribute set and it may also generate new attributes from the original attribute set. YAGGA (Yet Another Generating Genetic Algorithm) does not change the original number of attributes unless adding or removing (or both) attributes prove to have a better fitness.

## Description

Sometimes the selection of features alone is not sufficient. In these cases other transformations of the feature space must be performed. The generation of new attributes from the given attributes extends the feature space. Maybe a hypothesis can be easily found in the extended feature space. This operator can be considered to be a blend of attribute selection and attribute generation procedures. It may select some attributes from the original set of attributes and it may also generate new attributes from the original attributes. The (generating) mutation can do one of the following things with different probabilities:

- Probability p/4: Add a newly generated attribute to the feature vector.
- Probability p/4: Add a randomly chosen original attribute to the feature vector.
- Probability p/2: Remove a randomly chosen attribute from the feature vector.

Thus it is guaranteed that the length of the feature vector can both grow and shrink. On average it will keep its original length, unless longer or shorter individuals prove to have a better fitness.

A genetic algorithm (GA) is a search heuristic that mimics the process of natural evolution. This heuristic is routinely used to generate useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover. For studying the basic algorithm of a genetic algorithm please study the description of the Optimize Selection (Evolutionary) operator.

This operator is a nested operator i.e. it has a subprocess. The subprocess must return a performance vector. You need to have basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

## Differentiation

- **Optimize by Generation (YAGGA2)** The YAGGA2 operator is an improved version of the usual YAGGA operator, this operator allows more feature generators and provides several techniques for redundancy prevention. This leads to smaller ExampleSets containing less redundant features. See page 708 for details.

## Input Ports

**example set in *(exa)*** This input port expects an ExampleSet. This ExampleSet is available at the first port of the nested chain (inside the subprocess) for processing in the subprocess.

## Output Ports

**example set out *(exa)*** The genetic algorithm is applied on the input ExampleSet. The resultant ExampleSet is delivered through this port.

**attribute weights out *(att)*** The attribute weights are delivered through this port.

**performance out *(per)*** This port delivers the Performance Vector for the selected attributes. A Performance Vector is a list of performance criteria values.

## Parameters

**limit max total number of attributes *(boolean)*** This parameter indicates if the total number of attributes in all generations should be limited. If set to true, the maximum number is specified by the *max total number of attributes* parameter.

**max total number of attributes *(integer)*** This parameter is only available when the *limit max total number of attributes* parameter is set to true. This parameter specifies the maximum total number of attributes in all generations.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is available only if the *use local random seed* parameter is set to true.

**show stop dialog *(boolean)*** This parameter determines if a dialog with a *stop* button should be displayed which stops the search for the best feature space. If the search for best feature space is stopped, the best individual found till then will be returned.

**maximal fitness *(real)*** This parameter specifies the maximal fitness. The optimization will stop if the fitness reaches this value.

**population size *(integer)*** This parameter specifies the population size i.e. the number of individuals per generation.

**maximum number of generations *(integer)*** This parameter specifies the number of generations after which the algorithm should be terminated.

**use plus *(boolean)*** This parameter indicates if the summation function should be applied for generation of new attributes.

**use diff *(boolean)*** This parameter indicates if the difference function should be applied for generation of new attributes.

**use mult *(boolean)*** This parameter indicates if the multiplication function should be applied for generation of new attributes.

**use div *(boolean)*** This parameter indicates if the division function should be applied for generation of new attributes.

**use reciprocals *(boolean)*** This parameter indicates if the reciprocal function should be applied for generation of new attributes.

**use early stopping *(boolean)*** This parameter enables early stopping. If not set to true, always the maximum number of generations are performed.

**generations without improval *(integer)*** This parameter is only available when the *use early stopping* parameter is set to true. This parameter specifies the stop criterion for early stopping i.e. it stops after *n* generations without improvement in the performance. *n* is specified by this parameter.

**tournament size *(real)*** This parameter specifies the fraction of the current population which should be used as tournament members.

**start temperature *(real)*** This parameter specifies the scaling temperature.

**dynamic selection pressure *(boolean)*** If this parameter is set to true, the selection pressure is increased to maximum during the complete optimization run.

**keep best individual *(boolean)*** If set to true, the best individual of each generation is guaranteed to be selected for the next generation.

**p initialize *(real)*** The initial probability for an attribute to be switched on is specified by this parameter.

**p crossover *(real)*** The probability for an individual to be selected for crossover is specified by this parameter.

**crossover type *(selection)*** The type of the crossover can be selected by this parameter.

**use heuristic mutation probability *(boolean)*** If this parameter is set to true, the probability for mutations will be chosen as *1/n* where *n* is the number of attributes. Otherwise the probability for mutations should be specified through the *p mutation* parameter

**p mutation *(real)*** The probability for an attribute to be changed is specified by this parameter. If set to -1, the probability will be set to *1/n* where *n* is the total number of attributes.

## Related Documents

- **Optimize by Generation (YAGGA2)** (page 708)

## Tutorial Processes

### Applying YAGGA on the Polynomial data set

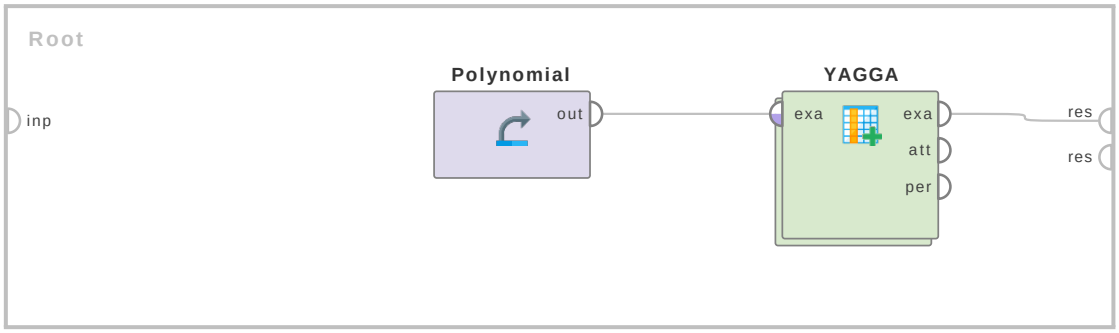
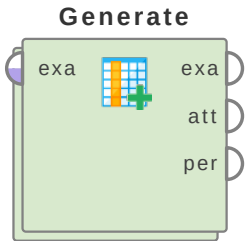



Figure 4.110: Tutorial process 'Applying YAGGA on the Polynomial data set'.

The ‘Polynomial’ data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes other then the label attribute. The Optimize by Generation (YAGGA) operator is applied on the ExampleSet. Optimize by Generation (YAGGA) is a nested operator i.e. it has a subprocess. It is necessary for the subprocess to deliver a performance vector. This performance vector is used by the underlying Genetic Algorithm. Have a look at the subprocess of this operator. The Split Validation operator is used which itself is a nested operator. Have a look at the subprocesses of the Split Validation operator. The Linear Regression operator is used in the ‘Training’ subprocess to train a model. The trained model is applied using the Apply Model operator in the ‘Testing’ subprocess. The performance is measured through the Performance (Regression) operator and the resultant performance vector is used by the underlying algorithm. Run the process and switch to the Results Workspace. You can see that the ExampleSet that had 5 attributes now has 6 attributes. The attributes ‘a1’ and ‘a2’ were selected from the original attribute set and the attributes ‘gensym2’, ‘gensym35’, ‘gensym63’ and ‘gensym72’ were generated. The number of resultant attributes is not less than the number of original attributes because YAGGA is not an attribute reduction operator. It may (or may not) increase or decrease the number of attributes depending on what proves to have a better fitness.

# Optimize by Generation (YAGGA2)




This operator may select some attributes from the original attribute set and it may also generate new attributes from the original attribute set. YAGGA2 (Yet Another Generating Genetic Algorithm 2) does not change the original number of attributes unless adding or removing (or both) attributes proves to have a better fitness. This algorithm is an improved version of YAGGA.

## Description

Sometimes the selection of features alone is not sufficient. In these cases other transformations of the feature space must be performed. The generation of new attributes from the given attributes extends the feature space. Maybe a hypothesis can be easily found in the extended feature space. This operator can be considered to be a blend of attribute selection and attribute generation procedures. It may select some attributes from the original set of attributes and it may also generate new attributes from the original attributes. The (generating) mutation can do one of the following things with different probabilities:

- Probability p/4: Add a newly generated attribute to the feature vector.
- Probability p/4: Add a randomly chosen original attribute to the feature vector.
- Probability p/2: Remove a randomly chosen attribute from the feature vector.

Thus it is guaranteed that the length of the feature vector can both grow and shrink. On average it will keep its original length, unless longer or shorter individuals prove to have a better fitness.

In addition to the usual YAGGA operator, this operator allows more feature generators and provides several techniques for redundancy prevention. This leads to smaller ExampleSets containing less redundant features.

A genetic algorithm (GA) is a search heuristic that mimics the process of natural evolution. This heuristic is routinely used to generate useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover. For studying the basic algorithm of a genetic algorithm please study the description of the Optimize Selection (Evolutionary) operator.

This operator is a nested operator i.e. it has a subprocess. The subprocess must return a performance vector. You need to have basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

## Differentiation

- **Optimize by Generation (YAGGA)** The YAGGA2 operator is an improved version of the usual YAGGA operator, this operator allows more feature generators and provides several techniques for redundancy prevention. This leads to smaller ExampleSets containing less redundant features. See page 704 for details.

## Input Ports

**example set in *(exa)*** This input port expects an ExampleSet. This ExampleSet is available at the first port of the nested chain (inside the subprocess) for processing in the subprocess.

## Output Ports

**example set out *(exa)*** The genetic algorithm is applied on the input ExampleSet. The resultant ExampleSet is delivered through this port.

**attribute weights out *(att)*** The attribute weights are delivered through this port.

**performance out *(per)*** This port delivers the Performance Vector for the selected attributes. A Performance Vector is a list of performance criteria values.

## Parameters

**limit max total number of attributes *(boolean)*** This parameter indicates if the total number of attributes in all generations should be limited. If set to true, the maximum number is specified by the *max total number of attributes* parameter.

**max total number of attributes *(integer)*** This parameter is only available when the *limit max total number of attributes* parameter is set to true. This parameter specifies the maximum total number of attributes in all generations.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

**show stop dialog *(boolean)*** This parameter determines if a dialog with a *stop* button should be displayed which stops the search for the best feature space. If the search for the best feature space is stopped, the best individual found till then will be returned.

**maximal fitness *(real)*** This parameter specifies the maximal fitness. The optimization will stop if the fitness reaches this value.

**population size *(integer)*** This parameter specifies the population size i.e. the number of individuals per generation.

**maximum number of generations *(integer)*** This parameter specifies the number of generations after which the algorithm should be terminated.

**use plus *(boolean)*** This parameter indicates if the summation function should be applied for a generation of new attributes.

**use diff *(boolean)*** This parameter indicates if the difference function should be applied for a generation of new attributes.

**use mult *(boolean)*** This parameter indicates if the multiplication function should be applied for a generation of new attributes.

**use div *(boolean)*** This parameter indicates if the division function should be applied for a generation of new attributes.

**reciprocal value** ***(boolean)*** This parameter indicates if the reciprocal function should be applied for a generation of new attributes.

**use early stopping** ***(boolean)*** This parameter enables early stopping. If not set to true, always the maximum number of generations are performed.

**generations without improval** ***(integer)*** This parameter is only available when the *use early stopping* parameter is set to true. This parameter specifies the stop criterion for early stopping i.e. it stops after *n* generations without improvement in the performance. *n* is specified by this parameter.

**tournament size** ***(real)*** This parameter specifies the fraction of the current population which should be used as tournament members.

**start temperature** ***(real)*** This parameter specifies the scaling temperature.

**dynamic selection pressure** ***(boolean)*** If this parameter is set to true, the selection pressure is increased to maximum during the complete optimization run.

**keep best individual** ***(boolean)*** If set to true, the best individual of each generation is guaranteed to be selected for the next generation.

**p initialize** ***(real)*** The initial probability for an attribute to be switched on is specified by this parameter.

**p crossover** ***(real)*** The probability for an individual to be selected for crossover is specified by this parameter.

**crossover type** ***(selection)*** The type of the crossover can be selected by this parameter.

**use heuristic mutation probability** ***(boolean)*** If this parameter is set to true, the probability for mutations will be chosen as *1/n* where *n* is the number of attributes. Otherwise the probability for mutations should be specified through the *p mutation* parameter

**p mutation** ***(real)*** The probability for an attribute to be changed is specified by this parameter. If set to -1, the probability will be set to *1/n* where *n* is the total number of attributes.

**use square roots** ***(boolean)*** This parameter indicates if the square root function should be applied for a generation of new attributes.

**use power functions** ***(boolean)*** This parameter indicates if the power (of one attribute to another attribute) function should be applied for a generation of new attributes.

**use sin** ***(boolean)*** This parameter indicates if the sine function should be applied for a generation of new attributes.

**use cos** ***(boolean)*** This parameter indicates if the cosine function should be applied for a generation of new attributes.

**use tan** ***(boolean)*** This parameter indicates if the tangent function should be applied for a generation of new attributes.

**use atan** ***(boolean)*** This parameter indicates if the arc tangent function should be applied for a generation of new attributes.

**use exp** ***(boolean)*** This parameter indicates if the exponential function should be applied for a generation of new attributes.

**use log** ***(boolean)*** This parameter indicates if the logarithmic function should be applied for a generation of new attributes.

**use absolute values** ***(boolean)*** This parameter indicates if the absolute function should be applied for a generation of new attributes.

**use min** ***(boolean)*** This parameter indicates if the minimum function should be applied for a generation of new attributes.

**use max** ***(boolean)*** This parameter indicates if the maximum function should be applied for a generation of new attributes.

**use sgn** ***(boolean)*** This parameter indicates if the signum function should be applied for a generation of new attributes.

**use floor ceil functions** ***(boolean)*** This parameter indicates if the floor and ceiling functions should be applied for a generation of new attributes.

**restrictive selection** ***(boolean)*** This parameter indicates if the restrictive generator selection should be used. Execution is usually faster if this parameter is set to true.

**remove useless** ***(boolean)*** This parameter indicates if useless attributes should be removed.

**remove equivalent** ***(boolean)*** This parameter indicates if equivalent attributes should be removed.

**equivalence samples** ***(integer)*** *n* number of samples are checked to prove equivalency where *n* is the value of this parameter.

**equivalence epsilon** ***(real)*** Two attributes are considered equivalent if their difference is not bigger than epsilon.

**equivalence use statistics** ***(boolean)*** If this parameter is set to true, attribute statistics are recalculated before equivalence check.

**unused functions** ***(string)*** This parameter specifies the space separated list of functions which are not allowed in arguments for the attribute construction.

**constant generation prob** ***(real)*** This parameter specifies the probability for a generation of random constant attributes.

**associative attribute merging** ***(boolean)*** This parameter specifies if post processing should be performed after the crossover. It is only possible for runs with only one generator.

## Related Documents

- **Optimize by Generation (YAGGA)** (page 704)

## Tutorial Processes

### Applying YAGGA2 on the Polynomial data set

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes other then the label attribute. The Optimize by Generation (YAGGA2) operator is applied on the ExampleSet. It is a nested operator i.e. it has a subprocess. It is necessary for the

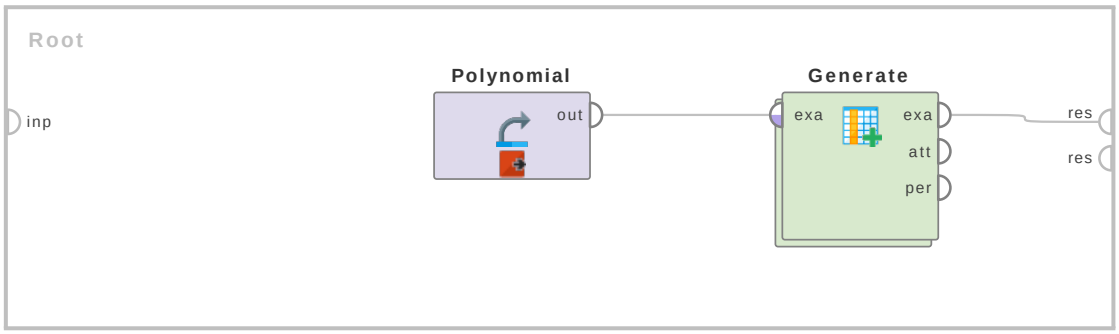
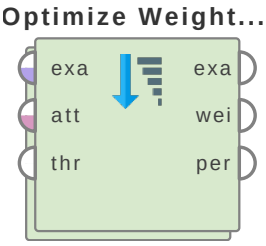



Figure 4.111: Tutorial process 'Applying YAGGA2 on the Polynomial data set'.

subprocess to deliver a performance vector which is used by the underlying Genetic Algorithm. Have a look at the subprocess of this operator. The Split Validation operator is used there which itself is a nested operator. Have a look at the subprocesses of the Split Validation operator. The Linear Regression operator is used in the 'Training' subprocess to train a model. The trained model is applied using the Apply Model operator in the 'Testing' subprocess. The performance is measured through the Performance (Regression) operator and the resultant performance vector is used by the underlying algorithm. Run the process and switch to the Results Workspace. You can see that the ExampleSet that had 5 attributes now has 7 attributes. All attributes were selected from the original attribute set and the attributes 'gensym5' and 'gensym6' were generated. The number of resultant attributes is not less than the number of original attributes because YAGGA2 is not an attribute reduction operator. It may (or may not) increase or decrease the number of attributes depending on what proves to have a better fitness.

# 4.7.4 Feature Weighting

## Optimize Weights (Evolutionary)




This operator calculates the relevance of the attributes of the given ExampleSet by using an evolutionary approach. The weights of the attributes are calculated using a Genetic Algorithm.

### Description

The Optimize Weights (Evolutionary) operator is a nested operator i.e. it has a subprocess. The subprocess of the Optimize Weights (Evolutionary) operator must always return a performance vector. For more information regarding subprocesses please study the Subprocess operator. The Optimize Weights (Evolutionary) operator calculates the weights of the attributes of the given ExampleSet by using a Genetic Algorithm. The higher the weight of an attribute, the more relevant it is considered.

A genetic algorithm (GA) is a search heuristic that mimics the process of natural evolution. This heuristic is routinely used to generate useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover.

In genetic algorithm 'mutation' means switching features on and off and 'crossover' means interchanging used features. Selection is done by the specified selection scheme which is selected by the *selection scheme* parameter. A genetic algorithm works as follows:

Generate an initial population consisting of $p$ individuals. The number $p$ can be adjusted by the *population size* parameter.

For all individuals in the population

1. Perform mutation, i.e. set used attributes to unused with probability $p\_m$ and vice versa. The probability $p\_m$ can be adjusted by the corresponding parameters.

2. Choose two individuals from the population and perform crossover with probability $p\_c$. The probability $p\_c$ can be adjusted by the *p crossover* parameter. The type of crossover can be selected by the *crossover type* parameter.

3. Perform selection, map all individuals according to their fitness and draw $p$ individuals at random according to their probability where $p$ is the population size which can be adjusted by the *population size* parameter.

4. As long as the fitness improves, go to step number 2.

If the ExampleSet contains value series attributes with block numbers, the whole block will be switched on and off. Exact, minimum or maximum number of attributes in combinations to be tested can be specified by the appropriate parameters. Many other options are also available for this operator. Please study the parameters section for more information.

## Input Ports

**example set in *(exa)*** This input port expects an ExampleSet. This ExampleSet is available at the first port of the nested chain (inside the subprocess) for processing in the subprocess.

**attribute weights in *(att)*** This port expects attribute weights. It is not compulsory to use this port.

**through *(thr)*** This operator can have multiple *through* ports. When one input is connected with the *through* port, another *through* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *through* port of this operator is available at the first *through* port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at the subprocess level.

## Output Ports

**example set out *(exa)*** The genetic algorithm is applied on the input ExampleSet. The resultant ExampleSet with reduced attributes is delivered through this port.

**weights *(wei)*** The attribute weights are delivered through this port.

**performance *(per)*** This port delivers the Performance Vector for the selected attributes. A Performance Vector is a list of performance criteria values.

## Parameters

**population size *(integer)*** This parameter specifies the population size i.e. the number of individuals per generation.

**maximum number of generations *(integer)*** This parameter specifies the number of generations after which the algorithm should be terminated.

**use early stopping *(boolean)*** This parameter enables early stopping. If not set to true, always the maximum number of generations are performed.

**generations without improval *(integer)*** This parameter is only available when the *use early stopping* parameter is set to true. This parameter specifies the stop criterion for early stopping i.e. it stops after *n* generations without improvement in the performance. *n* is specified by this parameter.

**normalize weights *(boolean)*** This parameter indicates if the final weights should be normalized. If set to true, the final weights are normalized such that the maximum weight is 1 and the minimum weight is 0.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

**show stop dialog *(boolean)*** This parameter determines if a dialog with a *stop* button should be displayed which stops the search for the best feature space. If the search for the best feature space is stopped, the best individual found till then will be returned.

**user result individual selection** ***(boolean)*** If this parameter is set to true, it allows the user to select the final result individual from the last population.

**show population plotter** ***(boolean)*** This parameter determines if the current population should be displayed in the performance space.

**population criteria data file** ***(filename)*** This parameter specifies the path to the file in which the criteria data of the final population should be saved.

**maximal fitness** ***(real)*** This parameter specifies the maximal fitness. The optimization will stop if the fitness reaches this value.

**selection scheme** ***(selection)*** This parameter specifies the selection scheme of this evolutionary algorithms.

**tournament size** ***(real)*** This parameter is only available when the *selection scheme* parameter is set to 'tournament'. It specifies the fraction of the current population which should be used as tournament members.

**start temperature** ***(real)*** This parameter is only available when the *selection scheme* parameter is set to 'Boltzmann'. It specifies the scaling temperature.

**dynamic selection pressure** ***(boolean)*** This parameter is only available when the *selection scheme* parameter is set to 'Boltzmann' or 'tournament'. If set to true the selection pressure is increased to maximum during the complete optimization run.

**keep best individual** ***(boolean)*** If set to true, the best individual of each generations is guaranteed to be selected for the next generation.

**save intermediate weights** ***(boolean)*** This parameter determines if the intermediate best results should be saved.

**intermediate weights generations** ***(integer)*** This parameter is only available when the *save intermediate weights* parameter is set to true. The intermediate best results would be saved every $k$ generations where $k$ is specified by this parameter.

**intermediate weights file** ***(filename)*** This parameter specifies the file into which the intermediate weights should be saved.

**mutation variance** ***(real)*** This parameter specifies the (initial) variance for each mutation.

**1 5 rule** ***(boolean)*** This parameter determines if the 1/5 rule for variance adaption should be used.

**bounded mutation** ***(boolean)*** If this parameter is set to true, the weights are bounded between 0 and 1.

**p crossover** ***(real)*** The probability for an individual to be selected for crossover is specified by this parameter.

**crossover type** ***(selection)*** The type of the crossover can be selected by this parameter.

**use default mutation rate** ***(boolean)*** This parameter determines if the default mutation rate should be used for nominal attributes.

**nominal mutation rate** ***(real)*** This parameter specifies the probability to switch nominal attributes between 0 and 1.

**initialize with input weights *(boolean)*** This parameter indicates if this operator should look for attribute weights in the given input and use them as a starting point for the optimization.

## Tutorial Processes

### Calculating the weights of the attributes of the Polynomial data set

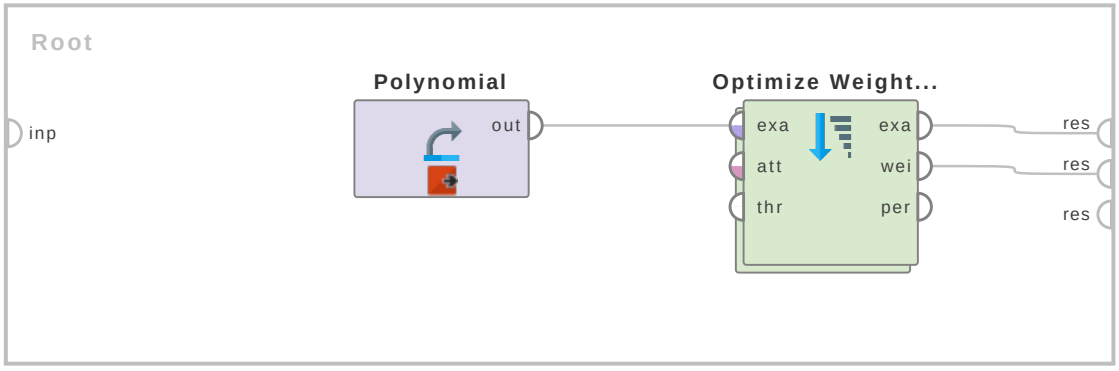
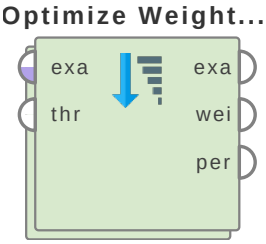



Figure 4.112: Tutorial process 'Calculating the weights of the attributes of the Polynomial data set'.

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes other than the label attribute. The Optimize Weights (Evolutionary) operator is applied on the ExampleSet which is a nested operator i.e. it has a subprocess. It is necessary for the subprocess to deliver a performance vector. This performance vector is used by the underlying Genetic Algorithm. Have a look at the subprocess of this operator. The Split Validation operator has been used there which itself is a nested operator. Have a look at the subprocesses of the Split Validation operator. The SVM operator is used in the 'Training' subprocess to train a model. The trained model is applied using the Apply Model operator in the 'Testing' subprocess. The performance is measured through the Performance operator and the resultant performance vector is used by the underlying algorithm. Run the process and switch to the Results Workspace. You can see that the ExampleSet that had 5 attributes has now been reduced to 2 attributes. Also take a look at the weights of the attributes in the Results Workspace. You can see that two attributes have weight 1 and the remaining attributes have weight 0.

# Optimize Weights (Forward)




This operator calculates the relevance of the attributes of the given ExampleSet by calculating the attribute weights. This operator assumes that the attributes are independent and optimizes the weights of the attributes with a linear search.

## Description

The Optimize Weights (Forward) operator is a nested operator i.e. it has a subprocess. The subprocess of the Optimize Weights (Forward) operator must always return a performance vector. For more information regarding subprocesses please study the Subprocess operator. The Optimize Weights (Forward) operator calculates the weights of the attributes of the given ExampleSet by using the performance vector returned by the subprocess. The higher the weight of an attribute, the more relevant it is considered.

This operator performs the weighting under the naive assumption that the features are independent from each other. Each attribute is weighted with a linear search. This approach may deliver good results after short time if the features indeed are not highly correlated.

## Differentiation

- **Optimize Weights (Evolutionary)** The Optimize Weights (Evolutionary) operator calculates the relevance of the attributes of the given ExampleSet by using an evolutionary approach. The weights of the attributes are calculated using a Genetic Algorithm. See page 713 for details.

## Input Ports

**example set in *(exa)*** This input port expects an ExampleSet. This ExampleSet is available at the first port of the nested chain (inside the subprocess) for processing in the subprocess.

**through *(thr)*** This operator can have multiple *through* ports. When one input is connected with the *through* port, another *through* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *through* port of this operator is available at the first *through* port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at subprocess level.

## Output Ports

**example set out *(exa)*** The resultant ExampleSet with reduced attributes is delivered through this port.

**weights *(wei)*** The attribute weights are delivered through this port.

**performance *(per)*** This port delivers the Performance Vector for the selected attributes. A Performance Vector is a list of performance criteria values.

## Parameters

**keep best *(integer)*** This parameter specifies the number of best individuals to keep in each generation.

**generations without improval *(integer)*** This parameter specifies the stop criterion for early stopping i.e. it stops after *n* generations without improvement in the performance. *n* is specified by this parameter.

**weights *(string)*** This parameter specifies the weights to be used for the creation of individuals in each generation.

**normalize weights *(boolean)*** This parameter indicates if the final weights should be normalized. If set to true, the final weights are normalized such that the maximum weight is 1 and the minimum weight is 0.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is available only if the *use local random seed* parameter is set to true.

**show stop dialog *(boolean)*** This parameter determines if a dialog with a *stop* button should be displayed which stops the search for the best feature space. If the search for the best feature space is stopped, the best individual found till then will be returned.

**user result individual selection *(boolean)*** If this parameter is set to true, it allows the user to select the final result individual from the last population.

**show population plotter *(boolean)*** This parameter determines if the current population should be displayed in the performance space.

**plot generations *(integer)*** This parameter is only available when the *show population plotter* parameter is set to true. The population plotter is updated in these generations.

**constraint draw range *(boolean)*** This parameter is only available when the *show population plotter* parameter is set to true. This parameter determines if the draw range of the population plotter should be constrained between 0 and 1.

**draw dominated points *(boolean)*** This parameter is only available when the *show population plotter* parameter is set to true. This parameter determines if only points which are not Pareto dominated should be drawn on the population plotter.

**population criteria data file *(filename)*** This parameter specifies the path to the file in which the criteria data of the final population should be saved.

**maximal fitness *(real)*** This parameter specifies the maximal fitness. The optimization will stop if the fitness reaches this value.

## Related Documents

- **Optimize Weights (Evolutionary)** (page 713)

## Tutorial Processes

### Calculating the weights of the attributes of the Polynomial data set

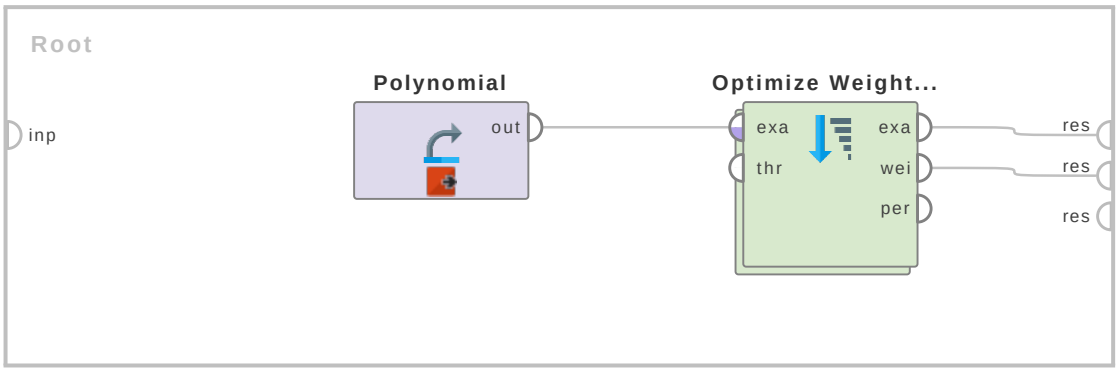
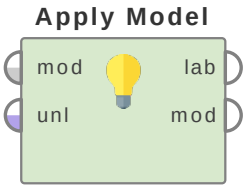



Figure 4.113: Tutorial process 'Calculating the weights of the attributes of the Polynomial data set'.

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes other then the label attribute. The Optimize Weights (Forward) operator is applied on the ExampleSet which is a nested operator i.e. it has a subprocess. It is necessary for the subprocess to deliver a performance vector. This performance vector is used by the underlying algorithm. Have a look at the subprocess of this operator. The Split Validation operator has been used there which itself is a nested operator. Have a look at the subprocesses of the Split Validation operator. The SVM operator is used in the 'Training' subprocess to train a model. The trained model is applied using the Apply Model operator in the 'Testing' subprocess. The performance is measured through the Performance operator and the resultant performance vector is used by the underlying algorithm. Run the process and switch to the Results Workspace. You can see that the ExampleSet that had 5 attributes has now been reduced to 2 attributes. Also take a look at the weights of the attributes in the Results Workspace. You can see that two attributes have non-zero weight and the remaining attributes have weight 0.

# 5Scoring

## Apply Model




This Operator applies a model on an ExampleSet.

### Description

A model is first trained on an ExampleSet by another Operator, which is often a learning algorithm. Afterwards, this model can be applied on another ExampleSet. Usually, the goal is to get a prediction on unseen data or to transform data by applying a preprocessing model.

The ExampleSet upon which the model is applied, has to be compatible with the Attributes of the model. This means, that the ExampleSet has the same number, order, type and role of Attributes as the ExampleSet used to generate the model.

### Differentiation

- **Group Models**

  If you want to apply several models in a row you can use the Group Models Operator. This is helpful if you for example want to apply preprocessing models before applying a prediction model.

  See page 406 for details.

### Input Ports

**Model *(Mod)*** This port expects a model. The number, order, type and role of Attributes of the ExampleSet on which this model was trained has to be consistent with the ExampleSet on the *unlabeled data* input port.

**Unlabelled data *(Unl)*** This port expects an ExampleSet. The number, order, type and role of Attributes of this ExampleSet has to be consistent with ExampleSet on which the model delivered to the *model* input port was trained.

### Output Ports

**labelled data *(lab)*** The ExampleSet delivered from this port is changed by means of the model. For the case of predictions, new Attributes like 'prediction(*Label*)' and 'confidence(*Value*)' are added. Applying preprocessing models updates the existing ExampleSet.

**model *(mod)*** The input model is passed without changing to the output through this port.

## Parameters

**application parameters** This parameter can change the settings of certain models before they are applied to provided ExampleSet. This is only possible for a few Operators and can be considered a legacy option.

**create view** If the model applied at the input port supports Views, it is possible to create a View instead of changing the underlying data. If this option is checked, the application of the model is delayed until the transformations are needed. Most models no longer support Views and it can be considered a legacy option.

## Tutorial Processes

### Train and apply a linear regression model

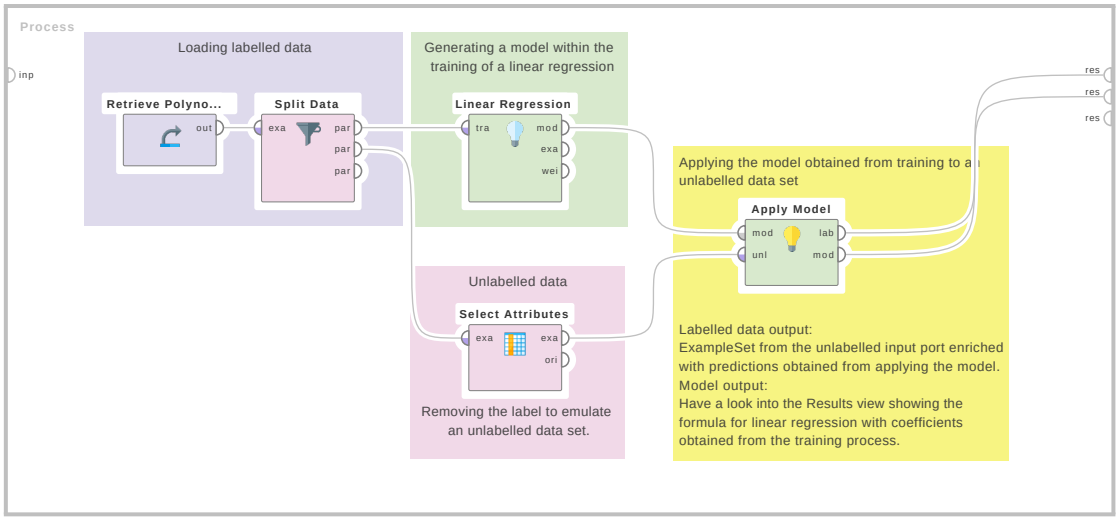
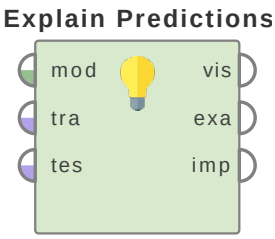



Figure 5.1: Tutorial process 'Train and apply a linear regression model'.

In this tutorial Process, a model is created within the training of a linear regression.

This tutorial Process first trains a linear regression model on the 'Polynomial' ExampleSet. Afterwards, the model is applied on an unlabelled 'Polynomial' ExampleSet. The resulting output ExampleSet has a new Attribute: 'prediction(label)'.

# Explain Predictions




This operator identifies the attributes that play the largest role when making a prediction.

## Description

Given a model and an input, you can generate a prediction, but which of the attributes plays the largest role in forming that prediction? This operator takes a model and an ExampleSet as input, and generates a table highlighting the attributes that most strongly support (green) or contradict (red) each prediction. Alternatively, the table can be displayed with two extra columns (support and predict) containing numeric details.

For each Example in an ExampleSet, this operator generates a neigboring set of data points, and uses correlation to identify the local attribute weights in that neighborhood. Although the relationship between attributes and predictions may be highly non-linear globally, the local linear relationship is more than powerful enough to explain the predictions.

This operator works with all data types and data sizes. It supports both classification and regression problems. The only model type which is not recommended is k-Nearest Neighbors, since this model typically suffers from long runtimes.

## Input Ports

**model *(mod)*** This input port expects a model.

**training data *(tra)*** This input port expects an ExampleSet identical to the one that trained the model.

**test data *(tes)*** This input port expects an ExampleSet with test data.

## Output Ports

**visualization output *(vis)*** This output port displays the test data with predictions and color highlighting of attributes: green when the value of the attribute supports the prediction, and red when the value of the attribute contradicts the prediction.

**example set output *(exa)*** This output port displays the test data with predictions and two extra columns: one that details the attributes that support the prediction and one that details the attributes that contradict the prediction.

**importances output *(imp)*** This output port displays the test data in a long table format including the importance of all attributes for each row. This can be useful if the data should be visualized later on.

## Parameters

**maximal explaining attributes *(integer)*** The maximal number of attributes used to support the predictions, also the maximal number of attributes used for contradicting it. The whole point about explanations is that they allow you to focus on the factors that matter in each particular case. We recommend a value of 3 to achieve this but you can increase this number if you feel that you need more factors to explain the predictions to you. Please note that you might end up with less factors if only less attribute values than the maximal number support or contradict a prediction in this case.

**local sample size *(integer)*** The number of locally generated samples around each prediction data point to identify the attributes with the biggest impact on this decision. You might want to increase this number for high-dimensional data sets in case the quality of predictions become worse. Please note that the runtime of this algorithm slows down with higher numbers. In general, a value of around 500 delivers high-quality explanations in a reasonable amount of time.

## Tutorial Processes

### Explaining Predictions for Titanic

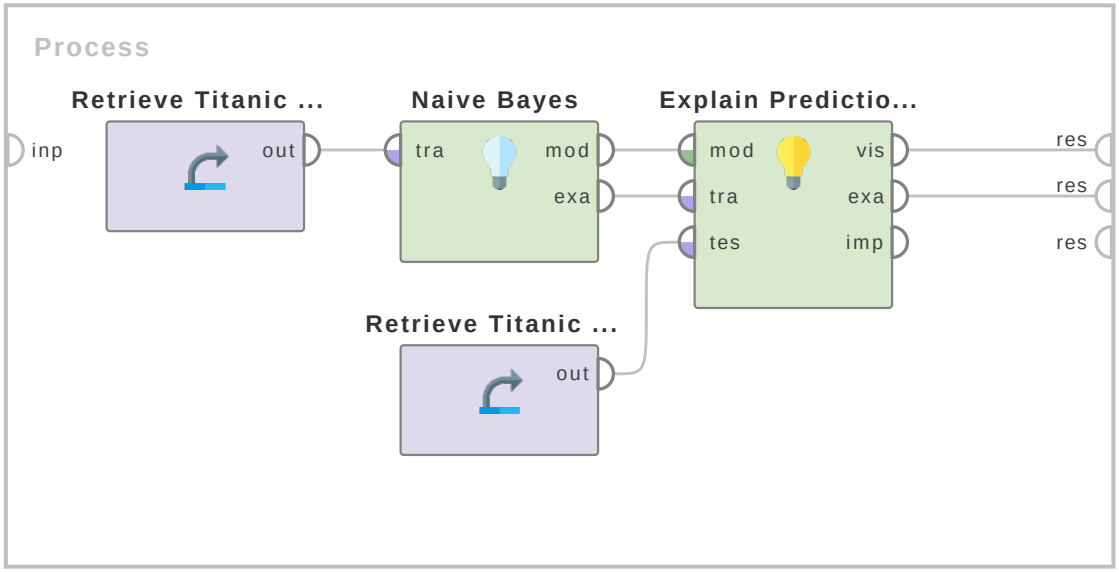


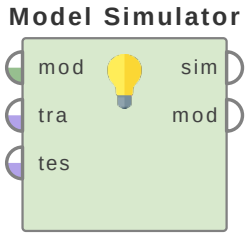

Figure 5.2: Tutorial process 'Explaining Predictions for Titanic'.

This process trains a Naive Bayes model on the Titanic data. It then uses the Explain Predictions operator to create the predictions and all local explanations for the second data set.

You can see the two results. First the data with additional columns for the predictions, the confidences, and the new explanations. The other result directly visualizes the explanations with colors. Green means a value which strongly supports the prediction. Red means that this value contradicts the prediction. Have a look at the 3rd row for example. The model predicts "Yes" for survival despite the fact that the gender is male. In general, most men died during the accident though so the model made this prediction based on the other values. In this case, this

would be the age of 71, the amount of money paid, and the fact that this person traveled without parents or children.

# Model Simulator




This Operator provides an easy, real-time method to change the inputs to a model and view the output. It shows predictions, confidences, and explanations for those inputs.

## Description

The outputs are designed to achieve three goals: First, users will get a better understanding of how the model comes to its conclusions, even for black box models like deep learning neural networks. Second, users can simulate cases where they know the outcome, and check if the model behaves as expected. Third, users can use the built-in optimization method to find optimal input settings in order to achieve a desired outcome. The latter turns predictive models into prescriptive models.

The result is displayed in two panels. In the left panel, users can change the input settings for all attributes, while in the right panel, the outputs are calculated and displayed in real time. Each input attribute (independent variable) of the model is displayed in a row, together with a user interface element corresponding to the value type of the attribute. At the end of each row is a little information symbol; when hovered, it displays additional information about the attribute, including statistics and the distribution of values. The length of the gray bars below each attribute name depicts the global importance of this attribute for the model (in contrast to the local importance for each specific prediction, which will be discussed below), based on its correlation with the predictions.

Users can select categorical values from a drop-down element, turn binary values on or off, and move numerical sliders to arbitrary values within the range defined by the minimum and maximum. Please note that attributes with value type date are not supported.

The “Optimize” button at the bottom of the input panel spawns a dialog enabling the user to determine the optimal input values needed to obtain a desired output. Also constrained optimizations are supported. When the optimization is completed, the optimal input values are displayed in the input panel.

All the outputs can be found on the right side and are calculated in real-time. There are five different parts which slightly differ depending on if you have a classification or a regression problem.

- Most Likely / Prediction (top left): You can easily see what the current prediction would be. It shows the most likely class in case of classification and the predicted number in case of regression tasks. A bar chart showing the confidences for other likely classes is also shown in case of classification.
- Confidence Distribution / Distributions of Prediction (top right): In case of classification, you will see the distribution of all confidence values for this class on a test data set if it was provided. The current confidence is highlighted. In case of regression, you will see how the current prediction relates to the distribution of predictions on a test set. Again, the distribution is only shown if a test set was provided.
- Important Factors (bottom left): You can see how much the most important attributes contribute to the current prediction. An attribute value can either support a prediction

(green bar) or contradict it (red bar). In contrast with the global importance of an attribute described previously (the gray bar in the input panel), the local importance of an attribute is based on its correlation with the predictions in the neighborhood of the selected input. See also the documentation for the Operator Explain Predictions.

- Accuracy (bottom right): If a test data set was provided, and if it contains a label attribute, you will see how accurate the model works overall and for the currently predicted class (in case of classification).
- Interpretation (bottom): A short summary of some major and outstanding points of all of the results above.

The simulator works well independent of the training data size. It has been successfully used for more than 10 Million data rows. The number of attributes has an impact though. It works well for less than 1,000 columns. In this case, the simulator provides all calculations in real time. For more than 1,000 columns, the real-time updates of the local feature importance is disabled. The automatic optimization of input features is disabled for more than 10,000 input features.

The model simulator supports all model types. The one exception are k-Nearest Neighbors models for massive amounts of training data since the model application time of this model type is too slow to support interactive, real-time exploration. Hence, we do not recommend to use the simulator or the optimization for k-Nearest Neighbors models.

## Input Ports

**model *(mod)*** This input port expects a model.

**input *(inp)*** This input port expects an ExampleSet identical to the one that trained the model.

**input *(inp)*** This input port expects an ExampleSet with test data. This data is optional.

## Output Ports

**simulator output *(sim)*** This port delivers the model simulator, used to simulate inputs and observe the model's behavior. It also provides an optimization algorithm which finds the optimal input needed to provide a desired output.

**model *(mod)*** The input model is passed without changing to the output through this port.

## Tutorial Processes

### Model Simulator for the Titanic data

This process trains a Naive Bayes model on the Titanic data. It then uses the Model Simulator operator to create a new user interface for simulating model input and observing the model's output in real-time. Can you find out how likely it is that you personally would survive when buying a third class ticket? Also, what is the best situation you could be in given your age and gender?

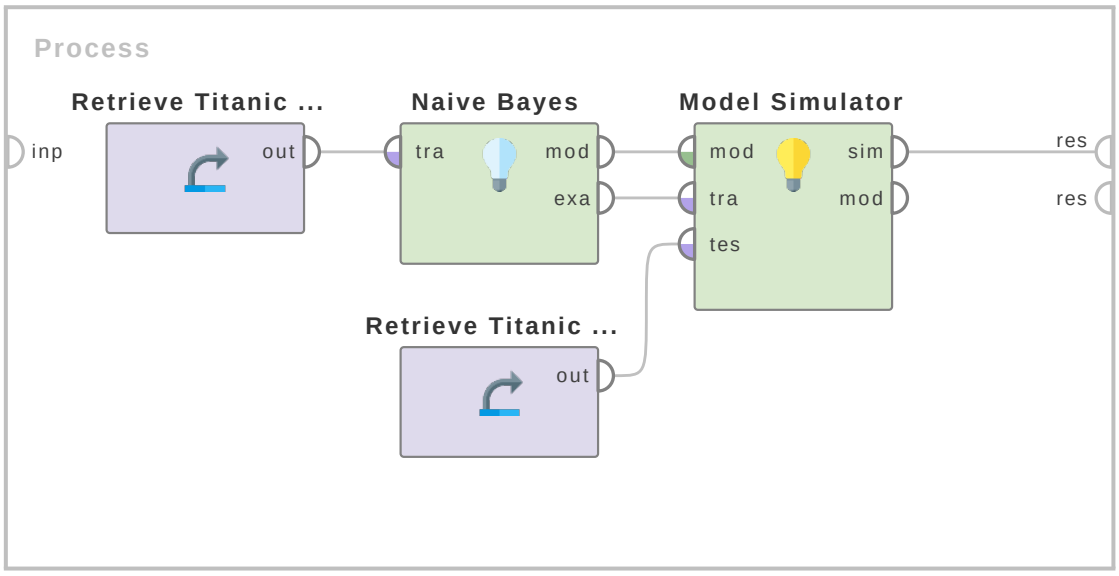
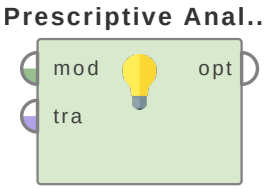



Figure 5.3: Tutorial process 'Model Simulator for the Titanic data'.

# Prescriptive Analytics




Given a model and a desired output, this operator automatically finds the optimal inputs.

## Description

In predictive modeling, a model is used to predict an outcome, given an input. This operator reverses that procedure, starting with a model and a desired output, and prescribing an optimized input to achieve the desired outcome.

The operator uses an evolutionary optimization method, based on the model, with one of the following targets:

- minimize confidence for a class
- maximize confidence for a class
- get as close as possible to a certain confidence for a class
- minimize regression prediction
- maximize regression prediction
- get as close as possible to a certain regression prediction

The training data can be used to to constrain the optimization, so that all numerical values satisfy one or more of the following conditions:

- stay close to the average, within 1 / 2 / 3 times the standard deviation
- stay above the minimum
- stay below the maximum
- stay above a certain value
- stay below a certain value

Moreover, the user may assign constant values to any of the attributes, overriding the above conditions.

## Input Ports

**model *(mod)*** This port expects a model, whose optimal inputs should be identified.

**training data *(tra)*** This port expects an ExampleSet, the same ExampleSet that was used to create the model.

## Output Ports

**optimal data *(opt)*** The optimal data which, when used as an input to the model, delivers the desired result.

## Parameters

**classification *(boolean)*** Indicates if the model is a classification model or a regression model.

**class name *(string)*** The class for which the confidence should be optimized.

**optimization direction *(selection)*** The optimization strategy: minimize, maximize, or specify a value. A specific value can be useful for regression / forecasting problems.

**value to reach *(real)*** Specify a confidence or regression value which should be reached. Only available if the value for "optimization direction" is "specific value".

**stay around average (numerical) *(boolean)*** Indicates if numerical values should stay in a specified range around the average value which helps to prevent extreme values which might be not feasible as inputs.

**standard deviations around average *(real)*** Defines the number of standard deviations the values can move away from the numerical average.

**stay above global minimum (numerical) *(boolean)*** Indicates if numerical values should stay above the minimum value of the corresponding attribute.

**stay below global maximum (numerical) *(boolean)*** Indicates if numerical values should stay below the maximum value of the corresponding attribute.

**stay above value (numerical) *(boolean)*** Indicates if numerical values should stay above a specified value.

**minimum value *(real)*** Attribute values during optimization should stay above this value.

**stay below value (numerical)** Indicates if numerical values should stay below a specified value.

**maximum value *(real)*** Attribute values during optimization should stay below this value.

**constant attribute values *(list)*** A list of attributes which should be kept at constant values. You can specify name-value pairs with the attribute name on the left and the desired constant value on the right.

**limit type *(selection)*** Defines when the optimization ends. No limit uses a heuristic to detect the optimum. Time limit stops after specified time. Generations stops after the specified number of generations is reached.

**maximum generations *(integer)*** The maximum number of generations for the evolutionary optimization algorithm. Only available if the limit is "generations and population size".

**population size *(integer)*** The number of individuals in the population of the evolutionary optimization algorithm. Only available if the limit is "generations and population size".

**time limit (in seconds) *(integer)*** The maximum number of seconds the optimization will run. Only available if the limit is "time limit".

## Tutorial Processes

### Prescriptive Analytics for Titanic

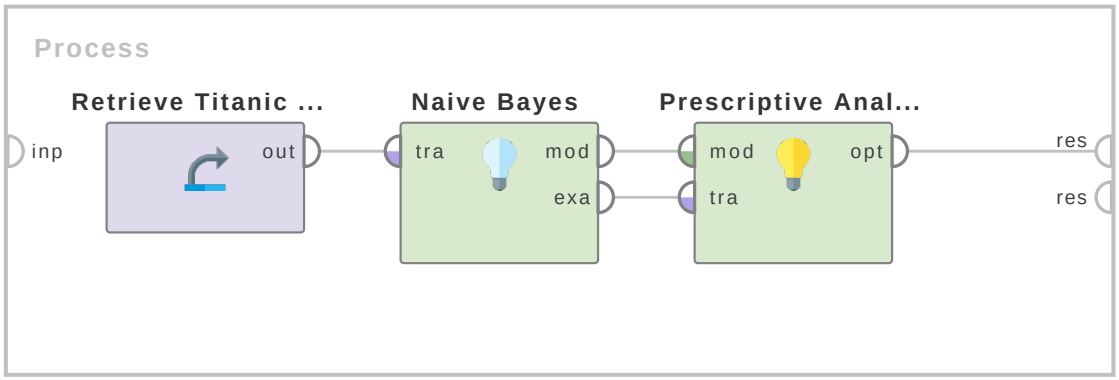



Figure 5.4: Tutorial process 'Prescriptive Analytics for Titanic'.

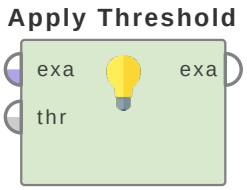
This process trains a Naive Bayes model on the Titanic data. It then uses the operator Prescriptive Analytics to find the optimal attribute values which maximize the likelihood for survival.

Please note that most default parameter values will deliver reasonable results without going to the extremes. But we made some important settings. First, we defined that this is a classification problem and that we want to maximize the confidence for the prediction of "Yes". We also set some constant values which are things which you cannot easily change when being a passenger of the Titanic. In this case, this would be the age of the person and the gender. We used the values 40 and Female here.

After the process is executed, you will get a new ExampleSet as a result which will show the optimal settings in this case. If you purchase a first class ticket for $133 and only travel with one parent or child, you will have a 99% likelihood of survival.

# 5.1 Confidences

## Apply Threshold




This operator applies a threshold on soft classified data.

### Description

The Apply Threshold operator applies the given threshold to a labeled ExampleSet and maps a soft prediction to crisp values. The threshold is provided through the *threshold* port. Mostly the Create Threshold operator is used for creating thresholds before it is applied using the Apply Threshold operator. If the confidence for the second class is greater than the given threshold the prediction is set to this class otherwise it is set to the other class. This can be easily understood by studying the attached Example Process.

Among various classification methods, there are two main groups of methods: soft and hard classification. In particular, a soft classification rule generally estimates the class conditional probabilities explicitly and then makes the class prediction based on the largest estimated probability. In contrast, hard classification bypasses the requirement of class probability estimation and directly estimates the classification boundary.

### Input Ports

**example set *(exa)*** This input port expects a labeled ExampleSet. The ExampleSet should have *label* and *prediction* attributes as well as attributes for confidence of predictions.

**threshold *(thr)*** The threshold is provided through this input port. Frequently, the Create Threshold operator is used for providing threshold at this port.

### Output Ports

**example set *(exa)*** The predictions of the input ExampleSet are changed according to the threshold given at the *threshold* port and the modified ExampleSet is delivered through this port.

### Tutorial Processes

#### Creating and Applying thresholds

This Example Process starts with a Subprocess operator. This subprocess provides the labeled ExampleSet. Double-click on the Subprocess operator to see what is happening inside the subprocess although it is not directly relevant to the use of the Apply Threshold operator. In the subprocess, the K-NN classification model is learned and applied on different samples of the 'Weighting' data set. The resultant labeled ExampleSet is output of this subprocess. A breakpoint is inserted after this subprocess so that you can have a look at the labeled ExampleSet before the application of the Apply Threshold operator. You can see that the ExampleSet has 20 examples. 11 of them are predicted as 'positive' and the remaining 9 examples are predicted

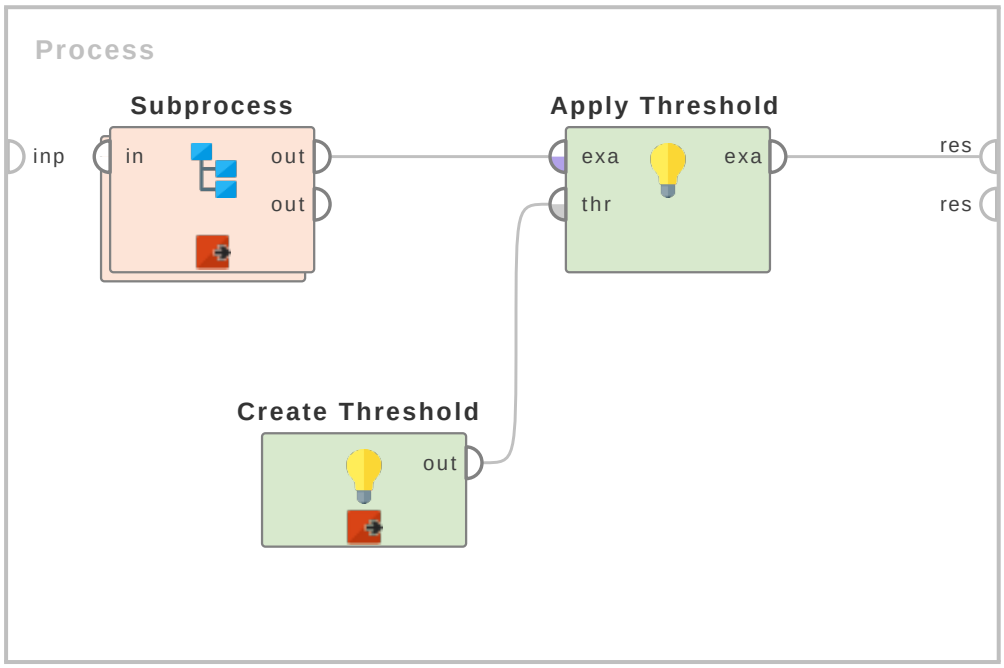



Figure 5.5: Tutorial process ‘Creating and Applying thresholds’.

as ‘negative’. If you sort the results according to the confidence of positive prediction, you will easily see that among 11 examples predicted as ‘positive’, 3 examples have confidence 0.600, 4 examples have confidence 0.700, 3 examples have confidence 0.800 and 1 example has confidence 0.900.

Now let us have a look at what is happening outside the subprocess. The Create Threshold operator is used for creating a threshold. The threshold parameter is set to 0.700 and the first class and second class parameters are set to ‘negative’ and ‘positive’ respectively. This threshold is applied on the labeled ExampleSet using the Apply Threshold operator. We know that when the Apply Threshold operator is applied on an ExampleSet, if the confidence for the second class is greater than the given threshold then the prediction is set to this class otherwise it is set to the other class. In this process, if the confidence for the second class i.e. ‘positive’ (class specified in the second class parameter of the Create Threshold operator) is greater than the given threshold i.e. 0.700 (threshold specified in the threshold parameter of the Create Threshold operator) the prediction is set to ‘positive’ otherwise it is set to ‘negative’. In the labeled ExampleSet only 4 examples had confidence (positive) greater than 0.700. When the Apply Threshold operator is applied only these 4 examples are assigned ‘positive’ prediction and all other examples are assigned ‘negative’ predictions.

# Create Threshold


This operator creates a user defined threshold for crisp classification based on the prediction confidences (soft predictions). This threshold can be applied by using the Apply Threshold operator.

## Description

The *threshold* parameter specifies the required threshold. The *first class* and *second class* parameters are used for specifying the classes of the ExampleSet that should be considered as first and second class respectively. The threshold created by this operator can be applied on the labeled ExampleSet using the Apply Threshold operator. Should it occur that the confidence for the second class is greater than the given threshold then the prediction is set to this second class otherwise it is set to the first class. This can be easily understood by studying the attached Example Process.

The Apply Threshold operator applies the given threshold to a labeled ExampleSet and maps a soft prediction to crisp values. The threshold is provided through the *threshold* port. Mostly the Create Threshold operator is used for creating thresholds before they are applied using the Apply Threshold operator.

Among various classification methods, there are two main groups of methods: soft and hard classification. In particular, a soft classification rule generally estimates the class conditional probabilities explicitly and then makes the class prediction based on the largest estimated probability. In contrast, hard classification bypasses the requirement of class probability estimation and directly estimates the classification boundary.

## Output Ports

**output *(out)*** This port delivers the threshold. This threshold can be applied on a labeled ExampleSet by using the Apply Threshold operator.

## Parameters

**threshold *(real)*** This parameter specifies the threshold of the prediction confidence. It should be in range 0.0 to 1.0. If the prediction confidence for the second class is greater than this threshold the prediction is set to second class (i.e. the class specified through the *second class* parameter) otherwise it is set to the first class(i.e. the class specified through the *first class* parameter).

**first class *(string)*** This parameter specifies the class which should be considered as the first class.

**second class *(string)*** This parameter specifies the class which should be considered as the second class.

## Tutorial Processes

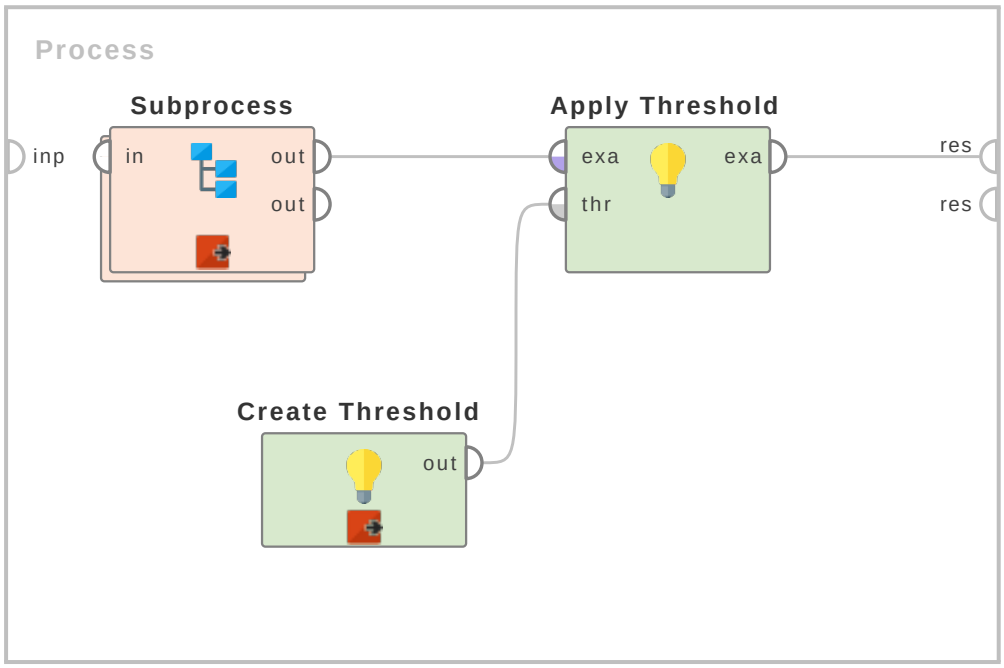



Figure 5.6: Tutorial process 'Creating and Applying thresholds'.

### Creating and Applying thresholds

This Example Process starts with a Subprocess operator. This subprocess provides the labeled ExampleSet. Double-click on the Subprocess operator to see what is happening inside the subprocess although it is not directly relevant to the use of the Create Threshold operator. In the subprocess, the K-NN classification model is learned and applied on different samples of the 'Weighting' data set. The resultant labeled ExampleSet is output of this subprocess. A breakpoint is inserted after this subprocess so that you can have a look at the labeled ExampleSet before the application of the Create Threshold and Apply Threshold operators. You can see that the ExampleSet has 20 examples. 11 of them are predicted as 'positive' and the remaining 9 examples are predicted as 'negative'. If you sort the results according to the confidence of positive prediction, you will easily see that among 11 examples predicted as 'positive', 3 examples have confidence 0.600, 4 examples have confidence 0.700, 3 examples have confidence 0.800 and 1 example has confidence 0.900.

Now let us have a look at what is happening outside the subprocess. The Create Threshold operator is used for creating a threshold. The threshold parameter is set to 0.700 and the first class and second class parameters are set to 'negative' and 'positive' respectively. A breakpoint is inserted here so that you can see the threshold in the Results Workspace. This statement in the Results Workspace explains everything:

if confidence(positive) $>$ 0.7 then positive; else negative

This statement means that if confidence(positive) is greater than 0.7 then the class should be predicted as positive otherwise it should be predicted as negative. In a general form this statement would look something like this:

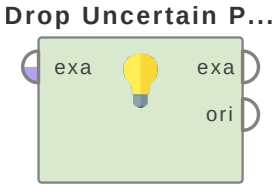
if confidence(second) $>$ T then second; else first.

where T, second and first are the values of the threshold, second class and first class parameters respectively.

This threshold is applied on the labeled ExampleSet using the Apply Threshold operator. We

know that when the Apply Threshold operator is applied on an ExampleSet there are two possibilities: if the confidence for the second class is greater than the given threshold the prediction is set to second otherwise to the first class. In this process, if the confidence for the second class i.e. 'positive' (class specified in the second class parameter of the Create Threshold operator) is greater than the given threshold i.e. 0.700 (threshold specified in the threshold parameter of the Create Threshold operator) the prediction is set to 'positive' otherwise it is set to 'negative'. In the labeled ExampleSet only 4 examples had confidence (positive) greater than 0.700. When the Apply Threshold operator is applied only these 4 examples are assigned 'positive' predictions and all other examples are assigned 'negative' predictions.

# Drop Uncertain Predictions




This operator sets all predictions to 'unknown' (missing value) if the corresponding confidence is less than the specified minimum confidence. This operator is used for dropping predictions with low confidence values.

## Description

The Drop Uncertain Predictions operator expects a labeled ExampleSet i.e. an ExampleSet with label and prediction attributes along with prediction confidences. The minimum confidence threshold is specified through the *min confidence* parameter. All those predictions of the given ExampleSet are dropped where the corresponding prediction confidence is below the specified threshold. Suppose an ExampleSet with two possible classes 'positive' and 'negative'. If the *min confidence* parameter is set to 0.700, all the examples that were predicted as 'positive' but their corresponding 'confidence (positive)' value is less than 0.700 are classified as missing values. Similarly the label value is set to missing value for all those examples that were predicted as 'negative' but their corresponding confidence '(negative)' value is less than 0.700. This operator also allows you to define different minimum confidence thresholds for different classes through the *min confidences* parameter.

## Input Ports

**example set input *(exa)*** This input port expects a labeled ExampleSet. It is the output of the Apply Model operator in the attached Example Process. The output of other operators can also be used as input if it is a labeled ExampleSet.

## Output Ports

**example set output *(exa)*** The uncertain predictions are dropped and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**class handling *(selection)*** This parameter specifies the mode of class handling which defines if all classes are handled equally or if individual class thresholds are set.

- **balanced** In this case all classes are handled equally i.e. the same confidence threshold is applied on all possible values of the label. The minimum confidence threshold is specified through the *min confidence* parameter.
- **unbalanced** In this case classes are not handled equally i.e. different confidence thresholds can be specified for different classes through the *min confidences* parameter.

**min confidence *(real)*** This parameter is only available when the *class handling* parameter is set to 'balanced'. This parameter sets the minimum confidence threshold for all the classes. Predictions below this confidence will be dropped.

**min confidences *(list)*** This parameter is only available when the *class handling* parameter is set to 'unbalanced'. This parameter specifies individual thresholds for classes. Predictions below these confidences will be dropped.

## Tutorial Processes

### Dropping uncertain predictions of the Naive Bayes operator

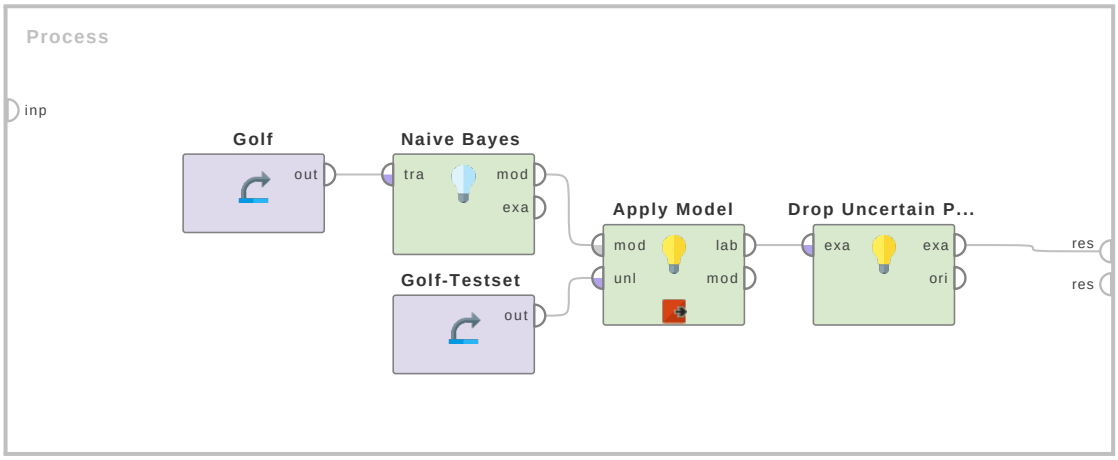
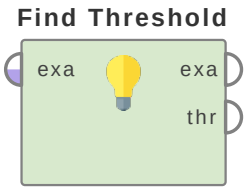



Figure 5.7: Tutorial process 'Dropping uncertain predictions of the Naive Bayes operator'.

The 'Golf' data set is loaded using the Retrieve operator. The Naive Bayes operator is applied on it to generate a classification model. The resultant classification model is applied on the 'Golf-Testset' data set by using the Apply Model operator. A breakpoint is inserted here so that you can see the labeled ExampleSet generated by the Apply Model operator. You can see that 10 examples have been classified as 'yes' but only 6 of them have 'confidence (yes)' above 0.700. Only 2 examples have been classified as 'no' but only 1 of them has 'confidence (no)' above 0.700. This labeled ExampleSet is provided to the Drop Uncertain Predictions operator. The min confidence parameter is set to 0.7. Thus all the examples where the prediction confidence is below 0.7 are set to missing values. This can be seen in the Results Workspace. 7 examples had a prediction confidence below 0.7 and all of them have been dropped.

# Find Threshold




This operator finds the best threshold for crisp classification of soft classified data based on user defined costs. The optimization step is based on ROC analysis.

## Description

This operator finds the threshold for given prediction confidences of soft classified predictions in order to turn it into a crisp classification. The optimization step is based on ROC analysis. ROC is discussed at the end of this description.

The Find Threshold operator finds the threshold of a labeled ExampleSet to map a soft prediction to crisp values. The threshold is delivered through the *threshold* port. Mostly the Apply Threshold operator is used for applying a threshold after it has been delivered by the Find Threshold operator. If the confidence for the second class is greater than the given threshold the prediction is set to this class otherwise it is set to the other class. This can be easily understood by studying the attached Example Process.

Among various classification methods, there are two main groups of methods: soft and hard classification. In particular, a soft classification rule generally estimates the class conditional probabilities explicitly and then makes the class prediction based on the largest estimated probability. In contrast, hard classification bypasses the requirement of class probability estimation and directly estimates the classification boundary.

Receiver operating characteristic (ROC), or simply ROC curve, is a graphical plot of the true positive rate vs. false positive rate for a binary classifier system as its discrimination threshold is varied. The ROC can also be represented equivalently by plotting the fraction of true positives out of the positives (TP/P = true positive rate) vs. the fraction of false positives out of the negatives (FP/N = false positive rate). TP/P determines a classifier or a diagnostic test performance on classifying positive instances correctly among all positive samples available during the test. FP/N, on the other hand, defines how many incorrect positive results occur among all negative samples available during the test.

A ROC space is defined by FP/N and TP/P as x and y axes respectively, which depicts relative trade-offs between true positive (benefits) and false positive (costs). Each prediction result or one instance of a confusion matrix represents one point in the ROC space.The best possible prediction method would yield a point in the upper left corner or coordinate (0,1) of the ROC space, representing 100% TP/P and 0% FP/N. The (0,1) point is also called a perfect classification. A completely random guess would give a point along a diagonal line from the left bottom to the top right corners.

The diagonal divides the ROC space. Points above the diagonal represent good classification results, points below the line represent poor results. Note that the Find Threshold operator finds a threshold where points of bad classification are inverted to convert them to good classification.

## Input Ports

**example set *(exa)*** This input port expects a labeled ExampleSet. The ExampleSet should have *label* and *prediction* attributes as well as attributes for the confidence of predictions.

## Output Ports

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**threshold *(thr)*** The threshold is delivered through this output port. Frequently, the Apply Threshold operator is used for applying this threshold on the soft classified data.

## Parameters

**define labels *(boolean)*** This is an expert parameter. If set to true, the first and second label can be defined explicitly using the *first label* and *second label* parameters.

**first label *(string)*** This parameter is only available when the *define labels* parameter is set to true. It explicitly defines the first label.

**second label *(string)*** This parameter is only available when the *define labels* parameter is set to true. It explicitly defines the second label.

**misclassification costs first *(real)*** This parameter specifies the costs assigned when an example of the first class is misclassified as one of the second.

**misclassification costs second *(real)*** This parameter specifies the costs assigned when an example of the second class is misclassified as one of the first.

**show roc plot *(boolean)*** This parameter indicates whether to display a plot of the ROC curve.

**use example weights *(boolean)*** This parameter indicates if example weights should be used.

**roc bias *(selection)*** This is an expert parameter. It determines how the ROC (and AUC) are evaluated.

## Tutorial Processes

### Introduction to the Find Threshold operator

This Example Process starts with a Subprocess operator. This subprocess provides the labeled ExampleSet. Double-click on the Subprocess operator to see what is happening inside although it is not directly relevant to the understanding of the Find Threshold operator. In the subprocess, the Generate Data operator is used for generation of testing and training data sets with binominal label. The SVM classification model is learned and applied on training and testing data sets respectively. The resultant labeled ExampleSet is output of this subprocess. A breakpoint is inserted after this subprocess so that you can have a look at the labeled ExampleSet before application of the Find Threshold operator. You can see that the ExampleSet has 500 examples. If you sort the results according to the confidence of positive prediction, and scroll through the data set, you will see that all examples with 'confidence(positive)' greater than 0.500 are classified as positive and all examples with 'confidence(positive)' less than 0.500 are classified as negative.

Now have a look at what is happening outside the subprocess. The Find Threshold operator is used for finding a threshold. All its parameters are used with default values. The Find Threshold operator delivers a threshold through the threshold port. This threshold is applied on the labeled ExampleSet using the Apply Threshold operator. We know that when the Apply Threshold operator is applied on an ExampleSet, if the confidence for the second class is greater than the

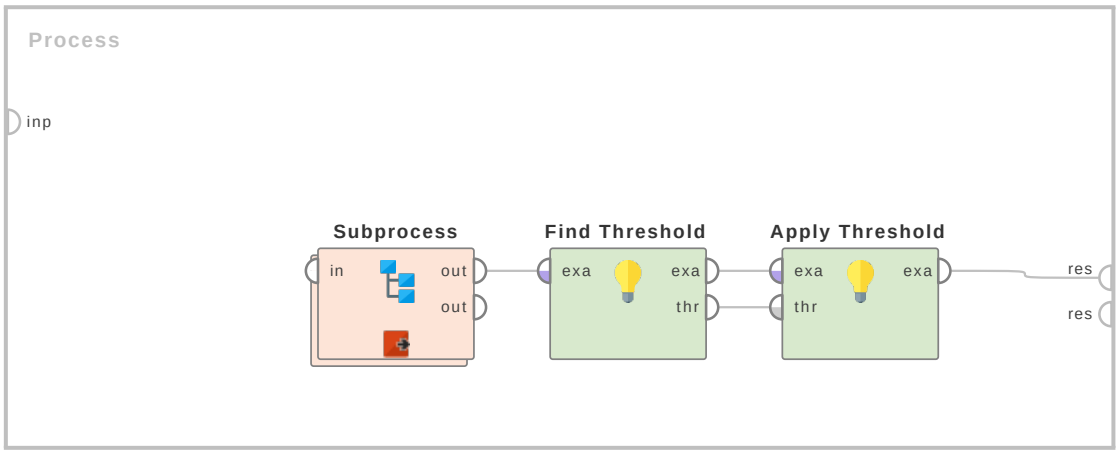
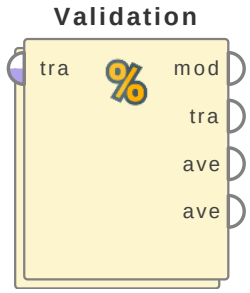



Figure 5.8: Tutorial process ‘Introduction to the Find Threshold operator’.

given threshold the prediction is set to this class otherwise it is set to the other class. Have a look at the resultant ExampleSet. Sort the ExampleSet according to ‘confidence(positive)’ and scroll through the ExampleSet. You will see that all examples where ‘confidence(positive)’ is greater than 0.306 are classified as positive and all examples where ‘confidence(positive)’ is less than or equal to 0.306 are classified as negative. In the original ExampleSet the boundary value was 0.500 but the Find Threshold operator found a better threshold for a crisp classification of soft classified data.

# 6Validation

## Bootstrapping Validation




This operator performs validation after bootstrapping a sampling of training data set in order to estimate the statistical performance of a learning operator (usually on unseen data sets). It is mainly used to estimate how accurately a model (learnt by a particular learning operator) will perform in practice.

### Description

The Bootstrapping Validation operator is a nested operator. It has two subprocesses: a training subprocess and a testing subprocess. The training subprocess is used for training a model. The trained model is then applied in the testing subprocess. The performance of the model is also measured during the testing phase. The training subprocess must provide a model and the testing subprocess must provide a performance vector.

The input ExampleSet is partitioned into two subsets. One subset is used as the training set and the other one is used as the test set. The size of two subsets can be adjusted through the *sample ratio* parameter. The *sample ratio* parameter specifies the ratio of examples to be used in the training set. The ratio of examples in the testing set is automatically calculated as *1-n* where *n* is the ratio of examples in the training set. The important thing to note here is that this operator performs bootstrapping sampling (explained in the next paragraph) on the training set before training a model. The model is learned on the training set and is then applied on the test set. This process is repeated *m* number of times where *m* is the value of the *number of validations* parameter.

Bootstrapping sampling is sampling with replacement. In sampling with replacement, at every step all examples have equal probability of being selected. Once an example has been selected for the sample, it remains candidate for selection and it can be selected again in any other coming steps. Thus a sample with replacement can have the same example multiple number of times. More importantly, a sample with replacement can be used to generate a sample that is greater in size than the original ExampleSet.

Usually the learning process optimizes the model parameters to make the model fit the training data as well as possible. If we then take an independent sample of testing data, it will generally turn out that the model does not fit the testing data as well as it fits the training data. This is called 'over-fitting', and is particularly likely to happen when the size of the training data set is small, or when the number of parameters in the model is large. Bootstrapping Validation is a way to predict the fit of a model to a hypothetical testing set when an explicit testing set is not available.

### Differentiation

- **Split Validation** Its validation subprocess executes just once. It provides linear, shuffled and stratified sampling. See page 749 for details.

- **Cross Validation** The input ExampleSet is partitioned into *k* subsets of equal size. Of the *k* subsets, a single subset is retained as the testing data set (i.e. input of the testing subprocess), and the remaining *k – 1* subsets are used as the training data set (i.e. input of the training subprocess). The cross-validation process is then repeated *k* times, with each of the *k* subsets used exactly once as the testing data. The *k* results from the *k* iterations can then be averaged (or otherwise combined) to produce a single estimation. See page 744 for details.

## Input Ports

**training *(tra)*** This input port expects an ExampleSet for training a model (training data set). The same ExampleSet will be used during the testing subprocess for testing the model.

## Output Ports

**model *(mod)*** The training subprocess must return a model, which is trained on the input ExampleSet. Please note that model built on the complete input ExampleSet is delivered from this port.

**training *(tra)*** The ExampleSet that was given as input at the *training* input port is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**averagable *(ave)*** The testing subprocess must return a Performance Vector. This is usually generated by applying the model and measuring its performance. Two such ports are provided but more can also be used if required. Please note that the statistical performance calculated by this estimation scheme is only an estimate (instead of an exact calculation) of the performance which would be achieved with the model built on the complete delivered data set.

## Parameters

**number of validations *(integer)*** This parameter specifies the number of times the validation should be repeated i.e. the number of times the inner subprocess should be executed.

**sample ratio *(real)*** This parameter specifies the relative size of the training set. In other validation schemes this parameter should be between 1 and 0, where 1 means that the entire ExampleSet will be used as training set. In this operator its value can be greater than 1 because bootstrapping sampling can generate an ExampleSet with a number of examples greater than the original ExampleSet. All examples that are not selected for the training set are automatically selected for the test set.

**use weights *(boolean)*** If this parameter is checked, example weights will be used for bootstrapping if such weights are available.

**average performances only *(boolean)*** This parameter indicates if only performance vectors should be averaged or all types of averagable result vectors.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomizing examples of a subset. Using the same value of the *local random seed* will produce the same samples. Changing the value of this parameter changes the way examples are randomized, thus samples will have a different set of examples.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Related Documents

- **Split Validation** (page 749)
- **Cross Validation** (page 744)

## Tutorial Processes

### Validating Models using Bootstrapping Validation

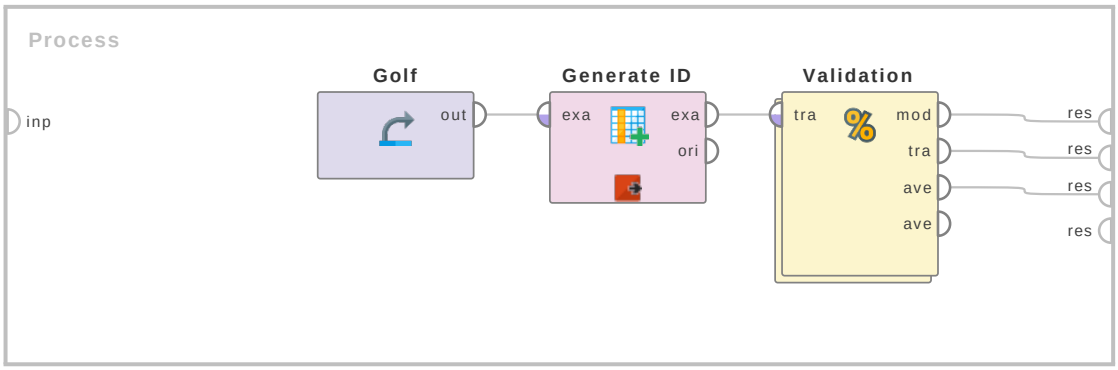
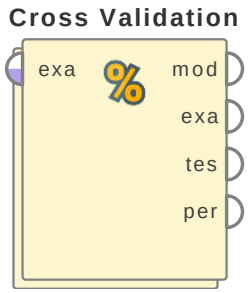



Figure 6.1: Tutorial process 'Validating Models using Bootstrapping Validation'.

The 'Golf' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it to uniquely identify examples. This is done so that you can understand this process easily; otherwise IDs are not required here. A breakpoint is added after this operator so that you can preview the data before the application of the Bootstrapping Validation operator. You can see that the ExampleSet has 14 examples with ids from 1 to 14. Double click the Bootstrapping Validation operator and you will see the training and testing subprocesses. The Decision Tree operator is used in the training subprocess. The trained model (i.e. Decision Tree) is passed to the testing subprocess through the model ports. The testing subprocess receives testing data from the testing port.

Now, have a look at the parameters of the Bootstrapping Validation operator. The no of validations parameter is set to 2 thus the inner subprocess will execute just twice. The sample ratio parameter is set to 0.5. The number of examples in the ExampleSet is 14 and sample ratio is 0.5, thus the training set will be composed of 7 (i.e. 14 x 0.5) examples. But it is not necessary that these examples will be unique because bootstrapping sampling can select an example multiple number of time. All the examples that are not selected for the training set automatically become part of the testing set. You can verify this by running the process. You will see that the training set has 7 examples but they are not all unique and all the examples that were not part of the training set are part of the testing set.

# Cross Validation




This Operator performs a cross validation to estimate the statistical performance of a learning model.

## Description

It is mainly used to estimate how accurately a model (learned by a particular learning Operator) will perform in practice.

The Cross Validation Operator is a nested Operator. It has two subprocesses: a Training subprocess and a Testing subprocess. The Training subprocess is used for training a model. The trained model is then applied in the Testing subprocess. The performance of the model is measured during the Testing phase.

The input ExampleSet is partitioned into *k* subsets of equal size. Of the *k* subsets, a single subset is retained as the test data set (i.e. input of the Testing subprocess). The remaining *k - 1* subsets are used as training data set (i.e. input of the Training subprocess). The cross validation process is then repeated *k* times, with each of the *k* subsets used exactly once as the test data. The *k* results from the *k* iterations are averaged (or otherwise combined) to produce a single estimation. The value *k* can be adjusted using the *number of folds* parameter.

The evaluation of the performance of a model on independent test sets yields a good estimation of the performance on unseen data sets. It also shows if 'overfitting' occurs. This means that the model represents the testing data very well, but it does not generalize well for new data. Thus, the performance can be much worse on test data.

## Differentiation

- **Split Validation**

  This Operator is similar to the Cross Validation Operator but only splits the data into one training and one test set. Hence it is similar to one iteration of the cross validation.

  See page 749 for details.

- **Split Data**

  This Operator splits an ExampleSet into different subsets. It can be used to manual perform a validation.

  See page 254 for details.

- **Bootstrapping Validation**

  This Operator is similar to the Cross Validation Operator. Instead of splitting the input ExampleSet into different subset, the Bootstrapping Validation Operator uses bootstrapping sampling to get the training data. Bootstrapping sampling is sampling with replacement.

  See page 741 for details.

- **Wrapper Split Validation**

  This Operator is similar to the Split Validation Operator. It has an additional Attribute Weighting subprocess to evaluate the attribute weighting method individually.

  See page 754 for details.

- **Wrapper-X-Validation**

  This Operator is similar to the Cross Validation Operator. It has an additional Attribute Weighting subprocess to evaluate the attribute weighting method individually.

  See page 756 for details.

## Input Ports

**example set *(exa)*** This input port receives an ExampleSet to apply the cross validation.

## Output Ports

**model *(mod)*** This port delivers the prediction model trained on the whole ExampleSet. Please note that this port should only be connected if you really need this model because otherwise the generation will be skipped.

**performance *(per)*** This is an expandable port. You can connect any performance vector (result of a Performance Operator) to the result port of the inner Testing subprocess. The performance output ports of the Cross Validation Operator deliver the average of the performances over the *number of folds* iterations.

**example set *(exa)*** This port returns the same ExampleSet which as been given as input.

**test result set *(tes)*** This port delivers only an ExampleSet if the test set results port of the inner Testing subprocess is connected. If so, the test sets are merged to one ExampleSet and delivered by this port. For example with this output port it is possible to get the labeled test sets, with the results of the Apply Model Operator.

## Parameters

**split on batch attribute** If this parameter is enabled, use the Attribute with the special role 'batch' to partition the data instead of randomly splitting the data. This gives you control over the exact Examples which are used to train the model in each fold. All other split parameters are not available in this case.

**leave one out** If this parameter is enabled, the test set (i.e. the input of the Testing subprocess) is only one Example from the original ExampleSet. The remaining Examples are used as the training data. This is repeated such that each Example in the ExampleSet is used once as the test data. Thus it is repeated 'n' times, where 'n' is the total number of Examples in the ExampleSet. The Cross Validation can take a very long time, as the Training and Testing subprocesses are repeated as many times as the number of Example. If set to true, the *number of folds* parameter is not available.

**number of folds** This parameter specifies the number of folds (number of subsets) the ExampleSet should be divided into. Each subset has equal number of Examples. Also the number of iterations that will take place is the same as the *number of folds*. If the model output port is connected, the Training subprocess is repeated one more time with all Examples to build the final model.

**sampling type** The Cross Validation Operator can use several types of sampling for building the subsets. Following options are available:

- **linear_sampling** The linear sampling divides the ExampleSet into partitions without changing the order of the Examples. Subsets with consecutive Examples are created.
- **shuffled_sampling** The shuffled sampling builds random subsets of the ExampleSet. Examples are chosen randomly for making subsets.
- **stratified_sampling** The stratified sampling builds random subsets. It ensures that the class distribution (defined by the label Attribute) in the subsets is the same as in the whole ExampleSet. For example in the case of a binominal classification, stratified sampling builds random subsets such that each subset contains roughly the same proportions of the two values of the label Attribute.
- **automatic** The automated mode uses stratified sampling per default. If it isn't applicable e.g. if the ExampleSet doesn't contain a nominal label, shuffled sampling will be used instead.

**use local random seed** This parameter indicates if a *local random seed* should be used for randomizing Examples of a subset. Using the same value of the *local random seed* will produce the same subsets. Changing the value of this parameter changes the way Examples are randomized, thus subsets will have a different set of Examples. This parameter is available only if shuffled or stratified sampling is selected. It is not available for linear sampling because it requires no randomization, Examples are selected in sequence.

**local random seed** If the *use local random seed* parameter is checked this parameter determines the local random seed. The same subsets will be created every time if the same value is used.

**enable parallel execution** This parameter enables the parallel execution of the inner processes. Please disable the parallel execution if you run into memory problems.

## Tutorial Processes

### Why validate Models

This tutorial process shows the reason why you always have to validate a learning model on an independent data set.

The 'Sonar' data set is retrieved from the Samples folder. The Split Data Operator splits it into two different subsets (with 90 % and 10 % of the Examples). A decision tree is trained on the larger data set (which is called training data).

The decision tree is applied on both the training data and the test data and the performance is calculated for both. Below that a Cross Validation Operator is used to calculate the performance of a decision tree on the Sonar data in a more sophisticated way.

All calculated performances are delivered to the result ports of the Process:

Performance on Training data: The accuracy is relatively high with 86.63 % Performance on Test data: The accuracy is only 61.90 %. This shows that the decision tree is trained to fit the Training data well, but perform worse on the test data. This effect is called 'overfitting'. Performance from Cross Validation: The accuracy is 62.12 % +/- 9.81%. The Cross Validation not only gives us a good estimation of the performance of the model on unseen data, but also the standard deviation of this estimation. The above mentioned Perfomance on Test data falls inside this estimation, whereas the performance on the Training data is above it and is effected by 'overfitting'.

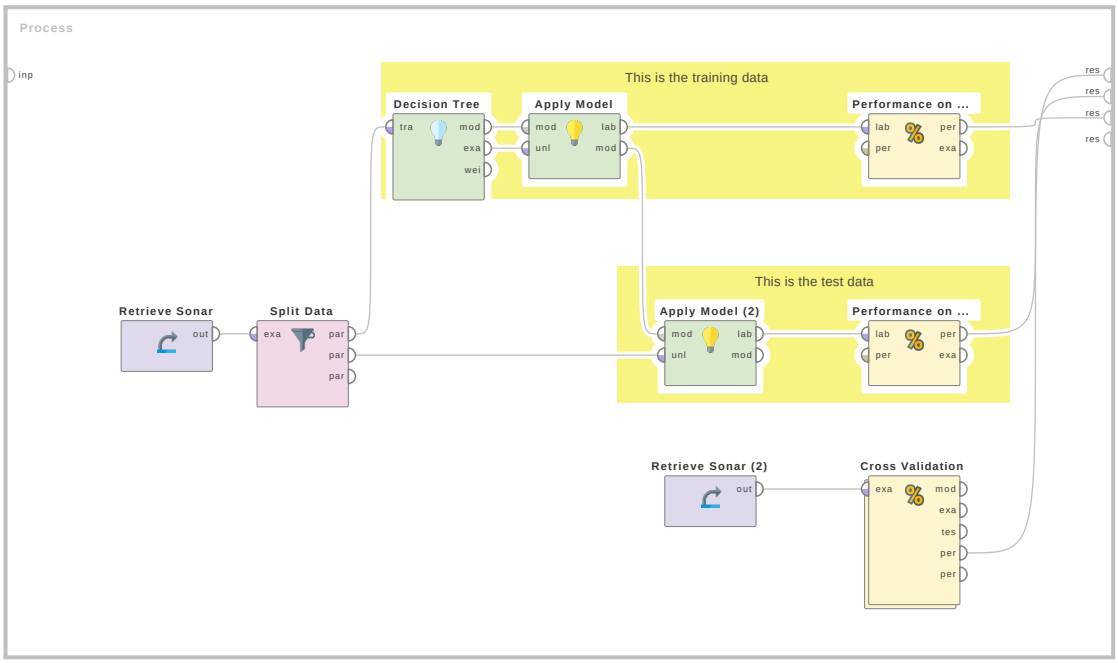



Figure 6.2: Tutorial process ‘Why validate Models’.

### Validating Models using Cross Validation

This tutorial process shows the basic usage of the Cross Validation Operator on the ‘Deals’ data set from the Sample folder.

The Cross Validation Operator divides the ExampleSet into 3 subsets. The sampling type parameter is set to linear sampling, so the subsets will have consecutive Examples (check the ID Attribute). A decision tree is trained on 2 of the 3 subsets inside the Training subprocess of the Cross Validation Operator.

The performance of the decision tree is then calculated on the remaining subset in the Testing subprocess.

This is repeated 3 times, so that each subset was used one time as a test set.

The calculated performances are averaged over the three iterations and delivered to the result port of the Process. Also the decision tree, which was trained on all Examples, is delivered to the result port. The merged test sets (the test result set output port of the Cross Validation Operator) is the third result of the Process.

Play around with the parameters of the Cross Validation Operator. The number of folds parameter controls the number of subsets, the input ExampleSet is divided into. Hence it is also the number of iterations of the cross validation. The sampling type changes the way the subsets are created.

If linear sampling is used the IDs of the Examples in the subsets will be consecutive values. If shuffled sampling is used the IDs of the Examples in the subsets will be randomized. If stratified sampling is used the IDs of the Examples are also randomized, but the class distribution in the subsets will be nearly the same as in the whole ‘Deals’ data set.

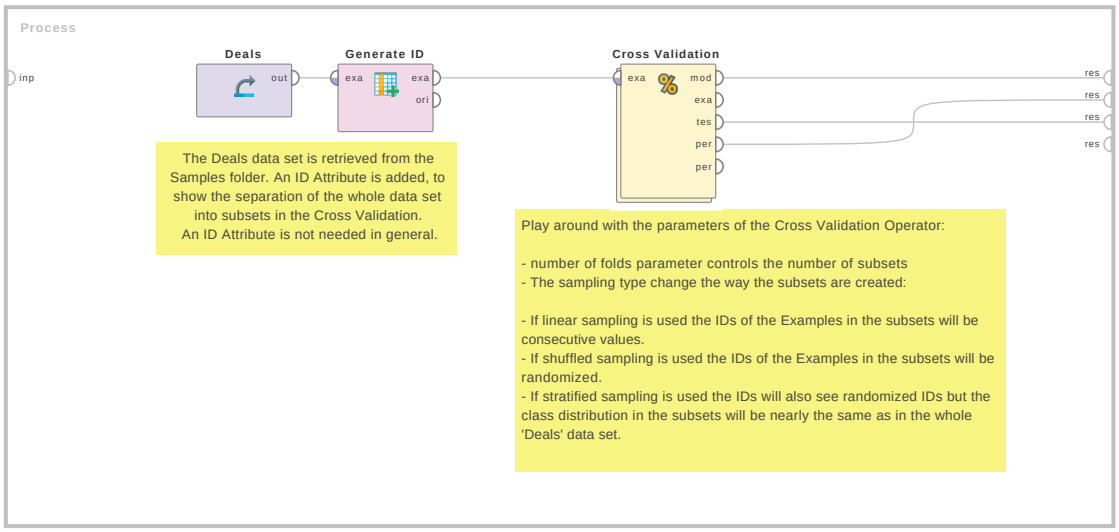



Figure 6.3: Tutorial process ‘Validating Models using Cross Validation’.

### Passing results from Training to Testing subprocess using through ports

This Process shows the usage of the through port to pass through RapidMiner Objects from the Training to the Testing subprocess of the Cross Validation Operator.

In this Process an Attribute selection is performed before a linear regression is trained. The Attribute weights are passed to the Testing subprocess. Also two different Performance Operators are used to calculate the performance of the model. Their results are connected to the expandable performance port of the Testing subprocess.

Both performances are averaged over the 10 iterations of the cross validation and are delivered to the result ports of the Process.

### Using the batch Attribute to split the training data

This Process shows the usage of the split on batch attribute parameter of the Cross Validation Operator.

The Titanic Training data set is retrieved from the Samples folder and the Passenger Class Attribute is set to ‘batch’ role. As the split on batch attribute parameter of the Cross Validation Operator is set to true, the data set is splitted into three subsets. Each subset has only Examples of one Passenger class.

In the Training subprocess, 2 of the subsets are used to train the decision tree. In the Testing subprocess, the remaining subset is used to test the decision tree.

Thus the decision tree is trained on all passengers from two Passenger Classes and tested on the remaining class. The performances of all three combinations are averaged and delivered to the result port of the Process.

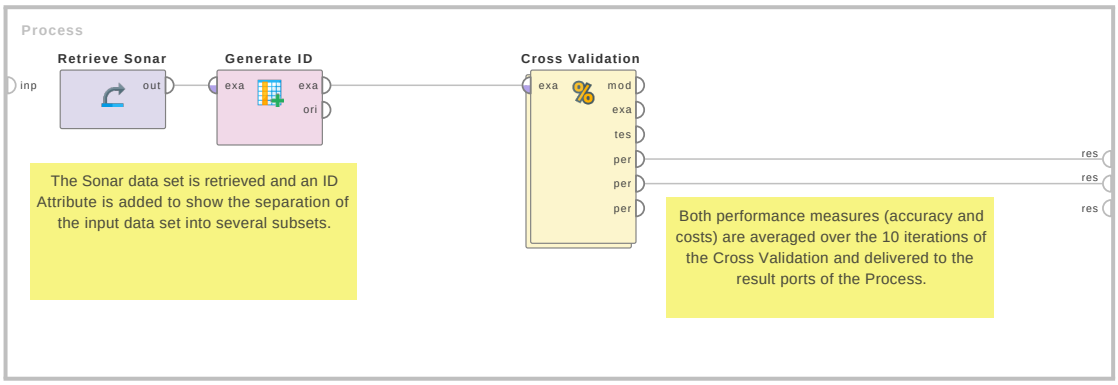



Figure 6.4: Tutorial process ‘Passing results from Training to Testing subprocess using through ports’.

## Split Validation


This operator performs a simple validation i.e. randomly splits up the ExampleSet into a training set and test set and evaluates the model. This operator performs a split validation in order to estimate the performance of a learning operator (usually on unseen data sets). It is mainly used to estimate how accurately a model (learnt by a particular learning operator) will perform in practice.

### Description

The Split Validation operator is a nested operator. It has two subprocesses: a training subprocess and a testing subprocess. The training subprocess is used for learning or building a model. The trained model is then applied in the testing subprocess. The performance of the model is also measured during the testing phase.

The input ExampleSet is partitioned into two subsets. One subset is used as the training set and the other one is used as the test set. The size of two subsets can be adjusted through different parameters. The model is learned on the training set and is then applied on the test set. This is done in a single iteration, as compared to the Cross Validation operator that iterates a number of times using different subsets for testing and training purposes.

Usually the learning process optimizes the model parameters to make the model fit the training data as well as possible. If we then take an independent sample of testing data, it will generally turn out that the model does not fit the testing data as well as it fits the training data. This is called ‘over-fitting’, and is particularly likely to happen when the size of the training data set is small, or when the number of parameters in the model is large. Split Validation is a way to predict the fit of a model to a hypothetical testing set when an explicit testing set is not available. The Split Validation operator also allows training on one data set and testing on another explicit testing data set.

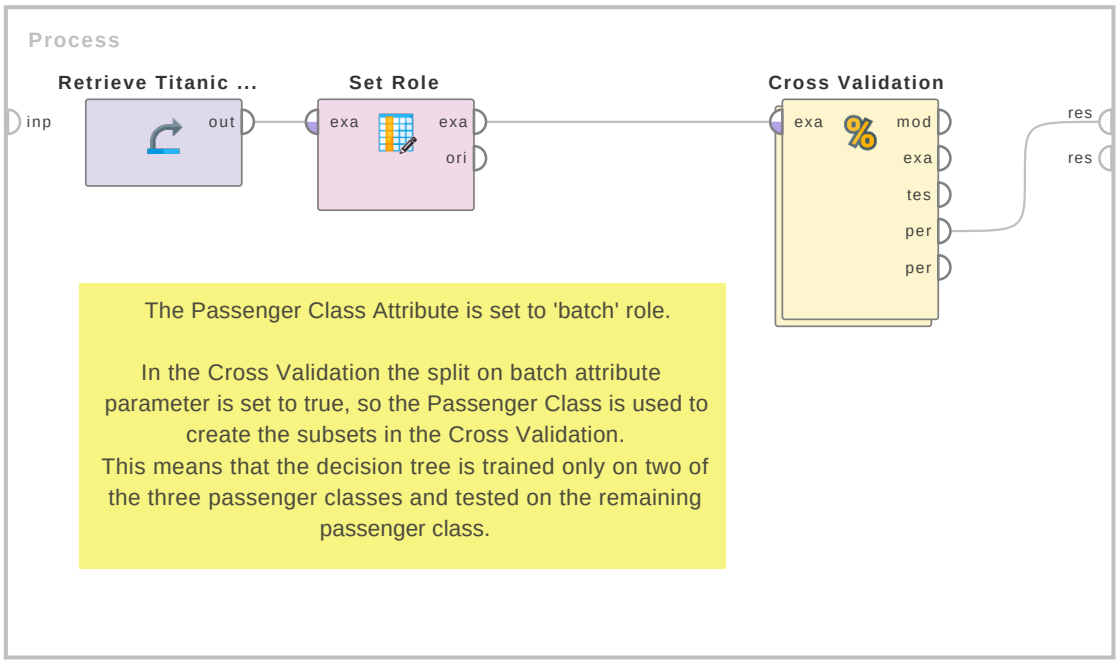



Figure 6.5: Tutorial process 'Using the batch Attribute to split the training data'.

## Input Ports

**training example set *(tra)*** This input port expects an ExampleSet for training a model (training data set). The same ExampleSet will be used during the testing subprocess for testing the model if no other data set is provided.

## Output Ports

**model *(mod)*** The training subprocess must return a model, which is trained on the input ExampleSet. Please note that the model built on the complete input ExampleSet is delivered from this port.

**training example set *(tra)*** The ExampleSet that was given as input at the*training* input port is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**averagable *(ave)*** The testing subprocess must return a Performance Vector. This is usually generated by applying the model and measuring its performance. Two such ports are provided but more can also be used if required. Please note that the performance calculated by this estimation scheme is only an estimate (instead of an exact calculation) of the performance which would be achieved with the model built on the complete delivered data set.

## Parameters

**split *(selection)*** This parameter specifies how the ExampleSet should be split

- **relative** If a relative split is required, the relative size of the training set should be provided in the *split ratio* parameter. Afterwards the relative size of the test set is automatically calculated by subtracting the value of the *split ratio* from 1.
- **absolute** If an absolute split is required, you have to specify the exact number of examples to use in the training or test set in the *training set size* parameter or in the *test set size* parameter. If either of these parameters is set to -1, its value is calculated automatically using the other one.

**split ratio *(real)*** This parameter is only available when the *split* parameter is set to 'relative'. It specifies the relative size of the training set. It should be between 1 and 0, where 1 means that the entire ExampleSet will be used as training set.

**training set size *(integer)*** This parameter is only available when the *split* parameter is set to 'absolute'. It specifies the exact number of examples to be used as training set. If it is set to -1, the *test size set* number of examples will be used for the test set and the remaining examples will be used as training set.

**test set size *(integer)*** This parameter is only available when the *split* parameter is set to 'absolute'. It specifies the exact number of examples to be used as test set. If it is set to -1, the *training size set* number of examples will be used for training set and the remaining examples will be used as test set.

**sampling type *(selection)*** The Split Validation operator can use several types of sampling for building the subsets. Following options are available:

- **linear_sampling** The linear sampling simply divides the ExampleSet into partitions without changing the order of the examples i.e. subsets with consecutive examples are created.
- **shuffled_sampling** The shuffled sampling builds random subsets of the ExampleSet. Examples are chosen randomly for making subsets.
- **stratified_sampling** The stratified sampling builds random subsets and ensures that the class distribution in the subsets is the same as in the whole ExampleSet. For example, in the case of a binominal classification, stratified sampling builds random subsets such that each subset contains roughly the same proportions of the two values of class *labels*.
- **automatic** The automated mode uses stratified sampling per default. If it isn't applicable, e.g., if the ExampleSet doesn't contain a nominal label, shuffled sampling will be used instead.

**use local random seed *(boolean)*** Indicates if a *local random seed*should be used for randomizing examples of a subset. Using the same value of*local random seed* will produce the same subsets. Changing the value of this parameter changes the way examples are randomized, thus subsets will have a different set of examples. This parameter is only available if Shuffled or Stratified sampling is selected. It is not available for Linear sampling because it requires no randomization, examples are selected in sequence.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

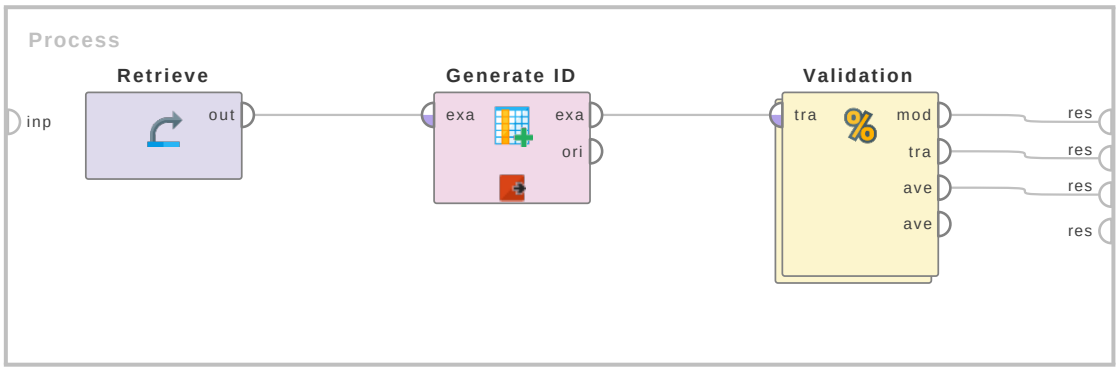



Figure 6.6: Tutorial process 'Validating Models using Split Validation'.

### Validating Models using Split Validation

The 'Golf' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it to uniquely identify examples. This is done so that you can understand this process easily; otherwise IDs are not required here. A breakpoint is added after this operator so that you can preview the data before the Split Validation operator starts. Double click the Split Validation operator and you will see training and testing subprocesses. The Decision Tree operator is used in the training subprocess. The trained model (i.e. Decision Tree) is passed to the testing subprocess through the model ports. The testing subprocess receives testing data from the testing port.

Now, have a look at the parameters of the Split Validation operator. The split parameter is set to 'absolute'. The training set size parameter is set to 10 and the test set size parameter is set to -1. As there are 14 total examples in the 'Golf' data set, the test set automatically gets 4 remaining examples. The sampling type parameter is set to Linear Sampling. Remaining parameters have default values. Thus two subsets of the 'Golf' data set will be created. You will observe later that these two subsets are created: training set: examples with IDs 1 to 10 (10 examples)test set: examples with IDs 11 to 14 (4 examples)

You can see that all examples in a subset are consecutive (i.e. with consecutive IDs). This is because Linear Sampling is used.

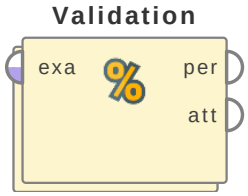
Breakpoints are inserted to make you understand the process. Here is what happens when you run the process: First the 'Golf' data set is displayed with all rows uniquely identified using the ID attribute. There are 14 rows with ids 1 to 14. Press the green-colored Run button to continue.Now a Decision tree is shown. This was trained from the training set of the 'Golf' data set. Hit the Run button to continue.The Decision tree was applied on the testing data. Here you can see the results after application of the Decision Tree model. Have a look at IDs of the testing data here. They are 11 to 14. Compare the label and prediction columns and you will see that only 2 predictions out of 4 are correct (only ID 1 and 3 are correct predictions). Hit the Run button again.Now the Performance Vector of the Decision tree is shown. As only 2 out of 4 predictions were correct, the accuracy is 50%. Press the Run button again.Now you can see a different Decision tree. It was trained on the complete 'Golf' data set that is why it is different from the previous decision tree.

You can run the same process with different values of sampling type parameter. If linear sampling is used, as in our example process, you will see that IDs of examples in subsets will be consecutive values. If shuffled sampling is used you will see that IDs of examples in subsets will be random values. If stratified sampling is used you will see that IDs of examples in subsets will be

random values but the class distribution in the subsets will be nearly the same as in the whole 'Golf' data set.

To get an understanding of how objects are passed using through ports please study the Example Process of Cross Validation operator.

# Wrapper Split Validation




A simple validation method to check the performance of a feature weighting or selection wrapper.

## Description

This operator evaluates the performance of feature weighting algorithms including feature selection. The first inner operator is the weighting algorithm to be evaluated itself. It must return an attribute weights vector which is applied on the data. Then a new model is created using the second inner operator and a performance is retrieved using the third inner operator. This performance vector serves as a performance indicator for the actual algorithm. This implementation is described for the *RandomSplitValidationChain*.

## Input Ports

**example set in *(exa)*** This input port expects an ExampleSet. Subsets of this ExampleSet will be used as training and testing data sets.

## Output Ports

**performance vector out *(per)*** The Model Evaluation subprocess must return a Performance Vector in each iteration. This is usually generated by applying the model and measuring its performance. Please note that the statistical performance calculated by this estimation scheme is only an estimate (instead of an exact calculation) of the performance which would be achieved with the model built on the complete delivered data set.

**attribute weights out *(att)*** The Attribute Weighting subprocess must return an attribute weights vector in each iteration. Please note that the attribute weights vector built on the complete input ExampleSet is delivered from this port.

## Parameters

**split ratio** Relative size of the training set.

**sampling type** The Wrapper Split Validation operator can use several types of sampling for building the subsets. Following options are available:

- **linear_sampling** The linear sampling simply divides the ExampleSet into partitions without changing the order of the examples i.e. subsets with consecutive examples are created.
- **shuffled_sampling** The shuffled sampling builds random subsets of the ExampleSet. Examples are chosen randomly for making subsets.
- **stratified_sampling** The stratified sampling builds random subsets and ensures that the class distribution in the subsets is the same as in the whole ExampleSet. For example, in the case of a binominal classification, stratified sampling builds random

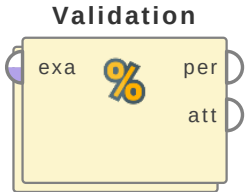
subsets such that each subset contains roughly the same proportions of the two values of class *labels*.

- **automatic** The automated mode uses stratified sampling per default. If it isn't applicable, e.g., if the ExampleSet doesn't contain a nominal label, shuffled sampling will be used instead.

**use local random seed** This parameter indicates if a *local random seed* should be used for randomizing examples of a subset. Using the same value of the *local random seed* will produce the same subsets. Changing the value of this parameter changes the way examples are randomized, thus subsets will have a different set of examples. This parameter is available only if shuffled, stratified or automatic sampling is selected. It is not available for linear sampling because it requires no randomization, examples are selected in sequence.

**local random seed** This parameter specifies the *local random seed*. This parameter is available only if the *use local random seed* parameter is set to true.

# Wrapper-X-Validation




This operator performs a cross-validation in order to evaluate the performance of a feature weighting or selection scheme. It is mainly used for estimating how accurately a scheme will perform in practice.

## Description

The Wrapper-X-Validation operator is a nested operator. It has three subprocesses: an Attribute Weighting subprocess, a Model Building subprocess and a Model Evaluation subprocess. The Attribute Weighting subprocess contains the algorithm to be evaluated. It must return an attribute weights vector which is then applied on the training data set. The Model Building subprocess is used for training a new model in each iteration. This model is trained on the same training data set that was used in the first subprocess. But the training data set for this subprocess does not contain those attributes that had weight 0 in the weights vector of the first subprocess. The trained model is then applied and evaluated in the Model Evaluation subprocess. The model is tested on the testing data set. This subprocess must return a performance vector. This performance vector serves as a performance indicator of the actual algorithm.

The input ExampleSet is partitioned into $k$ subsets of equal size. Of the $k$ subsets, a single subset is retained as the testing data set (i.e. input of the third subprocess), and the remaining $k - 1$ subsets are used as training data set (i.e. input of the first two subprocesses). The cross-validation process is then repeated $k$ times, with each of the $k$ subsets used exactly once as the testing data. The $k$ results from the $k$ iterations then can be averaged (or otherwise combined) to produce a single estimation. The value $k$ can be adjusted using the *number of validations* parameter. Please study the attached Example Process for more information.

Just as for learning, it is also possible that overfitting occurs during preprocessing. In order to estimate the generalization performance of a preprocessing method RapidMiner supports several validation operators for preprocessing steps. The basic idea is the same as for all other validation operators with a slight difference: the first inner operator must produce a transformed example set, the second must produce a model from this transformed data set and the third operator must produce a performance vector of this model on a test set transformed in the same way.

## Input Ports

**example set in *(exa)*** This input port expects an ExampleSet. Subsets of this ExampleSet will be used as training and testing data sets.

## Output Ports

**performance vector out *(per)*** The Model Evaluation subprocess must return a Performance Vector in each iteration. This is usually generated by applying the model and measuring its performance. Two such ports are provided but more can also be used if required. Please note that the statistical performance calculated by this estimation scheme is only an estimate (instead of an exact calculation) of the performance which would be achieved with the model built on the complete delivered data set.

**attribute weights out *(att)*** The Attribute Weighting subprocess must return an attribute weights vector in each iteration. Please note that the attribute weights vector built on the complete input ExampleSet is delivered from this port.

## Parameters

**leave one out *(boolean)*** As the name suggests, the *leave one out* cross-validation involves using a single example from the original ExampleSet as the testing data, and the remaining examples as the training data. This is repeated such that each example in the ExampleSet is used once as the testing data. Thus, it is repeated 'n' number of times, where 'n' is the total number of examples in the ExampleSet. This is the same as applying the Batch-X-Validation operator with the *number of validations* parameter set equal to the number of examples in the original ExampleSet. This is usually very expensive for large ExampleSets from a computational point of view because the training process is repeated a large number of times (number of examples time). If set to true, the *number of validations* parameter is ignored.

**number of validations *(integer)*** This parameter specifies the number of subsets the ExampleSet should be divided into (each subset has an equal number of examples). Also the same number of iterations will take place. If this is set equal to the total number of examples in the ExampleSet, it is equivalent to the Batch-X-Validation operator with the *leave one out* parameter set to true.

**sampling type *(selection)*** The Batch-X-Validation operator can use several types of sampling for building the subsets. Following options are available:

- **linear_sampling** The linear sampling simply divides the ExampleSet into partitions without changing the order of the examples i.e. subsets with consecutive examples are created.
- **shuffled_sampling** The shuffled sampling builds random subsets of the ExampleSet. Examples are chosen randomly for making subsets.
- **stratified_sampling** The stratified sampling builds random subsets and ensures that the class distribution in the subsets is the same as in the whole ExampleSet. For example, in the case of a binominal classification, stratified sampling builds random subsets such that each subset contains roughly the same proportions of the two values of class *labels*.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomizing examples of a subset. Using the same value of the *local random seed* will produce the same subsets. Changing the value of this parameter changes the way examples are randomized, thus subsets will have a different set of examples. This parameter is available only if Shuffled or Stratified sampling is selected. It is not available for Linear sampling because it requires no randomization, examples are selected in sequence.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is available only if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Evaluating an attribute selection scheme

This Example Process starts with the Subprocess operator which provides an ExampleSet. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that there

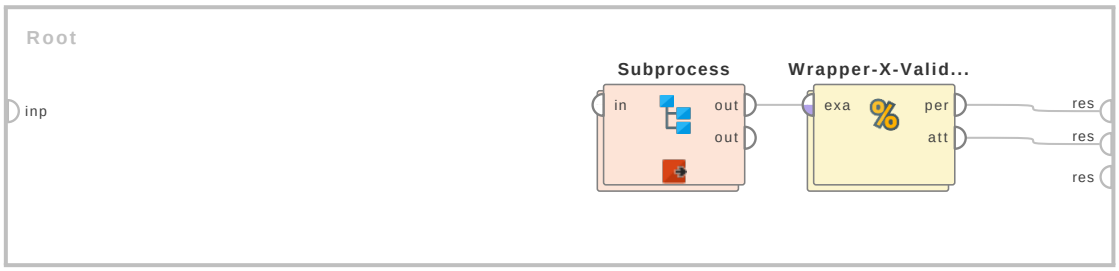



Figure 6.7: Tutorial process 'Evaluating an attribute selection scheme'.

are 60 examples, uniquely identified by the id attribute. There are 6 attributes in the ExampleSet. The Wrapper-X-Validation operator is applied on this ExampleSet for evaluating an attribute selection scheme. The scheme to be evaluated is placed in the Attribute Weighting subprocess of the Wrapper-X-Validation operator. The Optimize Selection operator is used in this Example Process. Its subprocess is not discussed here for the sake of simplicity.

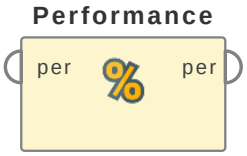
Have a look at the parameters of the Wrapper-X-Validation operator. The number of validations parameter is set to 6 and the sampling type parameter is set to 'linear sampling'. Thus the given ExampleSet will be broken into 6 subsets linearly (i.e. each subset will have consecutive examples). The Wrapper-X-Validation operator will have 6 iterations. In every iteration 5 out of 6 subsets will serve as the training data set and the remaining subset will serve as the testing subset.

The following steps are followed in every iteration: The Attribute Weighting subprocess trains an attribute selection scheme using the training data set. The Model Building subprocess receives the training data set but with only those attributes that had non-zero weight in the resultant weights vector of the first subprocess. A model is trained using this data set.The Model Evaluation subprocess tests this model on the testing data set and delivers a performance vector.

Breakpoints are inserted at the following places in the process: Before the Attribute Weighting subprocess so that you can see the training data set of the iteration.After the Attribute Weighting subprocess so that you can see the attribute weights vector.Before the Model Building subprocess so that you can see the training data set (without attributes that had 0 weight) that will be used for training the model.Before the Model Evaluation subprocess so that you can see the testing data set of the iteration.

# 6.1 Performance

## Combine Performances




This operator takes a performance vector as input and returns a performance vector containing the weighted fitness value of the specified criteria.

### Description

This Combine Performances operator takes a performance vector as input and returns a performance vector containing the weighted fitness value of the specified criteria. The user can specify the weights of different criteria. This operator takes the weighted average of the values of the specified criteria. It should be noted that some criteria values are considered positive by this operator e.g. accuracy. On the other hand some criteria values (usually error related) are considered negative by this operator e.g. relative error. Please study the attached Example Process for better understanding of this operator.

### Input Ports

**performance *(per)*** This port expects a performance vector. A performance vector is a list of performance criteria values.

### Output Ports

**performance *(per)*** The performance vector containing the weighted fitness value of the specified criteria is returned through this port.

### Parameters

**default weight *(real)*** This parameter specifies the default weight for all criteria that are not assigned a weight through the *criteria weights* parameter.

**criteria weights *(list)*** Different performance criteria can be assigned different weights through this parameter. The criteria that are not assigned a weight through this parameter will have the default weight (i.e. specified by the *default weight* parameter).

### Tutorial Processes

#### Introduction to the Combine Performances operator

This Example Process starts with the Subprocess operator. The subprocess is used for generating a sample performance vector. Therefore it is not necessary to understand the operators in the subprocess. A breakpoint is inserted after the Subprocess operator so that you can have a look at the performance vector. The performance vector has the following criteria values:
Accuracy: 0.250Absolute error: 0.750Root mean squared error: 0.866 It is important to note that the accuracy is considered positive and the remaining two criteria are considered negative in the calculations by the Combine Performances operator.

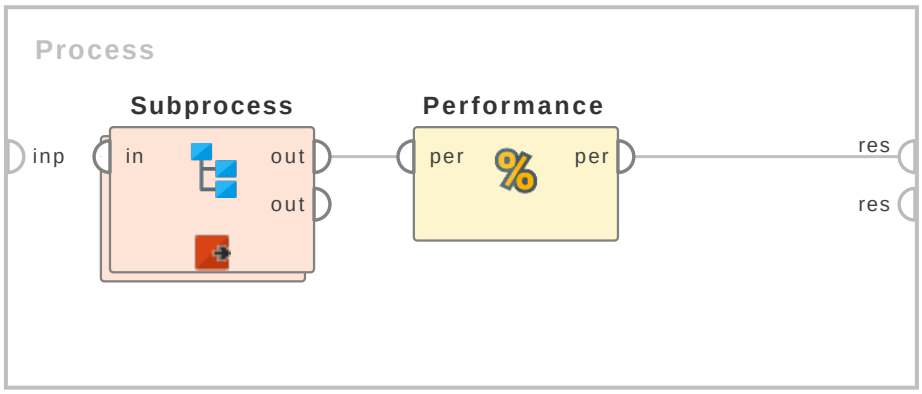
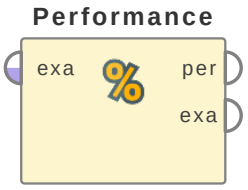



Figure 6.8: Tutorial process ‘Introduction to the Combine Performances operator’.

The Combine Performances operator is applied on this performance vector. Have a look at the criteria weights parameter of the Combine Performances operator. The following weights are assigned to criteria: Accuracy: 2.0Absolute error: 1.0Root mean squared error: 0.0 The weighted fitness value is calculated by multiplying the weight with the corresponding value and finally averaging the results. In this case the following calculation is performed: (2(0.250) + 1(-0.750) + 0(0.866)) / 3= (0.500 - 0.750 + 0.000) / 3= -0.083

# Extract Performance




This operator can be used for deriving a performance measure (in form of a performance vector) from the given ExampleSet.

## Description

This operator can be used for generating a performance vector from the properties of the given ExampleSet. This includes properties like the number of examples or number of attributes of the input ExampleSet. Specific data value of the input ExampleSet can also be used as the value of the performance vector. Various statistical properties of the input ExampleSet e.g. average, min or max value of an attribute can also be used as the value of the performance vector. All these options can be understood by studying the parameters and the attached Example Process.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. The performance vector value will be extracted from this ExampleSet.

## Output Ports

**performance *(per)*** This port delivers a performance vector. A performance vector is a list of performance criteria values.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators.

## Parameters

**performance type *(selection)*** This parameter indicates the way the input ExampleSet should be used to define the performance vector.

- **number_of_examples** If this option is selected, the performance vector value is set to the total number of examples in the input ExampleSet.
- **number_of_attributes** If this option is selected, the performance vector value is set to the total number of attributes in the input ExampleSet.
- **data_value** If this option is selected, the performance vector value is set to the value of the specified attribute at the specified index. The attribute is specified using the *attribute name* parameter and the index is specified using the *example index* parameter.
- **statistics** If this option is selected, the performance vector value is set to the value obtained by applying the selected statistical operation on the specified attribute. The attribute is specified using the *attribute name* parameter and the statistical operation is selected using the *statistics* parameter.

**statistics *(selection)*** This parameter is only available when the *performance type* parameter is set to ‘statistics’. This parameter allows you to select the statistical operation to be applied on the attribute specified by the *attribute name* parameter.

**attribute name *(string)*** This parameter is only available when the *performance type* parameter is set to ‘statistics’ or ‘data value’. This parameter allows you to select the required attribute.

**attribute value *(string)*** This parameter is only available when the *performance type* parameter is set to ‘statistics’ and the *statistics* parameter is set to ‘count’. This parameter is used for specifying a particular value of the specified attribute. The performance vector value will be set to the number of occurrences of this value in the specified attribute. The attribute is specified by the *attribute name* parameter.

**example index *(integer)*** This parameter is only available when the *performance type* parameter is set to ‘data value’. This parameter allows you to select the index of the required example of the attribute specified by the *attribute name* parameter.

**optimization direction *(selection)*** This parameter indicates if the performance value should be minimized or maximized.

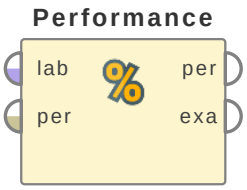
## Tutorial Processes

### Introduction to the Extract Performance operator


Figure 6.9: Tutorial process ‘Introduction to the Extract Performance operator’.

This is a very basic process that demonstrates the use of the Extract Performance operator. The ‘Golf’ data set is loaded using the Retrieve operator. The Extract Performance operator is applied on it. The performance type parameter is set to ‘statistics’, the statistics parameter is set to ‘average’ and the attribute name parameter is set to ‘Temperature’. Thus the value of the resultant performance vector will be the average of values of the Temperature attribute. The average of the Temperature attribute in all 14 examples of the ‘Golf’ data set is 73.571. The resultant performance vector and the ‘Golf’ data set can be seen in the Results Workspace. You can see that the value of the performance vector is 73.571.

# Performance




This operator is used for performance evaluation. It delivers a list of performance criteria values. These performance criteria are automatically determined in order to fit the learning task type.

## Description

In contrast to the other performance evaluation operators like the Performance (Classification) operator, the Performance (Binominal Classification) operator or the Performance (Regression) operator, this operator can be used for all types of learning tasks. It automatically determines the learning task type and calculates the most common criteria for that type. For more sophisticated performance calculations, you should use the operators mentioned above. If none of them meets your requirements, you can use Performance (User-Based) operator which allows you to write your own performance measure.

The following criteria are added for binominal classification tasks:

- Accuracy
- Precision
- Recall
- AUC (optimistic)
- AUC (neutral)
- AUC (pessimistic)

The following criteria are added for polynominal classification tasks:

- Accuracy
- Kappa statistic

The following criteria are added for regression tasks:

- Root Mean Squared Error
- Mean Squared Error

## Input Ports

**labelled data *(lab)*** This input port expects a labelled ExampleSet. The Apply Model operator for example provides labeled data. Make sure that the ExampleSet has a *label* attribute and a *prediction* attribute. See the Set Role operator for more details.

**performance *(per)*** This is an optional parameter. It requires a Performance Vector.

## Output Ports

**performance *(per)*** This port delivers a Performance Vector (we call it output-performance-vector for now). The Performance Vector is a list of performance criteria values. The output-performance-vector contains performance criteria calculated by this Performance operator (we call it calculated-performance-vector here). If a Performance Vector was also fed at the input port (we call it input-performance-vector here), the criteria of the input-performance-vector are also added in the output-performance-vector. If the input-performance-vector and the calculated-performance-vector both have the same criteria but with different values, the values of the calculated-performance-vector are delivered through the output port. This concept can be easily understood by studying the attached Example Process.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**use example weights *(boolean)*** This parameter allows example *weights* to be used for performance calculations if possible. This parameter has no effect if no attribute has *weight* role. In order to consider *weights* of examples the ExampleSet should have an attribute with weight role. Several operators are available that assign *weights* e.g. the Generate Weights operator. Please study the Set Roles operator for more information regarding *weight* roles.

## Tutorial Processes

### Assessing the performance of a prediction

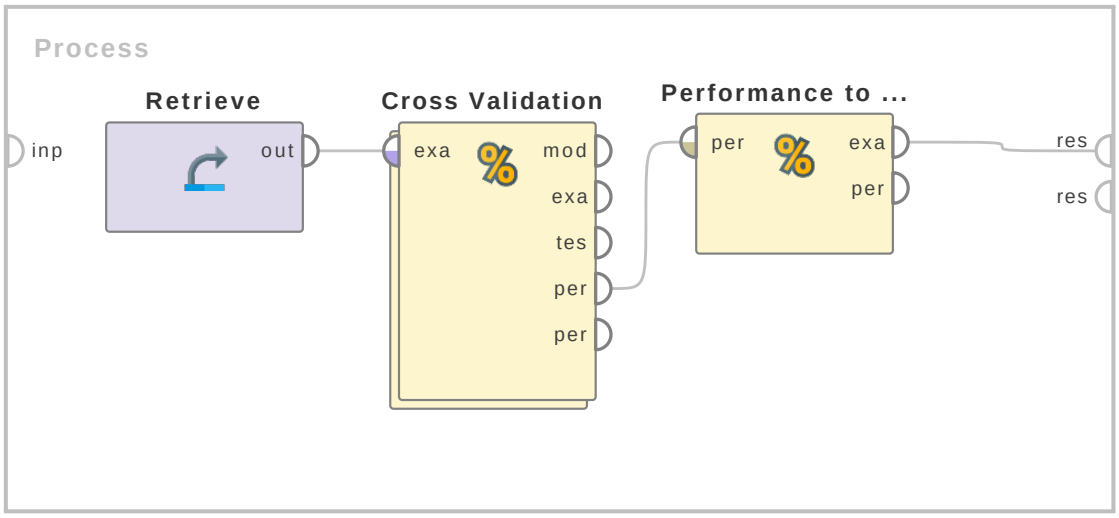



Figure 6.10: Tutorial process 'Assessing the performance of a prediction'.

This process is composed of two Subprocess operators and one Performance operator. Double click on the first Subprocess operator and you will see the operators within this subprocess. The

first subprocess 'Subprocess (labeled data provider)' loads the 'Golf' data set using the Retrieve operator and then learns a classification model using the k-NN operator. Then the learnt model is applied on the 'Golf-Testset' data set using the Apply Model operator. Then Generate Weight operator is used to add an attribute with weight role. Thus, this subprocess provides a labeled ExampleSet with a weight attribute. A breakpoint is inserted after this subprocess to show this ExampleSet. It is provided at the labeled data input port of the Performance operator in the main process.

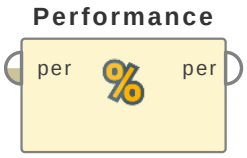
The second Subprocess operator 'Subprocess (performance vector provider)' loads the 'Golf' data set using the Retrieve operator and then learns a classification model using the k-NN operator. Then the learnt model is applied on the 'Golf' data set using the Apply Model operator. Then the Performance (Classification) operator is applied on the labeled data to produce a Performance Vector. A breakpoint is inserted after this subprocess to show this Performance Vector. Note that this model was trained and tested on the same data set ('Golf' data set), so its accuracy is 100%. Thus this subprocess provides a Performance Vector with 100% accuracy and 0.00% classification error. This Performance Vector is connected to the performance input port of the Performance operator in the main process.

When you run the process, first you will see an ExampleSet which is output of the first Subprocess operator. Press the Run button again and you will see a Performance Vector. This is the output of the second Subprocess operator. Press the Run button again and you will see various criteria in the criterion selector window in the Results Workspace. These include classification error, accuracy, precision, recall, AUC (optimistic), AUC and AUC (pessimistic). Now select accuracy from the criterion selector window, its value is 71.43%. On the contrary the accuracy of the input Performance Vector provided by the second subprocess was 100%. The accuracy of the final Performance Vector is 71.43% instead of 100% because if the input Performance Vector and the calculated Performance Vector both have the same criteria but with different values, the values of the calculated Performance Vector are delivered through the output port. Now, note that the classification error criterion is added to the criteria list because of the Performance Vector provided at the performance input port. Disable the second Subprocess operator and run the same process again, you will see that the classification error criterion does not appear now. This is because if a Performance Vector is fed at the performance input port, its criteria are also added to the output Performance Vector.

Accuracy is calculated by taking the percentage of correct predictions over the total number of examples. Correct prediction means examples where the value of the prediction attribute is equal to the value of the label attribute. If you look at the ExampleSet in the Results Workspace, you can see that there are 14 examples in this data set. 10 out of 14 examples are correct predictions i.e. their label and prediction attributes have the same values. This is why the accuracy was 71.43% (10 x 100 /14 = 71.43%). Now run the same process again but this time set the use example weights parameter to true. Check the results again. They have changed now because the weight of each example was taken into account this time. The accuracy is 68.89% this time. If you take the percentage of the weight of the correct predictions and the total weight you get the same answer (0.6889 x 100/1 = 68.89%). In this Example Process, using weights reduced the accuracy but this is not always the case.

Note: This Example Process is just for highlighting different perspectives of the Performance operator. It may not be very useful in real scenarios.

# Performance (Min-Max)




This operator takes a performance vector and puts all criteria into a min-max criterion which delivers the minimum instead of the average or arbitrary weighted combinations.

## Description

The Performance (Min-Max) operator wraps a min-max criterion around each performance criterion of the given performance vector. This criterion uses the minimum fitness achieved instead of the average fitness or arbitrary weightings. Please note that the average values stay the same and only the fitness values change.

## Input Ports

**performance *(per)*** This input port expects a performance vector. A performance vector is a list of performance criteria values.

## Output Ports

**performance *(per)*** The resultant performance vector is returned through this port.

## Parameters

**minimum weight *(real)*** This parameter defines the weight for the minimum fitness against the average fitness.

## Tutorial Processes

### Introduction to the Performance (Min-Max) operator

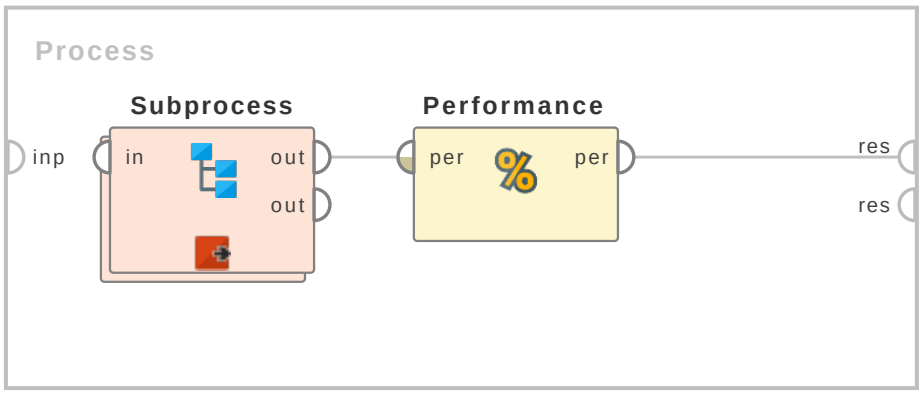
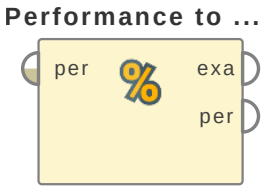



Figure 6.11: Tutorial process 'Introduction to the Performance (Min-Max) operator'.

This Example Process starts with the Subprocess operator. The subprocess delivers a performance vector. A breakpoint is inserted here so that you can have a look at the performance vector. This performance vector is provided as input to the Performance (Min-Max) operator which wraps a min-max criterion around each performance criterion of the given performance vector. The resultant performance vector can be seen in the Results Workspace.

# Performance to Data




This operator is used to convert a performance vector as produced by a Performance operator into an example set.

## Description

The operator creates an example set which contains one row for each performance criterion in the input data and a set of columns: the Criterion column contains the name of the criterion whereas Value, Standard Deviation and Variance list the value and the statistical properties of it.

## Input Ports

**performance vector *(per)*** This port expects a Performance Vector. It is the output of the Validation operator in the attached Example Process. The output of other Performance operators can also be used as input.

## Output Ports

**example set *(exa)*** The ExampleSet which results from the conversion of the Performance Vector.

**performance vector *(per)*** The performance vector that was given as input is passed without changing to the output through this port. This is usually used to reuse the same performance vector in further operators or to view it in the Results Workspace.

## Tutorial Processes

### Assessing the performance of a prediction

This process is composed from three parts: a Retrieve operator which retrieves the Sonar example data set, a Cross Validation operator which evaluates a simple Naive Bayes model, and the Performance to Data operator.

The performance to data operator simply converts the performance vector which is output from the Cross Validation operator into an ExampleSet. You can see this ExampleSet and the original Performance Vector in the Result view.

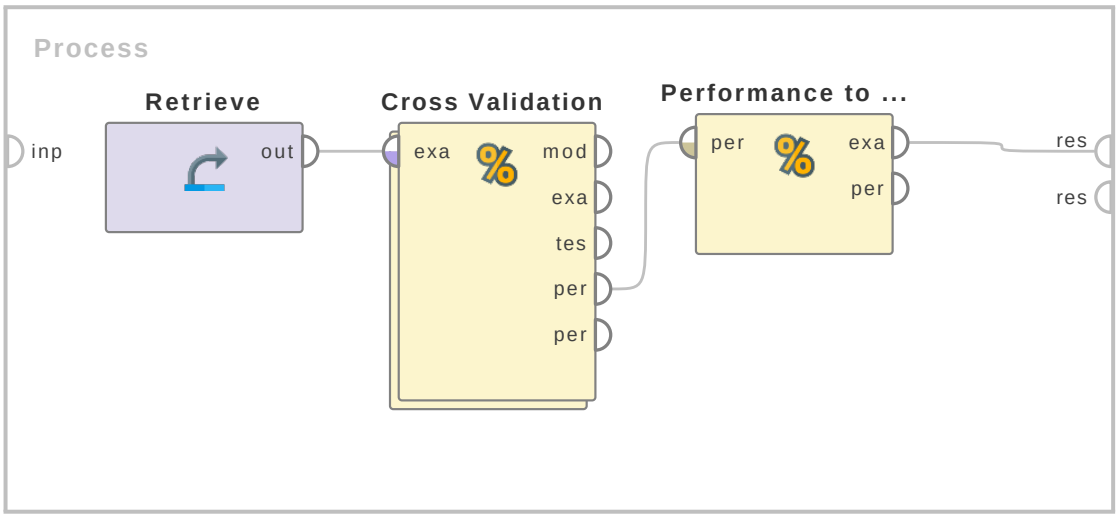
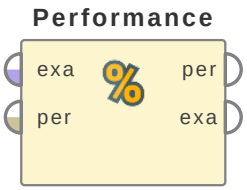



Figure 6.12: Tutorial process 'Assessing the performance of a prediction'.

## 6.1.1 Predictive

### Performance (Attribute Count)




This operator creates a performance vector containing the attribute count of the input ExampleSet.

#### Description

This is a very simple operator. It takes an ExampleSet as input and returns a performance vector that has the count of attributes in the given ExampleSet. Optionally, a performance vector can be provided as input as well. In that case the 'number of attributes' criteria is appended to the given performance vector.

#### Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process.

**performance *(per)*** This optional port expects a performance vector. A performance vector is a list of performance criteria values.

#### Output Ports

**performance *(per)*** The performance vector containing the 'number of attributes' criteria is returned through this port.

**example set *(exa)*** ExampleSet that was given as input is passed without any modifications to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**optimization direction *(selection)*** This is an expert parameter. It indicates if the fitness should be maximal for the maximal or the minimal number of features.

## Tutorial Processes

### Generating a performance vector with the 'number of attributes' criteria

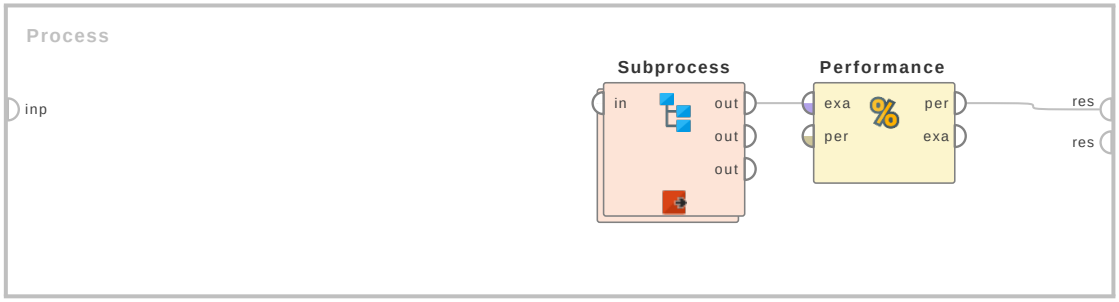
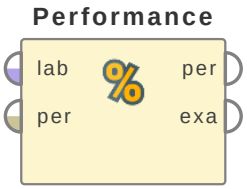



Figure 6.13: Tutorial process 'Generating a performance vector with the 'number of attributes' criteria'.

This Example Process starts with the Subprocess operator. The subprocess delivers an ExampleSet and a performance vector. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has four regular attributes. This ExampleSet is provided as input to the Performance (Attribute Count) operator which returns a performance vector with the 'number of attributes' criteria. As there were four attributes in the given ExampleSet, the 'number of attributes' criteria has value 4. Now connect the second output port of the Subprocess operator to the performance input port of the Performance (Attribute Count) operator. Run the process again, you will see that this time the 'number of attributes' parameter is appended to the given performance vector.

# Performance Binominal Classification




This Operator is used to statistically evaluate the strengths and weaknesses of a binary classification, after a trained model has been applied to labelled data.

## Description

A binary classification makes predictions where the outcome has two possible values: call them positive and negative. Moreover, the prediction for each Example may be right or wrong, leading to a 2x2 confusion matrix with 4 entries:

- TP - the number of "true positives", positive Examples that have been correctly identified
- FP - the number of "false positives", negative Examples that have been incorrectly identified
- FN - the number of "false negatives", positive Examples that have been incorrectly identified
- TN - the number of "true negatives", negative Examples that have been correctly identified

In the parameter section, numerous performance criteria are described, any of which can be calculated in terms of the above variables.

If the model has a probabilistic scoring system where scores above a certain threshold are identified as positive, then the elements of the confusion matrix will depend on the threshold. To create an ROC graph and calculate the area under the curve (AUC), the threshold is varied and a point (x, y) is plotted for each threshold value:

- y-axis - true positive rate = (True positive predictions)/(Number of positive Examples) = TP / (TP + FN)
- x-axis - false positive rate = (False positive predictions)/(Number of negative Examples) = FP / (FP + TN)

## Differentiation

There are numerous performance Operators, and you should choose the one that is best suited to your problem.

- **Performance (Classification)**

  Choose this Operator when the label is nominal and it has more than two values.

  See page 775 for details.

## Input Ports

**labeled data *(lab)*** This input port expects a labeled ExampleSet. Make sure the ExampleSet has both a label Attribute and a prediction Attribute, and that the label is of type binominal.

**performance *(per)*** This input port expects a performance vector. You need to connect a performance vector to the input if you want to do multi-objective optimization.

## Output Ports

**performance *(per)*** This output port delivers a performance vector – a list of performance criterion values based on the label and prediction Attributes of the input ExampleSet.

In the output, the performance criterion values from the input (if any) are combined with the values from this Operator; in case of overlap, the values from the input are overwritten.

**example set *(exa)*** The ExampleSet that was given as input is passed through without changes.

## Parameters

**main criterion** The main criterion is used when performance vectors are compared, e.g., parameter Optimization or Attribute selection. If not selected, the main criterion is the first criterion in the output performance vector.

If performance vectors are not compared, the main criterion is ignored.

**accuracy** accuracy = (Correct predictions)/(Number of Examples) = (TP + TN) / (TP + FP + FN + TN)

**classification error** classification error = (Incorrect predictions)/(Number of Examples) = (FP + FN) / (TP + FP + FN + TN)

**kappa** Cohen's kappa = (po - pe)/(1 - pe)

where:

po = observed accuracy = (TP + TN) / (TP + FP + FN + TN)

pe = expected accuracy = [(TP + FP)(TP + FN) + (FN + TN)(FP + TN)] / [(TP + FP + FN + TN)^2]

**AUC (optimistic)** When the ROC graph is plotted, before calculating the area under the curve (AUC), the predictions are sorted by score, from highest to lowest, and the graph is plotted Example by Example. If two or more Examples have the same score, the ordering is not well-defined; in this case, the optimistic version of AUC plots the positive Examples before plotting the negative Examples.

**AUC** When the ROC graph is plotted, before calculating the area under the curve (AUC), the predictions are sorted by score, from highest to lowest, and the graph is plotted Example by Example. If two or more Examples have the same score, the ordering is not well-defined. The normal version of AUC calculates the area by taking the average of AUC (optimistic) and AUC (pessimistic).

**AUC (pessimistic)** When the ROC graph is plotted, before calculating the area under the curve (AUC), the predictions are sorted by score, from highest to lowest, and the graph is plotted Example by Example. If two or more Examples have the same score, the ordering is not well-defined; in this case, the pessimistic version of AUC plots the negative Examples before plotting the positive Examples.

**precision** precision = (True positive predictions)/(All positive predictions) = TP / (TP + FP)

**recall** recall = (True positive predictions)/(Number of positive Examples) = TP / (TP + FN)

**lift** lift is the ratio of two quantities, representing the improvement over random sampling:

1. The probability of choosing a positive Example from the group of all positive predictions: TP / (TP + FP)

2. The probability of choosing a positive Example from the group of all Examples: (TP + FN) / (TP + FP + FN + TN)

lift = [TP / (TP + FP)] / [(TP + FN) / (TP + FP + FN + TN)]

**fallout** fallout = (False positive predictions)/(Number of negative Examples) = FP / (FP + TN)

**f measure** F1 = 2 (precision * recall) / (precision + recall) = 2TP / (2TP + FP + FN)

**false positive** The number of false positive predictions: FP

**false negative** The number of false negative predictions: FN

**true positive** The number of true positive predictions: TP

**true negative** The number of true negative predictions: TN

**sensitivity** sensitivity = recall = (True positive predictions)/(Number of positive Examples) = TP / (TP + FN)

**specificity** specificity = (True negative predictions)/(Number of negative Examples) = TN / (TN + FP)

**youden** Sometimes called informedness or DeltaP'.

J = sensitivity + specificity - 1

**positive predictive value** PPV = precision = (True positive predictions)/(All positive predictions) = TP / (TP + FP)

**negative predictive value** NPV = (True negative predictions)/(All negative predictions) = TN / (TN + FN)

**psep** Sometimes called markedness or DeltaP.

psep = PPV + NPV - 1

**skip undefined labels** When this parameter is true, Examples not belonging to a defined class are ignored.

**comparator class** The fully qualified classname of the PerformanceComparator implementation is specified here.

**use example weights** This parameter has no effect if no Attribute has the weight role.

## Tutorial Processes

### Separate mines from rocks

The Sonar data set contains 111 Examples obtained by bouncing sonar signals off a metal cylinder (a "mine") at various angles and under various conditions, and 97 Examples obtained from rocks under similar conditions. The transmitted sonar signal is a frequency-modulated chirp, rising in frequency. The data set contains signals obtained from a variety of different aspect angles, spanning 90 degrees for the cylinder and 180 degrees for the rock.

Each Example has 60 Attributes in the range 0.0 to 1.0. Each Attribute represents the energy within a particular frequency band, integrated over a certain period of time. The integration aperture for higher frequencies occur later in time, since these frequencies are transmitted later during the chirp.

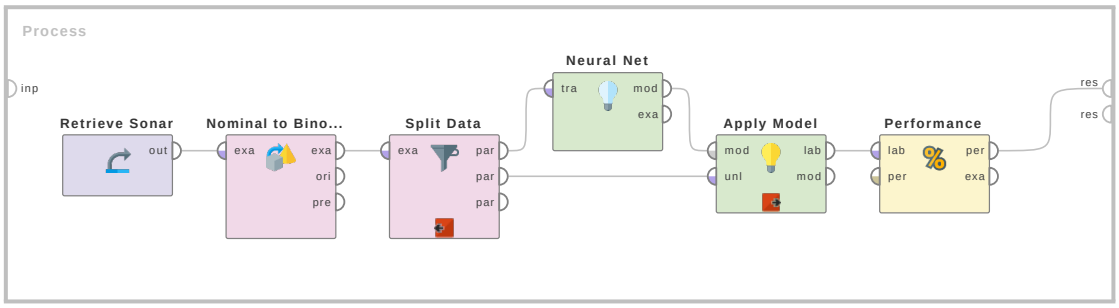



Figure 6.14: Tutorial process 'Separate mines from rocks'.

In the first Tutorial Process, a predictive model is created to identify mines, based on the sonar signal. When you run the Process, the output is displayed in three steps:

1. The whole Sonar data set is displayed.
2. A subset of the Sonar data set is displayed, with predictions based on Neural Net.
3. An ROC graph is displayed in red, together with the threshold values in blue. To see the confusion matrix, click on "recall" or "false negative", where you will learn that the model discovers 90% of the mines, with 4 false negatives (mines that were identified as rocks).

Because the input of the Operator Performance (Binominal Classification) demands labelled data of type "binominal", the label for the original Sonar data must first be converted from "nominal" to "binominal" via the Operator Nominal to Binominal. This type conversion step is unnecessary if the final Operator is Performance (Classification), which accepts a nominal label as input.

### Separate mines from rocks, with Cross Validation

A more realistic perspective on mine discovery is achieved by using Cross Validation. The second Tutorial Process is similar to the first Tutorial Process, but now 5 different versions of the Neural Net model are created, and the results are combined. The Operator Cross Validation takes the place of Split Data, and Performance (Binominal Classification) is part of the testing subprocess.

The output is again an ROC graph, but this time the lines on the graph have a spread which reflects the uncertainty in model building. If you click on "recall" to look at the confusion matrix, you will learn that the resultant model discovers 82% +/- 8% of the mines.

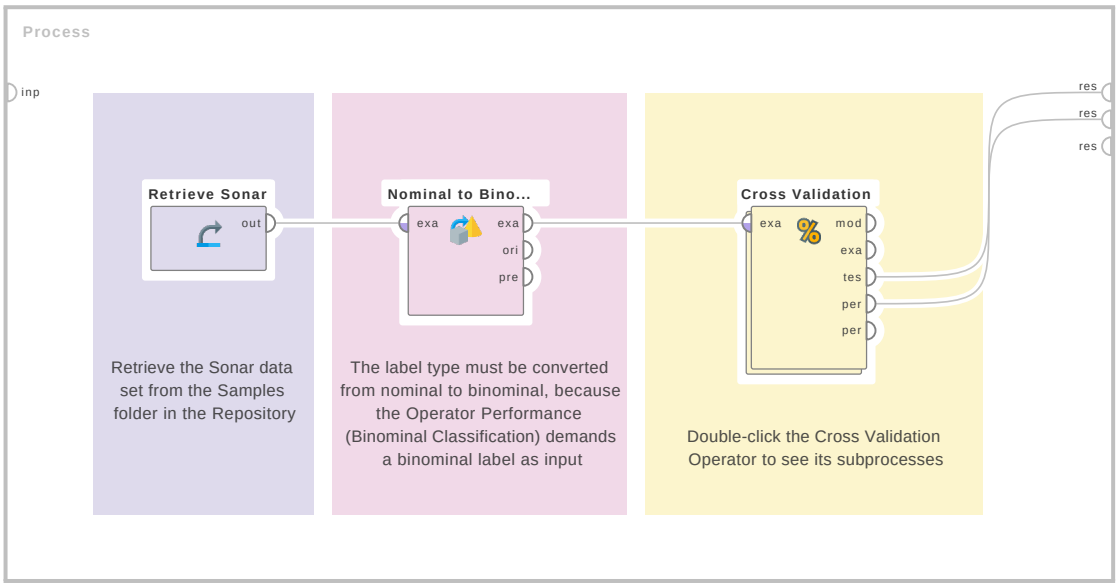
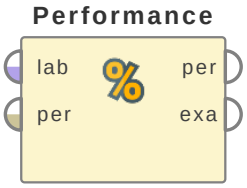



Figure 6.15: Tutorial process 'Separate mines from rocks, with Cross Validation'.

# Performance (Classification)




This operator is used for statistical performance evaluation of classification tasks. This operator delivers a list of performance criteria values of the classification task.

## Description

This operator should be used for performance evaluation of only classification tasks. Many other performance evaluation operators are also available in RapidMiner e.g. Performance operator, Performance (Binominal Classification) operator, Performance (Regression) operator etc. The Performance (Classification) operator is used with classification tasks only. On the other hand, the Performance operator automatically determines the learning task type and calculates the most common criteria for that type. You can use the Performance (User-Based) operator if you want to write your own performance measure.

Classification is a technique used to predict group membership for data instances. For example, you may wish to use classification to predict whether the train on a particular day will be 'on time', 'late' or 'very late'. Predicting whether a number of people on a particular event would be 'below- average', 'average' or 'above-average' is another example. For evaluating the statistical performance of a classification model the data set should be labeled i.e. it should have an attribute with *label* role and an attribute with *prediction* role. The *label* attribute stores the actual observed values whereas the *prediction* attribute stores the values of *label* predicted by the classification model under discussion.

## Input Ports

**labeled data *(lab)*** This input port expects a labeled ExampleSet. The Apply Model operator is a good example of such operators that provide labeled data. Make sure that the ExampleSet has a *label* attribute and a *prediction* attribute. See the Set Role operator for more details regarding *label* and *prediction* roles of attributes.

**performance *(per)*** This is an optional parameter. It requires a Performance Vector.

## Output Ports

**performance *(per)*** This port delivers a Performance Vector (we call it *output-performance-vector* for now). The Performance Vector is a list of performance criteria values. The Performance vector is calculated on the basis of the *label* attribute and the *prediction* attribute of the input ExampleSet. The *output-performance-vector* contains performance criteria calculated by this Performance operator (we call it *calculated-performance-vector* here). If a Performance Vector was also fed at the *performance* input port (we call it *input-performance-vector* here), criteria of the *input-performance-vector* are also added in the *output-performance-vector*. If the *input-performance-vector* and the *calculated-performance-vector* both have the same criteria but with different values, the values of *calculated-performance-vector* are delivered through the output port. This concept can be easily understood by studying the attached Example Process.

**example set *(exa)*** ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**main criterion** The main criterion is used for comparisons and needs to be specified only for processes where performance vectors are compared, e.g. attribute selection or other meta optimization process setups. If no *main criterion* is selected, the first criterion in the resulting performance vector will be assumed to be the *main criterion*.

**accuracy *(boolean)*** Relative number of correctly classified examples or in other words percentage of correct predictions

**classification error *(boolean)*** Relative number of misclassified examples or in other words percentage of incorrect predictions.

**kappa *(boolean)*** The kappa statistics for the classification. It is generally thought to be a more robust measure than simple percentage correct prediction calculation since it takes into account the correct prediction occurring by chance.

**weighted mean recall *(boolean)*** The weighted mean of all per class recall measurements. It is calculated through class recalls for individual classes. Class recalls are mentioned in the last row of the matrix displayed in the Results Workspace.

**weighted mean precision *(boolean)*** The weighted mean of all per class precision measurements. It is calculated through class precisions for individual classes. Class precisions are mentioned in the last column of the matrix displayed in the Results Workspace.

**spearman rho** ***(boolean)*** The rank correlation between the actual and predicted *labels*, using Spearman's rho. Spearman's rho is a measure of the linear relationship between two variables. The two variables in this case are *label* attribute and *prediction* attribute.

**kendall tau** ***(boolean)*** The rank correlation between the actual and predicted *labels*, using Kendall's tau. Kendall's tau is a measure of correlation, and so measures the strength of the relationship between two variables. The two variables in this case are the *label* attribute and the *prediction* attribute.

**absolute error** ***(boolean)*** Average absolute deviation of the prediction from the actual value. The values of the label attribute are the actual values.

**relative error** ***(boolean)*** Average relative error is the average of the absolute deviation of the prediction from the actual value divided by the actual value. The values of the *label* attribute are the actual values.

**relative error lenient** ***(boolean)*** Average lenient relative error is the average of the absolute deviation of the prediction from the actual value divided by the maximum of the actual value and the prediction. The values of the *label* attribute are the actual values.

**relative error strict** ***(boolean)*** Average strict relative error is the average of the absolute deviation of the prediction from the actual value divided by the minimum of the actual value and the prediction. The values of the *label* attribute are the actual values.

**normalized absolute error** ***(boolean)*** The absolute error divided by the error made if the average would have been predicted.

**root mean squared error** ***(boolean)*** The averaged root-mean-squared error.

**root relative squared error** ***(boolean)*** The averaged root-relative-squared error.

**squared error** ***(boolean)*** The averaged squared error.

**correlation** ***(boolean)*** Returns the correlation coefficient between the *label* and *prediction* attributes.

**squared correlation** ***(boolean)*** Returns the squared correlation coefficient between the *label* and *prediction* attributes.

**cross entropy** ***(boolean)*** The cross-entropy of a classifier, defined as the sum over the logarithms of the true label's confidences divided by the number of examples.

**margin** ***(boolean)*** The margin of a classifier, defined as the minimal confidence for the correct label.

**soft margin loss** ***(boolean)*** The average soft margin loss of a classifier, defined as the average of all 1 - confidences for the correct *label*

**logistic loss** ***(boolean)*** The logistic loss of a classifier, defined as the average of ln(1+exp(-[conf(CC)])) where 'conf(CC)' is the confidence of the correct class.

**skip undefined labels** ***(boolean)*** If set to true, examples with undefined *labels* are skipped.

**comparator class** ***(string)*** This is an expert parameter. The fully qualified *classname* of the *PerformanceComparator* implementation is specified here.

**use example weights *(boolean)*** This parameter allows example *weights* to be used for statistical performance calculations if possible. This parameter has no effect if no attribute has *weight* role. In order to consider *weights* of examples the ExampleSet should have an attribute with *weight* role. Several operators are available that assign *weights* e.g. Generate Weights operator. Study the Set Roles operator for more information regarding *weight* role.

**class weights** This is an expert parameter. It specifies the weights 'w' for all classes. The *Edit List* button opens a new window with two columns. The first column specifies the class name and the second column specifies the *weight* for that class. If the *weight* of a class is not specified, that class is assigned *weight = 1*.

## Tutorial Processes

### Use of performance port in Performance (Classification)

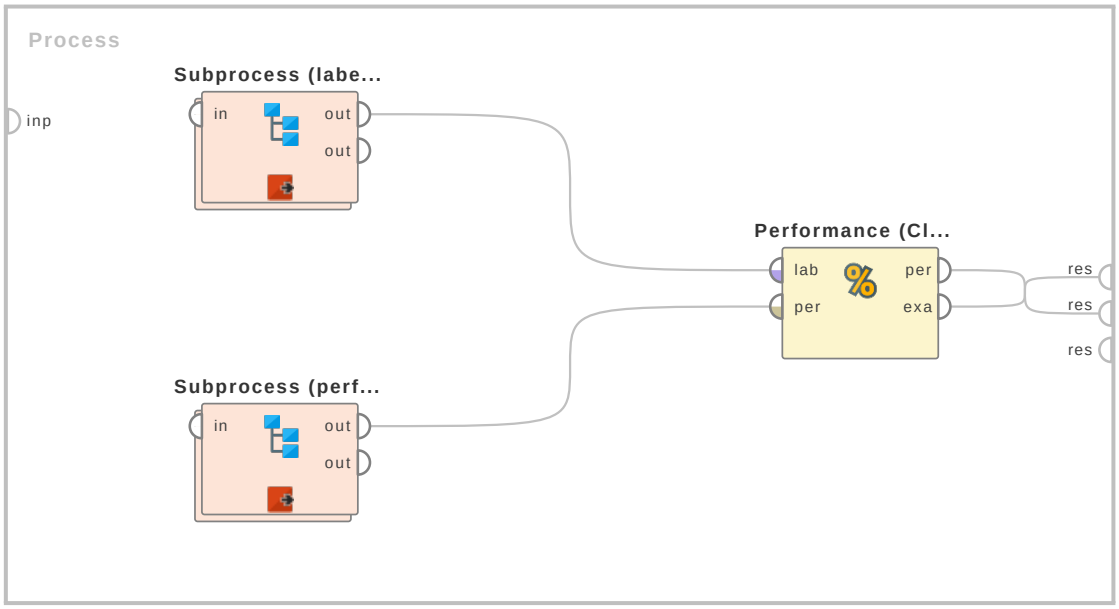



Figure 6.16: Tutorial process 'Use of performance port in Performance (Classification)'.

This Example Process is composed of two Subprocess operators and one Performance (Classification) operator. Double click on the first Subprocess operator and you will see the operators within this subprocess. The first subprocess 'Subprocess (labeled data provider)' loads the 'Golf' data set using the Retrieve operator and then learns a classification model using the k-NN operator. Then the learned model is applied on 'Golf-Testset' data set using the Apply Model operator. Then the Generate Weight operator is used to add an attribute with weight role. Thus, this subprocess provides a labeled ExampleSet with weight attribute. The Breakpoint is inserted after this subprocess to show this ExampleSet. This ExampleSet is provided at the labeled data input port of the Performance (Classification) operator in the main process.

The second Subprocess operator ' Subprocess (performance vector provider) ' loads the' Golf ' data set using the Retrieve operator and then learns a classification model using the k-NN op-

erator. Then the learned model is applied on the' Golf' data set using the Apply Model operator. Then the Performance (Classification) operator is applied on the labeled data to produce a Performance Vector. The Breakpoint is inserted after this subprocess to show this Performance Vector. Note that this model was trained and tested on the same data set (Golf data set), so its accuracy is 100%. Thus this subprocess provides a Performance Vector with 100% accuracy and 0.00% classification error. This Performance Vector is connected to the performance input port of the Performance (Classification) operator in the main process.

When you run the process, first you will see an ExampleSet which is the output of the first Subprocess operator. Press the Run button again and now you will see a Performance Vector. This is the output of the second Subprocess operator. Press the Run button again and you will see various criteria in the criterion selector window in the Results Workspace. These include classification error, accuracy, weighted mean recall and weighted mean precision. Now select the accuracy from the criterion selector window, its value is 71.43%. On the contrary the accuracy of the input Performance Vector provided by the second subprocess was 100%. The accuracy of the final Performance Vector is 71.43% instead of 100% because if the input-performance-vector and the calculated-performance-vector both have same criteria but with different values, the values of the calculated-performance -vector are delivered through the output port. Now, note that the classification error criterion is added to the criteria list because of the Performance Vector provided at the performance input port. Disable the second Subprocess operator and run the same process again, you will see that classification error criterion does not appear now. This is because if a Performance Vector is fed at the performance input port, its criteria are also added to the output-performance-vector.

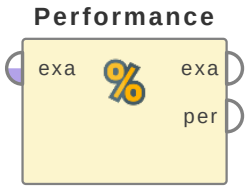
The accuracy is calculated by taking the percentage of correct predictions over the total number of examples. Correct prediction means the examples where the value of the prediction attribute is equal to the value of label attribute. If you look at the ExampleSet in the Results Workspace, you can see that there are 14 examples in this data set. 10 out of 14 examples are correct predictions i.e. their label and prediction attributes have the same values. This is why accuracy was 71.43% (10 x 100 /14 = 71.43%). Now run the same process again but this time set use example weights parameter to true. Check the results again. They have changed now because the weight of each example was taken into account this time. The accuracy is 68.89% this time. If you take the percentage of weight of correct predictions and the total weight you get the same answer (0.6889 x 100/1 = 68.89%). In this Example Process, using weights reduced the accuracy but this is not always the case.

The weighted mean recall is calculated by taking the average of recall of every class. As you can see in the last row of the resultant matrix in the Results Workspace, class recall for 'true no' is 60% and class recall for 'true yes' is 77.78%. Thus weighted mean recall is calculated by taking the average of these class recall values (((77.78%)+(60%))/2=68.89%).

The weighted mean precision is calculated by taking the average of precision of every class. As you can see in the last column of the resultant matrix in the Results Workspace, class precision for 'pred. no' is 60% and class precision for 'pred. yes' is 77.78%. Thus weighted mean precision is calculated by taking the average of these class precision values (((77.78%)+(60%))/2=68.89%). These values are for the case when the use example weights parameter is set to false.

Note: This Example Process is just for highlighting different perspectives of Performance (Classification) operator. It may not be very useful in real scenarios.

# Performance (Costs)




This operator provides the ability to evaluate misclassification costs for performance evaluation of classification tasks.

## Description

The Performance (Costs) operator provides the ability to evaluate misclassification costs. A cost matrix should be specified through the *cost matrix* parameter. The cost matrix is similar in structure to a confusion matrix because it has predicted classes in one dimension and actual classes on the other dimension. Therefore the cost matrix can denote the costs for every possible classification outcome: predicted label vs. actual label. Actually this matrix is a matrix of misclassification costs because you can specify different weights for certain classes misclassified as other classes. Weights can also be assigned to correct classifications but they are not taken into account for evaluating misclassification costs. The classes in the matrix are labeled as Class 1, Class 2 etc where classes are numbered according to their order in the internal mapping. The *class order definition* parameter allows you to specify the class order for the matrix in which case classes are ordered according to the order specified in this parameter (instead of internal mappings). For a better understanding of this operator please study the attached Example Process.

## Input Ports

**example set *(exa)*** This input port expects a labeled ExampleSet. The Apply Model operator is a good example of such operators that provide labeled data. Make sure that the ExampleSet has *label* and *prediction* attributes. Please see the Set Role operator for more details about attribute roles.

## Output Ports

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**performance *(per)*** This port delivers a Performance Vector which has information about the misclassification costs.

## Parameters

**cost matrix *(string)*** This parameter is used for specifying the cost matrix. The cost matrix is similar in structure to a confusion matrix because it has predicted classes in one dimension and actual classes on the other dimension. Therefore the cost matrix can denote the costs for every possible classification outcome: predicted label vs. actual label. Actually this matrix is a matrix of misclassification costs because you can specify different weights for certain classes misclassified as other classes. Weights can also be assigned to correct classifications but they are not taken into account for evaluating misclassification costs. The classes in the matrix are labeled as Class 1, Class 2 etc where classes are numbered

according to their order in the internal mapping. The *class order definition* parameter can be used for specifying the class order for the matrix (instead of internal mappings).

**class order definition *(enumeration)*** The *class order definition* parameter allows you to specify the class order for the cost matrix in which case classes are ordered according to the order specified in this parameter (instead of internal mappings).

## Tutorial Processes

### Measuring Misclassification costs of a classifier


Figure 6.17: Tutorial process 'Measuring Misclassification costs of a classifier'.

The 'Golf' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a classification model. The Naive Bayes operator is applied in the training subprocess of the Split Validation operator. The Naive Bayes operator trains a classification model. The Apply Model operator is used in the testing subprocess to apply this model. A breakpoint is inserted here so that you can have a look at the labeled ExampleSet. As you can see, out of 4 examples of the testing data set only 1 has been misclassified. The misclassified example was classified as 'class = yes' while actually it was 'class = no'.

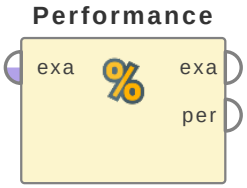
The resultant labeled ExampleSet is used by the Performance (Costs) operator for measuring the misclassification costs of the model. Have a look at the parameters of the Performance (Costs) operator. The class order definition parameter specifies the order of classes in the cost matrix. The classes 'yes' and 'no' are placed in first and second rows respectively. Thus class 'yes' is Class 1 and class 'no' is Class 2 in the cost matrix. Now have a look at the cost matrix in the cost matrix parameter. The case where Class 2 (i.e. class = no) is misclassified as Class 1 (i.e. class = yes) has been given weight 2.0. The case where Class 1 (i.e. class = yes) is misclassified as Class 2 (i.e. class = no) has been given weight 1.0.

Now let us see how this cost matrix is used for evaluating misclassification costs of the labeled ExampleSet. As 1 of the 4 classifications was wrong, one should expect the classification cost to be 1/4 or 0.250. But as this misclassification has weight 2.0 (because class = no is misclassified as class = yes) instead of 1.0 the cost for this misclassification is doubled. Therefore the cost in this case is 0.500. The misclassification cost can be seen in the Results Workspace.

Now set the sampling type parameter of the Split Validation operator to 'linear sampling' and run the process again. Have a look at the labeled ExampleSet generated by the Apply Model operator. 2 out of 4 examples have been misclassified. One example with class = no has been misclassified as class = yes (i.e. weight = 2.0) and one example with class = yes has been mis-

classified as class = no (i.e. weight = 1.0). The resultant misclassification cost is ((1 x 1.0)+(1 x 2.0))/4 which results to 0.750. The misclassification cost can be seen in the Results Workspace.

# Performance (Ranking)




This operator delivers a performance value representing costs for the confidence rank of the true label.

## Description

The Performance (Ranking) operator should be used for tasks, where it is not only important that the real class is selected, but also that it receives a comparably high confidence.

This operator will sort the confidences for each label and depending on the rank position of the real label, costs are generated. You can define these costs by the parameter ranking_costs. The costs are entered for whole intervals, so you don't have to enter a cost value for each rank. These intervals are defined by their start rank and range either until the start of the next interval or infinite. Everything before the first mentioned rank will receive costs of 0. The counting of rank starts with 0, so the most confident label is rank 0.

The costs are entered on the right side of the table.

For example, if you want to assign costs of zero if the true label is predicted with the highest confidence, 1 for the second place, 2 for the third and 10 for each following, you have to enter:

1 1
2 2
3 10

## Input Ports

**labeled data *(lab)*** This input port expects a labeled ExampleSet. The Apply Model operator is a good example of such operators that provide labeled data. Make sure that the ExampleSet has a *label* attribute and a *prediction* attribute. See the Set Role operator for more details regarding *label* and *prediction* roles of attributes.

## Output Ports

**example set *(exa)*** ExampleSet that was given as input is passed without change to this output port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**performance *(per)*** This port delivers a Performance Vector (we call it *output-performance-vector* for now). The Performance Vector is a list of performance criteria values. The Performance vector is calculated on the basis of the *label* attribute and the *prediction* attribute of the input ExampleSet. The *output-performance-vector* contains performance criteria calculated by this Performance operator (we call it *calculated-performance-vector* here). If a Performance Vector was also fed at the *performance* input port (we call it *input-performance-vector* here), criteria of the *input-performance-vector* are also added in the *output-performance-vector*. If the *input-performance-vector* and the *calculated-performance-vector* both have the same criteria but with different values, the values of *calculated-performance-vector* are delivered through the output port. This concept can be easily understood by studying the attached Example Process.

## Parameters

**ranking costs *(list)*** Table defining the costs when the real label isn't the one with the highest confidence

## Tutorial Processes

### Applying the Performance (Ranking) operator on the Golf data set

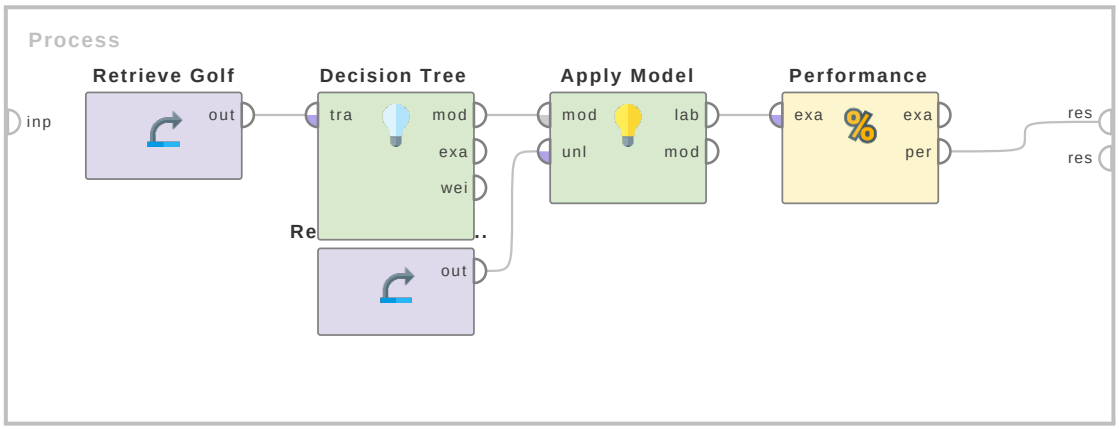
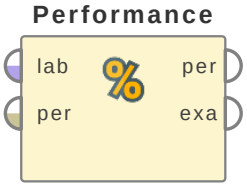



Figure 6.18: Tutorial process 'Applying the Performance (Ranking) operator on the Golf data set'.

The 'Golf' data set is loaded using the Retrieve operator. The Decision Tree operator is applied on it with default values for all parameters. The Tree model generated by the Decision Tree operator is applied on the 'Golf-Testset' data set using the Apply Model operator. Labeled data from the Apply Model operator is provided to the Performance (Ranking) operator. The ranking costs parameter is configured as described above. As result you can see the costs of the prediction made by the Apply Model operator.

# Performance (Regression)




This operator is used for statistical performance evaluation of regression tasks and delivers a list of performance criteria values of the regression task.

## Description

This operator should be used for performance evaluation of regression tasks only. Many other performance evaluation operators are also available in RapidMiner e.g. the Performance operator, Performance (Binominal Classification) operator, Performance (Classification) operator etc. The Performance (Regression) operator is used with regression tasks only. On the other hand, the Performance operator automatically determines the learning task type and calculates the most common criteria for that type. You can use the Performance (User-Based) operator if you want to write your own performance measure.

Regression is a technique used for numerical prediction and it is a statistical measure that attempts to determine the strength of the relationship between one dependent variable ( i.e. the label attribute) and a series of other changing variables known as independent variables (regular attributes). Just like Classification is used for predicting categorical labels, Regression is used for predicting a continuous value. For example, we may wish to predict the salary of university graduates with 5 years of work experience, or the potential sales of a new product given its price. Regression is often used to determine how much specific factors such as the price of a commodity, interest rates, particular industries or sectors influence the price movement of an asset. For evaluating the statistical performance of a regression model the data set should be labeled i.e. it should have an attribute with *label* role and an attribute with *prediction* role. The *label* attribute stores the actual observed values whereas the *prediction* attribute stores the values of *label* predicted by the regression model under discussion.

## Input Ports

**labeled data *(lab)*** This input port expects a labeled ExampleSet. The Apply Model operator is a good example of such operators that provide labeled data. Make sure that the ExampleSet has the *label* and *prediction* attribute. See the Set Role operator for more details regarding the *label* and *prediction* roles of attributes.

**performance *(per)*** This is an optional parameter. It requires a Performance Vector.

## Output Ports

**performance *(per)*** This port delivers a Performance Vector (we call it *output-performance-vector* for now). The Performance Vector is a list of performance criteria values. The Performance vector is calculated on the basis of the *label* and *prediction* attribute of the input ExampleSet. The *output-performance-vector* contains performance criteria calculated by this Performance operator (we call it *calculated-performance-vector* here). If a Performance Vector was also fed at the *performance* input port (we call it *input-performance-vector* here), the criteria of the *input-performance-vector* are also added in the *output-performance-vector*. If the *input-performance-vector* and the *calculated-performance-vector* both have the same

criteria but with different values, the values of the *calculated-performance-vector* are delivered through the output port. This concept can be easily understood by studying the Example Process of the Performance (Classification) operator.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**main criterion** The main criterion is used for comparisons and needs to be specified only for processes where performance vectors are compared, e.g. attribute selection or other meta optimization process setups. If no *main criterion* is selected, the first criterion in the resulting performance vector will be assumed to be the *main criterion*.

**root mean squared error *(boolean)*** The averaged root-mean-squared error.

**absolute error *(boolean)*** The average absolute deviation of the prediction from the actual value. The values of the *label* attribute are the actual values.

**relative error *(boolean)*** The average relative error is the average of the absolute deviation of the prediction from the actual value divided by actual value. Values of the *label* attribute are the actual values.

**relative error lenient *(boolean)*** The average lenient relative error is the average of the absolute deviation of the prediction from the actual value divided by the maximum of the actual value and the prediction. The values of the *label* attribute are the actual values.

**relative error strict *(boolean)*** The average strict relative error is the average of the absolute deviation of the prediction from the actual value divided by the minimum of the actual value and the prediction. The values of the *label* attribute are the actual values.

**normalized absolute error *(boolean)*** The absolute error divided by the error made if the average would have been predicted.

**root relative squared error *(boolean)*** The averaged root-relative-squared error.

**squared error *(boolean)*** The averaged squared error.

**correlation *(boolean)*** Returns the correlation coefficient between the *label* and *prediction* attributes.

**squared correlation *(boolean)*** Returns the squared correlation coefficient between the *label* and *prediction* attributes.

**prediction average *(boolean)*** Returns the average of all the predictions. All the predicted values are added and the sum is divided by the total number of predictions.

**spearman rho *(boolean)*** The rank correlation between the actual and predicted *labels*, using Spearman's rho. Spearman's rho is a measure of the linear relationship between two variables. The two variables in this case are the *label* and the *prediction* attribute.

**kendall tau *(boolean)*** The rank correlation between the actual and predicted *labels*, using Kendall's tau-b. Kendall's tau is a measure of correlation, and so measures the strength of the relationship between two variables. The two variables in this case are the *label* and the *prediction* attribute.

**skip undefined labels *(boolean)*** If set to true, examples with undefined *labels* are skipped.

**comparator class *(string)*** This is an expert parameter. Fully qualified *classname* of the *PerformanceComparator* implementation is specified here.

**use example weights *(boolean)*** This parameter allows example *weights* to be used for statistical performance calculations if possible. This parameter has no effect if no attribute has the *weight* role. In order to consider *weights* of examples the ExampleSet should have an attribute with the *weight* role. Several operators are available that assign *weights* e.g. the Generate Weights operator. Study the Set Roles operator for more information regarding the *weight* role.

## Tutorial Processes

### Applying the Performance (Regression) operator on the Polynomial data set

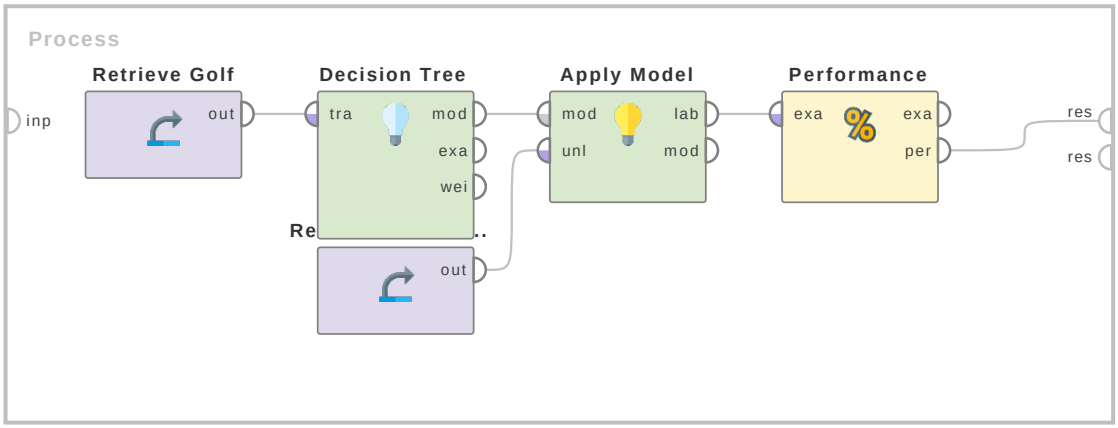
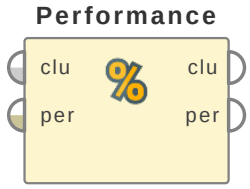



Figure 6.19: Tutorial process 'Applying the Performance (Regression) operator on the Polynomial data set'.

The 'Polynomial' data set is loaded using the Retrieve operator. The Filter Example Range operator is applied on it. The first example parameter of the Filter Example Range parameter is set to 1 and the last example parameter is set to 100. Thus the first 100 examples of the 'Polynomial' data set are selected. The Linear Regression operator is applied on it with default values of all parameters. The regression model generated by the Linear Regression operator is applied on the last 100 examples of the 'Polynomial' data set using the Apply Model operator. Labeled data from the Apply Model operator is provided to the Performance (Regression) operator. The absolute error and prediction average parameters are set to true. Thus the Performance Vector generated by the Performance (Regression) operator has information regarding the absolute error and prediction average in the labeled data set. The absolute error is calculated by adding the difference of all the predicted values from actual values of the label attribute, and dividing this sum by the total number of predictions. The prediction average is calculated by adding all the actual label values and dividing this sum by the total number of examples. You can verify this from the results in the Results Workspace.

## 6.1.2 Segmentation

### Cluster Count Performance




This operator creates a performance vector containing the 'Number of clusters' and 'Cluster Number Index' criteria from a cluster model.

#### Description

This is a very simple operator. It takes a cluster model as input and returns a performance vector that has the 'Number of clusters' and 'Cluster Number Index' criteria. The 'Number of clusters' criteria contains the number of clusters. The 'Cluster Number Index' criteria builds a derived index from the number of clusters by using the formula *1 - (k / n)* with *k* as thw number of clusters and *n* as the number of examples. This can be used for optimizing the coverage of a cluster result with respect to the number of clusters. Optionally, a performance vector can be provided as input as well. In that case the 'Number of clusters' and 'Cluster Number Index' criteria are appended to the given performance vector.

#### Input Ports

**cluster model *(clu)*** This input port expects a cluster model. It is the output of the Subprocess operator in the attached Example Process.

**performance *(per)*** This optional port expects a performance vector. A performance vector is a list of performance criteria values.

#### Output Ports

**cluster model *(clu)*** The cluster model that was given as input is passed without any modifications to the output through this port. This is usually used to reuse the same cluster model in further operators or to view the cluster model in the Results Workspace.

**performance *(per)*** The performance vector containing the 'Number of clusters' and 'Cluster Number Index' criteria is returned through this port.

#### Tutorial Processes

##### Generating a performance vector with the 'Number of clusters' criteria

This Example Process starts with the Subprocess operator. The subprocess delivers a cluster model and a performance vector. A breakpoint is inserted here so that you can have a look at the cluster model. You can see that the cluster model has two clusters. This cluster model is provided as input to the Cluster Count Performance operator which returns a performance vector with the 'Number of clusters' criteria. As there were two clusters in the given cluster model, the 'Number of clusters' criteria has value 2. Now connect the second output port of the Subprocess operator to the performance input port of the Cluster Count Performance operator. Run the

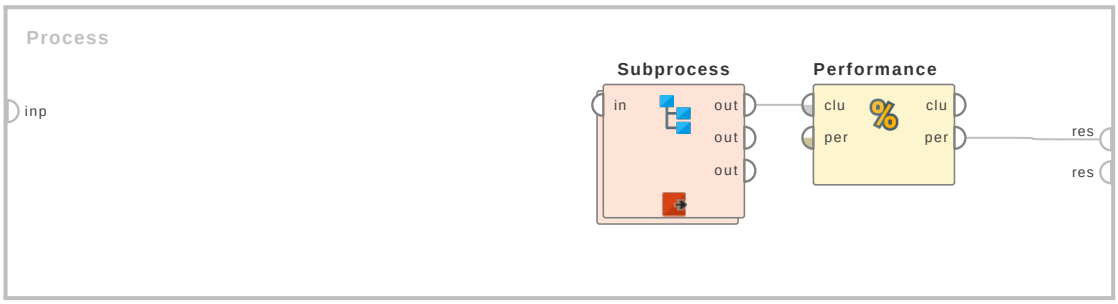
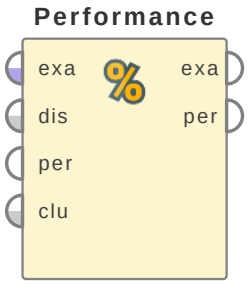



Figure 6.20: Tutorial process ‘Generating a performance vector with the ’Number of clusters’ criteria’.

process again, you will see that this time the ‘Number of clusters’ parameter is appended to the given performance vector.

# Cluster Density Performance




This operator is used for performance evaluation of the centroid based clustering methods. This operator delivers a list of performance criteria values based on cluster densities.

## Description

The centroid based clustering operators like the K-Means and K-Medoids produce a centroid cluster model and a clustered set. The centroid cluster model has information regarding the clustering performed. It tells which examples are parts of which cluster. It also has information regarding centroids of each cluster. The Cluster Density Performance operator takes this centroid cluster model and clustered set as input and evaluates the performance of the model based on the cluster densities. It is important to note that this operator also requires a SimilarityMeasure object as input. This operator is used for evaluation of non-hierarchical cluster models based on the average within cluster similarity/distance. It is computed by averaging all similarities / distances between each pair of examples of a cluster.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. It is a technique for extracting information from unlabeled data and can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Data to Similarity operator in the attached Example Process.

**distance measure *(dis)*** This input port expects a SimilarityMeasure object. It is output of the Data to Similarity operator in the attached Example Process.

**performance vector *(per)*** This optional input port expects a performance vector. A performance vector is a list of performance criteria values.

**cluster model *(clu)*** This input port expects a centroid cluster model. It is output of the K-Means operator in the attached Example Process. The centroid cluster model has information regarding the clustering performed. It tells which examples are part of which cluster. It also has information regarding centroids of each cluster.

## Output Ports

**example set *(exa)*** The ExampleSet that was given as input is passed without any modifications to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**performance vector *(per)*** The performance of the cluster model is evaluated and the resultant performance vector is delivered through this port. A performance vector is a list of performance criteria values.

## Tutorial Processes

### Evaluating the performance of the K-Means clustering model

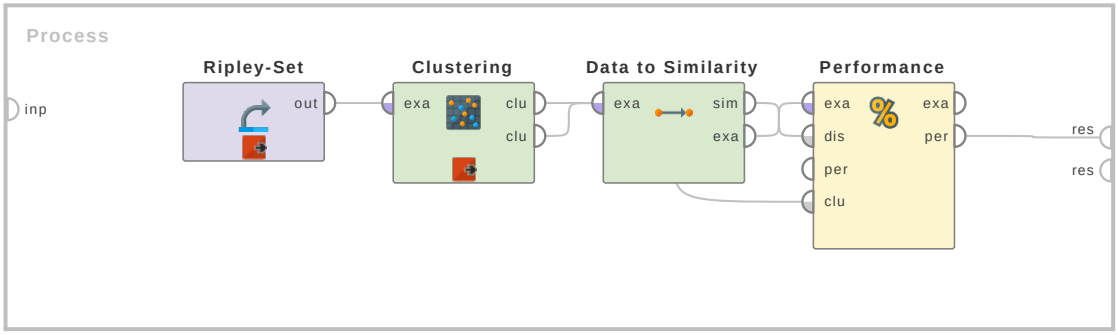
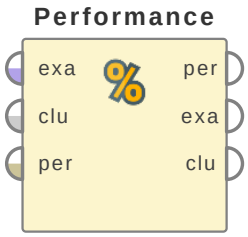



Figure 6.21: Tutorial process 'Evaluating the performance of the K-Means clustering model'.

The 'Ripley-Set' data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters. A breakpoint is inserted at this step so that you can have a look at the ExampleSet before the application of the K-Means operator. The 'Ripley-Set' has two real attributes; 'att1' and 'att2'. The K-Means operator is applied on this data set with default values for all parameters. A breakpoint is inserted at this step so that you can have a look at the results of the K-Means operator. You can see that two new attributes are created by the K-Means operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. As parameter k was set to 2, only two clusters are possible. That is why each example is assigned to either 'cluster_0' or 'cluster_1'.

The Data to Similarity operator is applied on the resultant ExampleSet. This generates a SimilarityMeasure object. The clustered ExampleSet, cluster model and the Similarity Measure object are provided as input to the Cluster Density Performance operator. The Cluster Density Performance operator evaluates the performance of this model and delivers a performance vector that has performance criteria values. The resultant performance vector can be seen in the results workspace.

# Cluster Distance Performance




This operator is used for performance evaluation of centroid based clustering methods. This operator delivers a list of performance criteria values based on cluster centroids.

## Description

The centroid based clustering operators like the K-Means and K-Medoids produce a centroid cluster model and a clustered set. The centroid cluster model has information regarding the clustering performed. It tells which examples are parts of which cluster. It also has information regarding centroids of each cluster. The Cluster Distance Performance operator takes this centroid cluster model and clustered set as input and evaluates the performance of the model based on the cluster centroids. Two performance measures are supported: Average within cluster distance and Davies-Bouldin index. These performance measures are explained in the parameters.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. It is a technique for extracting information from unlabeled data and can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the K-Medoids operator in the attached Example Process.

**cluster model *(clu)*** This input port expects a centroid cluster model. It is output of the K-Medoids operator in the attached Example Process. The centroid cluster model has information regarding the clustering performed. It tells which examples are part of which cluster. It also has information regarding centroids of each cluster.

**performance *(per)*** This input port expects a Performance Vector.

## Output Ports

**performance *(per)*** The performance of the cluster model is evaluated and the resultant Performance Vector is delivered through this port. The Performance Vector is a list of performance criteria values.

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**cluster model *(clu)*** The centroid cluster model that was given as input is passed without changing to the output through this port. This is usually used to reuse the same centroid cluster model in further operators or to view it in the Results Workspace.

## Parameters

**main criterion *(selection)*** This parameter specifies the main criterion to use for performance evaluation.

- **avg._within_centroid_distance** The average within cluster distance is calculated by averaging the distance between the centroid and all examples of a cluster.
- **davies_bouldin** The algorithms that produce clusters with low intra-cluster distances (high intra-cluster similarity) and high inter-cluster distances (low inter-cluster similarity) will have a low Davies–Bouldin index, the clustering algorithm that produces a collection of clusters with the smallest Davies–Bouldin index is considered the best algorithm based on this criterion.

**main criterion only *(boolean)*** This parameter specifies if only the main criterion should be delivered by the performance vector. The main criterion is specified by the *main criterion* parameter

**normalize *(boolean)*** This parameter specifies if the results should be normalized. If set to true, the criterion is divide by the number of features.

**maximize *(boolean)*** This parameter specifies if the results should be maximized. If set to true, the result is not multiplied by minus one.

## Tutorial Processes

### Evaluating the performance of the K-Medoids clustering model

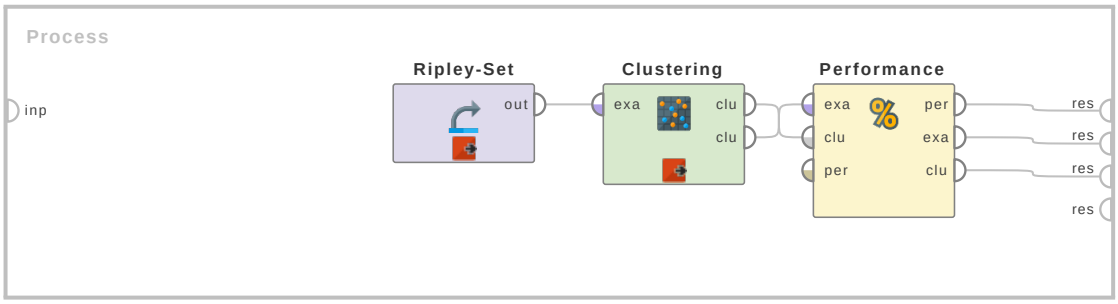


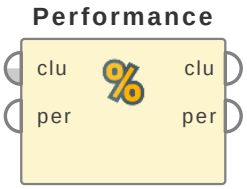

Figure 6.22: Tutorial process ‘Evaluating the performance of the K-Medoids clustering model’.

The ‘Ripley-Set’ data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A breakpoint is inserted at this step so that you can have a look at the ExampleSet before application of the K-Medoids operator. The ‘Ripley-Set’ has two real attributes; ‘att1’ and ‘att2’. The K-Medoids operator is applied on this data set with default values for all parameters. A breakpoint is inserted at this step so that you can have a look at the results of the K-Medoids operator. You can see that two new attributes are created by the K-Medoids operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. As parameter k was set to 2, only two clusters are possible. That is why each example is assigned to either ‘cluster_0’ or ‘cluster_1’. Also note the Plot View of this data. You can clearly see how the algorithm has created two separate groups in the Plot View. A cluster model is also delivered through the cluster model output port. It has information regarding

the clustering performed. Under the Folder View you can see members of each cluster in folder format. You can see information regarding centroids under the Centroid Table and Centroid Plot View tabs.

The Cluster Distance Performance operator is applied to measure the performance of this clustering model. The cluster model and clustered set produced by the K-Medoids operator are provided as input to the Cluster Distance Performance operator which evaluates the performance of this model and delivers a performance vector that has performance criteria values. The resultant performance vector can be seen in the results workspace.

# Item Distribution Performance




This operator is used for performance evaluation of flat clustering methods. It evaluates a cluster model based on the distribution of examples.

## Description

The clustering operators like the K-Means and K-Medoids produce a flat cluster model and a clustered set. The cluster model has information regarding the clustering performed. It tells which examples are parts of which cluster. The Item Distribution Performance operator takes this cluster model as input and evaluates the performance of the model based on the distribution of examples i.e. how well the examples are distributed over the clusters. Two distribution measures are supported: Sum of Squares and Gini Coefficient. These distribution measures are explained in the parameters. Flat clustering creates a flat set of clusters without any explicit structure that would relate clusters to each other. Hierarchical clustering, on the other hand, creates a hierarchy of clusters. This operator can only be applied on models produced by operators that produce flat cluster models e.g. K-Means or K-Medoids operators. It cannot be applied on models created by the operators that produce a hierarchy of clusters e.g. the Agglomerative Clustering operator.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. It is a technique for extracting information from unlabeled data and can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

## Input Ports

**cluster model *(clu)*** This input port expects a flat cluster model. It is output of the K-Medoids operator in the attached Example Process. The cluster model has information regarding the clustering performed. It tells which examples are part of which cluster.

**performance vector *(per)*** This input port expects a Performance Vector.

## Output Ports

**cluster model *(clu)*** The cluster model that was given as input is passed without changing to the output through this port. This is usually used to reuse the same cluster model in further operators or to view it in the Results Workspace.

**performance vector *(per)*** The performance of the cluster model is evaluated and the resultant Performance Vector is delivered through this port. It is a list of performance criteria values.

## Parameters

**measure *(selection)*** This parameter specifies the item distribution measure to apply. It has two options:

- **sumofsquares** If this option is selected, the sum of squares is used as the item distribution measure.
- **ginicoefficient** The Gini coefficient (also known as the Gini index or Gini ratio) is a measure of statistical dispersion. It measures the inequality among values of a frequency distribution. A low Gini coefficient indicates a more equal distribution, with 0 corresponding to complete equality, while higher Gini coefficients indicate a more unequal distribution, with 1 corresponding to complete inequality.

## Tutorial Processes

### Evaluating the performance of the K-Medoids clustering model

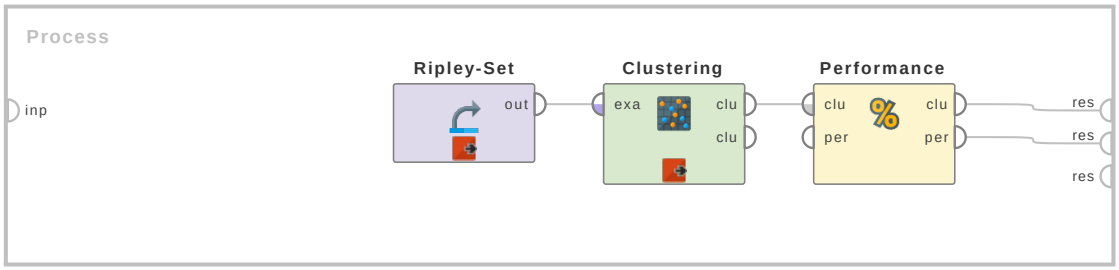
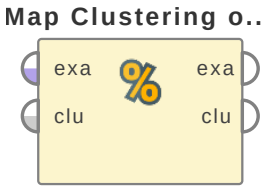



Figure 6.23: Tutorial process ‘Evaluating the performance of the K-Medoids clustering model’.

The ‘Ripley-Set’ data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters themselves. A breakpoint is inserted at this step so that you can have a look at the ExampleSet before the application of the K-Medoids operator. The ‘Ripley-Set’ has two real attributes; ‘att1’ and ‘att2’. The K-Medoids operator is applied on this data set with default values for all parameters. A breakpoint is inserted at this step so that you can have a look at the results of the K-Medoids operator. You can see that two new attributes are created by the K-Medoids operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. As parameter k was set to 2, only two clusters are possible. That is why each example is assigned to either ‘cluster_0’ or ‘cluster_1’. A cluster model is also delivered through the cluster model output port. It has information regarding the clustering performed. Under the Folder View you can see members of each cluster in folder format and under the Centroid Table and Centroid Plot View tabs information regarding centroids.

The Item Distribution Performance operator is applied to measure the performance of this clustering model on the basis of how well the examples are distributed over the clusters. The cluster model produced by the K-Medoids operator is provided as input to the Item Distribution Performance operator which evaluates the performance of this model and delivers a performance vector that has performance measured on the basis of example distribution. The resultant performance vector can be seen in the results workspace.

# Map Clustering on Labels




This operator converts the cluster attribute into a prediction attribute.

## Description

The Map Clustering on Labels operator expects a clustered ExampleSet and a cluster model as input. Using these inputs, it estimates a mapping between the given clustering and prediction. It adjusts the given clusters with the given labels and so estimates the best fitting pairs. The resultant ExampleSet has a prediction attribute which is derived from the cluster attribute.

## Input Ports

**example set *(exa)*** This input port expects a clustered ExampleSet. It is the output of the K-Means operator in the attached Example Process.

**cluster model *(clu)*** This input port expects a cluster model. It is the output of the K-Means operator in the attached Example Process.

## Output Ports

**example set *(exa)*** The prediction attribute is derived from the cluster attribute and the resultant ExampleSet is delivered through this port.

**cluster model *(clu)*** The cluster model that was given as input is passed without any modifications to the output through this port. This is usually used to reuse the same cluster model in further operators or to view the cluster model in the Results Workspace.

## Tutorial Processes

### Introduction to the Map Clustering on Labels operator

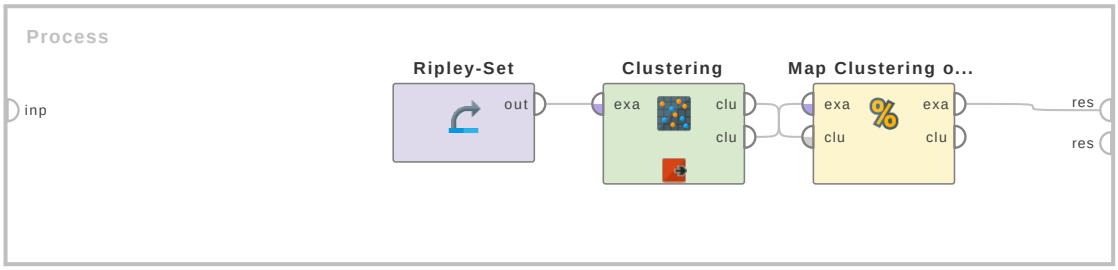


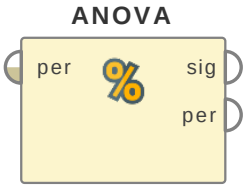

Figure 6.24: Tutorial process ‘Introduction to the Map Clustering on Labels operator’.

The ‘Ripley-Set’ data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters. Besides the label attribute the ‘Ripley-Set’ has two real attributes; ‘att1’ and ‘att2’. The K-Means operator is applied on this data set with default values for all parameters. Run the process and you will see that two new attributes are created by the K-Means operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. As parameter k was set to 2, only two clusters are possible. That is why each example is assigned to either ‘cluster_0’ or ‘cluster_1’

This clustered ExampleSet and cluster model are provided as input to the Map Clustering on Labels operator. The resultant ExampleSet can be seen it the Results Workspace. You can see that the ExampleSet has a prediction attribute now. You can also observe that the values of this attribute have been derived from the cluster attribute.

## 6.1.3 Significance Tests

### ANOVA




This operator is used for comparison of performance vectors. It performs an analysis of variance (ANOVA) test to determine the probability for the null hypothesis i.e. 'the actual means are the same'.

#### Description

ANalysis Of VAriance (ANOVA) is a statistical model in which the observed variance in a particular variable is partitioned into components attributable to different sources of variation. In its simplest form, ANOVA provides a statistical test of whether or not the means of several groups are all equal, and therefore generalizes t-test to more than two groups. Doing multiple two-sample t-tests would result in an increased chance of committing a type I error. For this reason, ANOVA is useful in comparing two, three, or more means. 'False positive' or Type I error is defined as the probability that a decision to reject the null hypothesis will be made when it is in fact true and should not have been rejected. RapidMiner provides the T-Test operator for performing the t-test. Paired t-test is a test of the null hypothesis that the difference between two responses measured on the same statistical unit has a mean value of zero.

#### Differentiation

- **T-Test** Doing multiple two-sample t-tests would result in an increased chance of committing a type I error. For this reason, ANOVA is useful in comparing two, three, or more means. See page 802 for details.

#### Input Ports

**performance *(per)*** This operator expects performance vectors as input it can have multiple inputs. When one input is connected, another *performance* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The performance vector supplied at the first *input* port of this operator is available at the first *performance* output port of the operator.

#### Output Ports

**significance *(sig)*** The given performance vectors are compared and the result of the significance test is delivered through this port.

**performance *(per)*** This operator can have multiple *performance* output ports. When one output is connected, another *performance* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The performance vector delivered at first *performance* input port of this operator is delivered at the first *performance* output port of the operator.

## Parameters

**alpha** ***(real)*** This parameter specifies the probability threshold which determines if differences are considered as significant. If a test of significance gives a p-value lower than the significance level *alpha*, the null hypothesis is rejected. It is important to understand that the null hypothesis can never be proven. A set of data can only reject a null hypothesis or fail to reject it. For example, if comparison of two groups reveals no statistically significant difference between the two, it does not mean that there is no difference in reality. It only means that there is not enough evidence to reject the null hypothesis (in other words, the experiment fails to reject the null hypothesis).

## Related Documents

- **T-Test** (page 802)

## Tutorial Processes

### Comparison of performance vectors using statistical significance tests

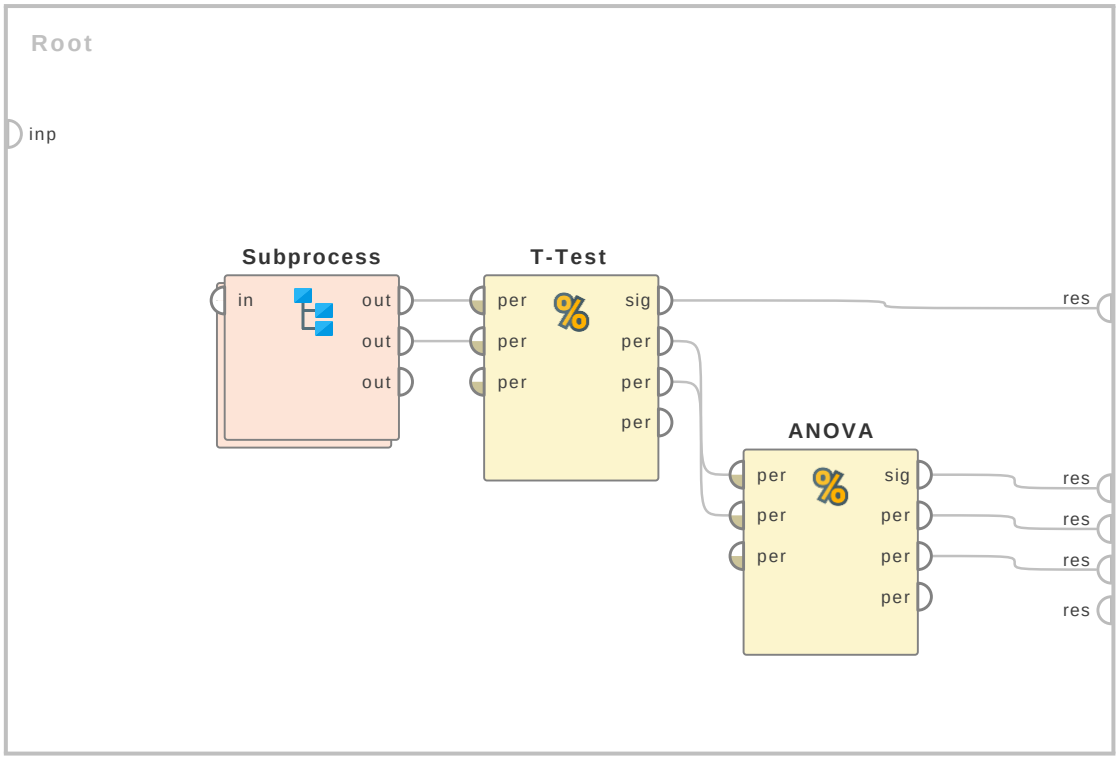



Figure 6.25: Tutorial process 'Comparison of performance vectors using statistical significance tests'.

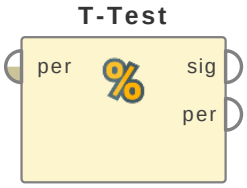
Many RapidMiner operators can be used to estimate the performance of a learner or a preprocessing step etc. The result of these validation operators is a performance vector which collects

the values of a set of performance criteria. For each criterion, the mean value and standard deviation are given. The question is how these performance vectors can be compared? Statistical significance tests like ANOVA or T-Test can be used to calculate the probability that the actual mean values are different. This Example Process performs exactly the same task.

This Example Process starts with a Subprocess operator which provides two performance vectors as output. Have a look at the inner operators of the Subprocess operator. The Generate Data operator is used for generating an ExampleSet. The Multiply operator is used for producing multiple copies of this ExampleSet. X-Validation operators are applied on both copies of the ExampleSet. The first X-Validation operator uses the Support Vector Machine (LibSVM) operator whereas the second X-Validation operator uses the Linear Regression operator in the training subprocess. The resultant performance vectors are the output of the Subprocess operator.

These performance vectors are compared using the T-Test and ANOVA operators respectively. The performance vectors and the results of the significance tests are connected to the result ports of the process and they can be viewed in the Results Workspace. Run the process and compare the results. The probabilities for a significant difference are equal since only two performance vectors were created. In this case the SVM is probably better suited for the data set at hand since the actual mean values are probably different. The SVM is considered better because its p-values is smaller than alpha which indicates a probably significant difference between the actual mean values.

# T-Test




This operator is used for comparison of performance vectors. This operator performs a t-test to determine the probability for the null hypothesis i.e. 'the actual means are the same'.

## Description

The T-Test operator determines if the null hypothesis (i.e. all actual mean values are the same) holds for the given performance vectors. This operator uses a simple paired t-test to determine the probability that the null hypothesis is wrong. Since a t-test can only be applied on two performance vectors this test will be applied to all possible pairs. The result is a significance matrix.

Paired t-test is a test of the null hypothesis that the difference between two responses measured on the same statistical unit has a mean value of zero. For example, suppose we measure the size of a cancer patient's tumor before and after a treatment. If the treatment is effective, we expect the tumor size for many of the patients to be smaller following the treatment. This is often referred to as the 'paired' or 'repeated measures' t-test.

In case of this operator the dependent samples (or 'paired') t-tests consist of a pair of performance vectors. Doing multiple paired t-tests would result in an increased chance of committing a type I error. 'False positive' or Type I error is defined as the probability that a decision to reject the null hypothesis will be made when it is in fact true and should not have been rejected. It is recommended to apply an additional ANOVA test to determine if the null hypothesis is wrong at all. Please use the ANOVA operator for performing the ANOVA test.

## Differentiation

- **ANOVA** Doing multiple two-sample t-tests would result in an increased chance of committing a type I error. For this reason, ANOVA is useful in comparing two, three, or more means. See page 799 for details.

## Input Ports

**performance *(per)*** This operator expects performance vectors as input and can have multiple inputs. When one input is connected, another *performance* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The performance vector supplied at the first *input* port of this operator is available at the first *performance* output port of the operator.

## Output Ports

**significance *(sig)*** The given performance vectors are compared and the result of the significance test is delivered through this port.

**performance *(per)*** This operator can have multiple *performance* output ports. When one output is connected, another *performance* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The performance vector delivered at the first *performance* input port of this operator is delivered at the first *performance* output port of the operator.

## Parameters

**alpha** ***(real)*** This parameter specifies the probability threshold which determines if differences are considered as significant. If a test of significance gives a p-value lower than the significance level *alpha*, the null hypothesis is rejected. It is important to understand that the null hypothesis can never be proven. A set of data can only reject a null hypothesis or fail to reject it. For example, if comparison of two groups reveals no statistically significant difference between the two, it does not mean that there is no difference in reality. It only means that there is not enough evidence to reject the null hypothesis (in other words, the experiment fails to reject the null hypothesis).

## Related Documents

- **ANOVA** (page 799)

## Tutorial Processes

### Comparison of performance vectors using statistical significance tests

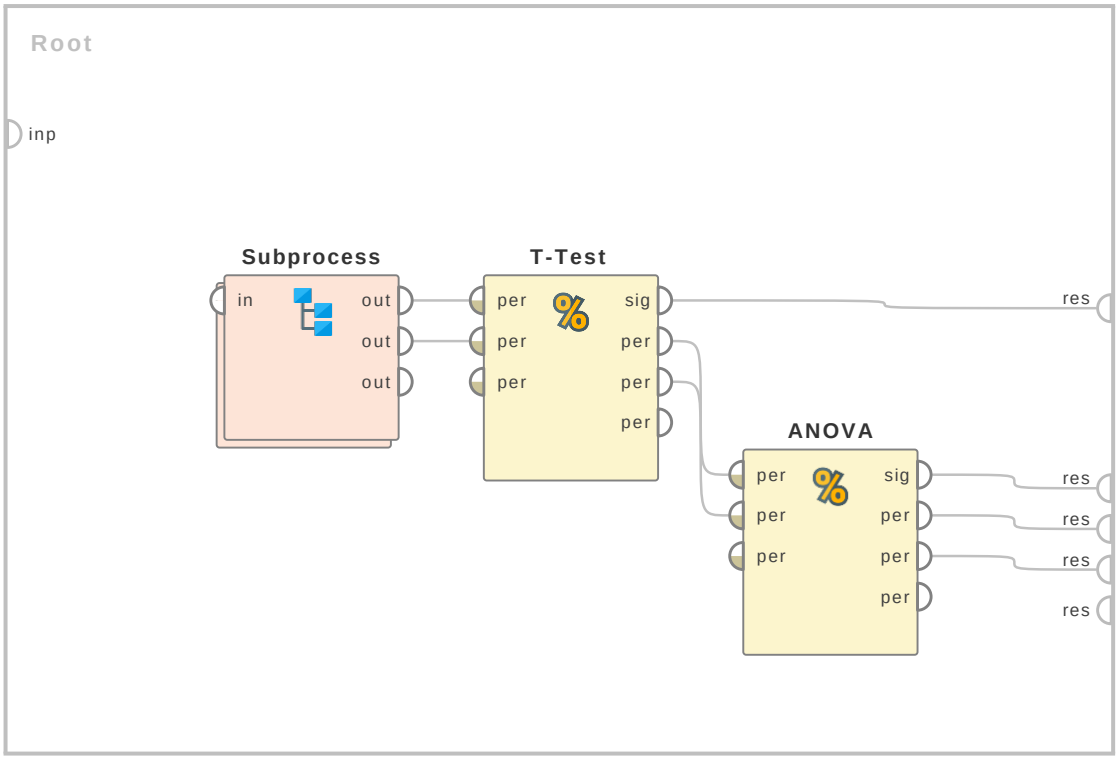



Figure 6.26: Tutorial process 'Comparison of performance vectors using statistical significance tests'.

Many RapidMiner operators can be used to estimate the performance of a learner or a preprocessing step etc. The result of these validation operators is a performance vector which collects

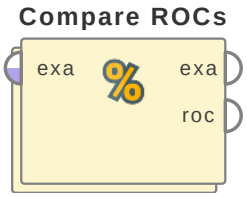
the values of a set of performance criteria. For each criterion, the mean value and standard deviation are given. The question is how can these performance vectors be compared? Statistical significance tests like ANOVA or T-Test can be used to calculate the probability that the actual mean values are different. This Example Process performs exactly the same task.

This Example Process starts with a Subprocess operator which provides two performance vectors as output. Have a look at the inner operators of the Subprocess operator. The Generate Data operator is used for generating an ExampleSet. The Multiply operator is used for producing multiple copies of this ExampleSet. X-Validation operators are applied on both copies of the ExampleSet. The first X-Validation operator uses the Support Vector Machine (LibSVM) operator whereas the second X-Validation operator uses the Linear Regression operator in the training subprocess. The resultant performance vectors are the output of the Subprocess operator.

These performance vectors are compared using the T-Test and ANOVA operators respectively. The performance vectors and the results of the significance tests are connected to the result ports of the process and they can be viewed in the Results Workspace. Run the process and compare the results. The probabilities for a significant difference are equal since only two performance vectors were created. In this case the SVM is probably better suited for the data set at hand since the actual mean values are probably different. SVM is considered better because its p-value is smaller than alpha which indicates a probably significant difference between the actual mean values.

# 6.2 Visual

## Compare ROCs




This operator generates ROC charts for the models created by the learners in its subprocess and plots all the charts in the same plotter for comparison.

### Description

The Compare ROCs operator is a nested operator i.e. it has a subprocess. The operators in the subprocess must produce a model. This operator calculates ROC curves for all these models. All the ROC curves are plotted together in the same plotter.

The comparison is based on the average values of a k-fold cross validation. Please study the documentation of the Cross Validation operator for more information about cross validation. Alternatively, this operator can use an internal split into a test and a training set from the given data set in this case the operator behaves like the Split Validation operator. Please note that any former predicted label of the given ExampleSet will be removed during the application of this operator.

ROC curve is a graphical plot of the sensitivity, or true positive rate, vs. false positive rate (one minus the specificity or true negative rate), for a binary classifier system as its discrimination threshold is varied. The ROC can also be represented equivalently by plotting the fraction of true positives out of the positives (TPR = true positive rate) vs. the fraction of false positives out of the negatives (FPR = false positive rate).

ROC curves are calculated by first ordering the classified examples by confidence. Afterwards all the examples are taken into account with decreasing confidence to plot the false positive rate on the x-axis and the true positive rate on the y-axis. With optimistic, neutral and pessimistic there are three possibilities to calculate ROC curves. If there is more than one example for a confidence with optimistic ROC calculation the correct classified examples are taken into account before looking at the false classification. With pessimistic calculation it is the other way round: wrong classifications are taken into account before looking at correct classifications. Neutral calculation is a mix of both calculation methods described above. Here correct and false classifications are taken into account alternately. If there are no examples with equal confidence or all examples with equal confidence are assigned to the same class the optimistic, neutral and pessimistic ROC curves will be the same.

### Input Ports

**example set *(exa)*** This input port expects an ExampleSet with binominal label. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

### Output Ports

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**rocComparison** ***(roc)*** The ROC curves for all the models are delivered from this port. All the ROC curves are plotted together in the same plotter.

## Parameters

**number of folds** ***(integer)*** This parameter specifies the number of folds to use for the cross validation evaluation. If this parameter is set to -1 this operator uses split ratio and behaves like the Split Validation operator.

**split ratio** ***(real)*** This parameter specifies the relative size of the training set. It should be between 1 and 0, where 1 means that the entire ExampleSet will be used as training set.

**sampling type** ***(selection)*** Several types of sampling can be used for building the subsets. Following options are available:

- **Linear sampling** Linear sampling simply divides the ExampleSet into partitions without changing the order of the examples i.e. subsets with consecutive examples are created.
- **Shuffled sampling** Shuffled sampling builds random subsets of the ExampleSet. Examples are chosen randomly for making subsets.
- **Stratified sampling** Stratified sampling builds random subsets and ensures that the class distribution in the subsets is the same as in the whole ExampleSet. For example in the case of a binominal classification, Stratified sampling builds random subsets so that each subset contains roughly the same proportions of the two values of class *labels*.

**use local random seed** ***(boolean)*** This parameter indicates if a *local random seed* should be used for randomizing examples of a subset. Using the same value of*local random seed* will produce the same subsets. Changing the value of this parameter changes the way examples are randomized, thus subsets will have a different set of examples. This parameter is only available if Shuffled or Stratified sampling is selected. It is not available for Linear sampling because it requires no randomization, examples are selected in sequence.

**local random seed** ***(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

**use example weights** ***(boolean)*** This parameter indicates if example weights should be considered. If this parameter is not set to true then weight 1 is used for each example.

**roc bias** ***(selection)*** This parameter determines how the ROC are evaluated i.e. correct predictions are counted first, last, or alternately. ROC curves are calculated by first ordering the classified examples by confidence. Afterwards all the examples are taken into account with decreasing confidence to plot the false positive rate on the x-axis and the true positive rate on the y-axis. With optimistic, neutral and pessimistic there are three possibilities to calculate ROC curves. If there are no examples with equal confidence or all examples with equal confidence are assigned to the same class the optimistic, neutral and pessimistic ROC curves will be the same.

- **optimistic** If there is more than one example for a confidence with optimistic ROC calculation the correct classified examples are taken into account before looking at the false classification.
- **pessimistic** With pessimistic calculation wrong classifications are taken into account before looking at correct classifications.

- **neutral** Neutral calculation is a mix of both optimistic and pessimistic calculation methods. Here correct and false classifications are taken into account alternately.

## Tutorial Processes

### Comparing different classifiers graphically by ROC curves

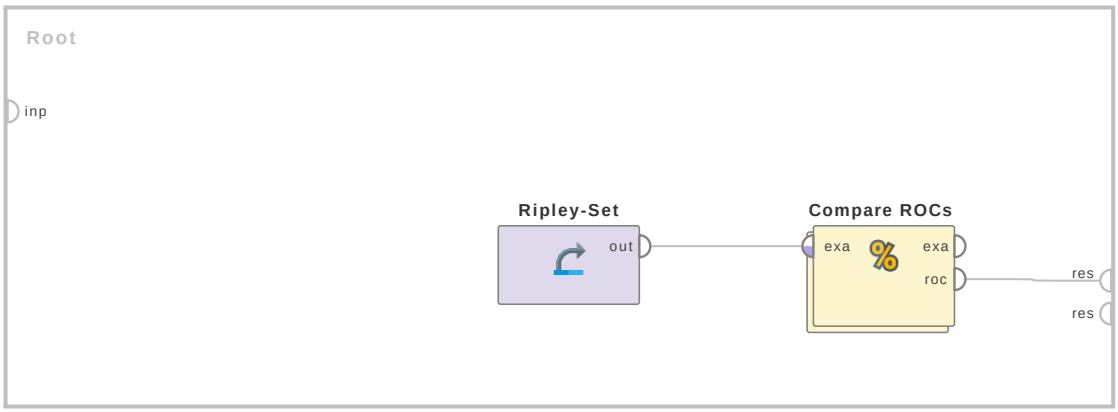
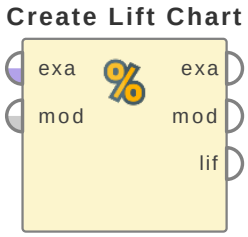



Figure 6.27: Tutorial process 'Comparing different classifiers graphically by ROC curves'.

This process shows how several different classifiers could be graphically compared by means of multiple ROC curves. The 'Ripley-Set' data set is loaded using the Retrieve operator. The Compare ROCs operator is applied on it. Have a look at the subprocess of the Compare ROCs operator. You can see that three different learners are applied i.e. Naive Bayes, Rule Induction and Decision Tree. The resultant models are connected to the outputs of the subprocess. The Compare ROCs operator calculates ROC curves for all these models. All the ROC curves are plotted together in the same plotter which can be seen in the Results Workspace.

# Create Lift Chart




This operator generates a lift chart for the given model and ExampleSet based on the discretized confidences and a Pareto chart.

## Description

The Create Lift Chart operator creates a lift chart based on a Pareto plot for the discretized confidence values of the given ExampleSet and model. The model is applied on the ExampleSet and a lift chart is produced afterwards. Please note that any predicted label of the given ExampleSet will be removed during the application of this operator. In order to produce reliable results, this operator must be applied on data that has not been used to build the model, otherwise the resulting plot will be too optimistic.

The lift chart measures the effectiveness of models by calculating the ratio between the result obtained with a model and the result obtained without a model. The result obtained without a model is based on randomly selected records.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Generate Direct Mailing Data operator in the attached Example Process. The output of other operators can also be used as input.

**model *(mod)*** This input port expects a model. It is the output of the Naive Bayes operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**model *(mod)*** The model that was given as input is passed without changing to the output through this port. This is usually used to reuse the same model in further operators or to view the model in the Results Workspace.

**lift pareto chart *(lif)*** For the given model and ExampleSet a lift chart is generated based on the discretized confidences and a Pareto chart. This lift chart is delivered through this port.

## Parameters

**target class *(string)*** This parameter indicates the target class for which the lift chart should be produced.

**binning type *(selection)*** This parameter indicates the binning type of the confidences.

**number of bins *(integer)*** This parameter specifies the number of bins the confidence should be discretized into. This parameter is only available when the *binning type* parameter is set to 'simple' or 'frequency'.

**size of bins *(integer)*** This parameter specifies the number of examples that each bin should contain when the confidence is discretized. This parameter is only available when the *binning type* parameter is set to 'absolute'.

**automatic number of digits *(boolean)*** This parameter indicates if the number of digits should be automatically determined for the range names.

**number of digits *(integer)*** This parameter specifies the minimum number of digits to be used for the interval names. If this parameter is set to -1 then the minimal number is determined automatically. This parameter is only available when the *automatic number of digits* parameter is set to false.

**show bar labels *(boolean)*** This parameter indicates if the bars should display the size of the bin together with the amount of the target class in the corresponding bin.

**show cumulative labels *(boolean)*** This parameter indicates if the cumulative line plot should display the cumulative sizes of the bins together with the cumulative amount of the target class in the corresponding bins.

**rotate labels *(boolean)*** This parameter indicates if the labels of the bins should be rotated.

## Tutorial Processes

### Creating lift chart for direct mailing data

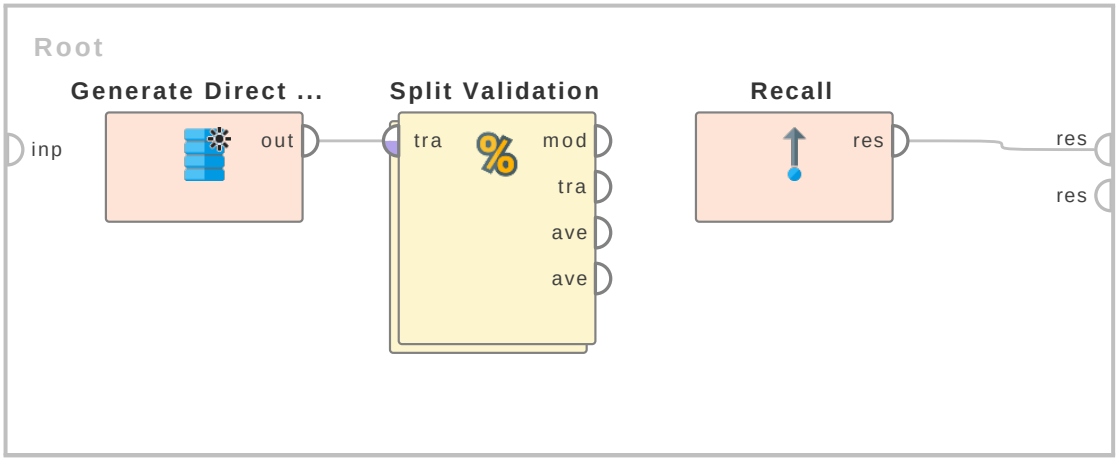


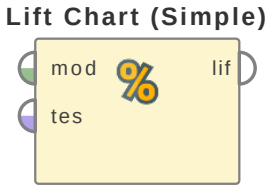

Figure 6.28: Tutorial process 'Creating lift chart for direct mailing data'.

The Direct Mailing Data operator is used for generating an ExampleSet with 10000 examples. The Split Validation operator is applied on this ExampleSet. The split ratio parameter is set to 0.7 and the sampling type parameter is set to 'shuffled sampling'. Here is an explanation of what happens inside the Split Validation operator.

The Split Validation operator provides a training data set through the training port of the training subprocess. This training data set is used as input for the Naive Bayes operator. Thus the Naive Bayes classification model is trained on this training data set.The Naive Bayes operator provides the Naive Bayes classification model as its output. This model is connected to the model port of the training subprocess.The Naive Bayes model that was provided at the model port of the training subprocess is delivered by the Split Validation operator at the model port of the testing subprocess. This model is provided as input at the model port of the Create Lift Chart operator.The Split validation operator provides the testing data set through the test set port of the testing subprocess. This testing data set is provided as input to the Create Lift Chart operator.The Create Lift Chart operator generates a lift chart for the given model and ExampleSet based on the discretized confidences and a Pareto chart. The lift chart is provided to the Remember operator to store it in the object store.The Apply Model operator is provided with the testing data set and the model. The Apply Model operator applies the model on the testing data set and the resultant labeled data set is delivered as output. This labeled data set is provided as input to the Performance operator.The Performance operator evaluates the statistical performance of the model through the given labeled data set and generates a performance vector which holds information about various performance criteria.

Outside the Split Validation operator, the Recall operator is used for fetching the lift chart from the object store. The lift chart is delivered to the output and it can be seen in the Results Workspace.

# Lift Chart (Simple)




This operator creates a lift chart which shows how much better the model performs for each confidence segment than random guessing.

## Description

A lift chart shows how much better a machine learning model performs compared with a random guess. It also shows you the point at which the predictions become less useful. This is in particular useful if you can optimize for a cost-benefit ration as it often happens for marketing-related use cases. For example, a lift chart can show you that to reach 80% of your respondents you only need to reach out to 30% of your total address base.

The lift chart shows you 10 bins for your test data. Each bin is filled with decreasing confidence of the model for the target class. That means that the examples with highest confidence values are in the first bin, then in the second, and so on. The chart consists of two parts. The bars show you the correct percentage for the target class. For example, if the first bar in the lift chart shows 95%, this means that 95% of all examples in this confidence bin are actually from the desired target class.

The second part of the chart is a line which shows you the cumulative coverage of the target class if you would consider only examples of at least the confidence of the corresponding bar or higher. A value of 60% at the third bar for example means that you covered 60% of the desired target class at that point. But the third bar only represents 30% of your total population. That means that this model would correctly identify 60% of the target with only using 30% of the total population (the 30% with the highest confidence for this class). In contrast to this, a random model would only achieve 30% of the target class.

## Input Ports

**model *(mod)*** This input port expects a prediction model.

**test data *(tes)*** The test data to create the lift chart for. Needs a label attribute to compare with model predictions.

## Output Ports

**lift chart *(lif)*** The lift chart for the given test data.

## Tutorial Processes

### Lift Chart for Naive Bayes on Titanic

This process creates a model on the Titanic data set. It first divides the data into a training and testing part. The model is built on the training data. It is then delivered together with the test data to the Lift Chart operator. Please note that you need to specify which class you are interested in. You can do this in the parameters of this operator.

Examining the lift chart, we can see that you can correctly identify 47% of all survivors while you are only looking at the first 20% of the passengers.

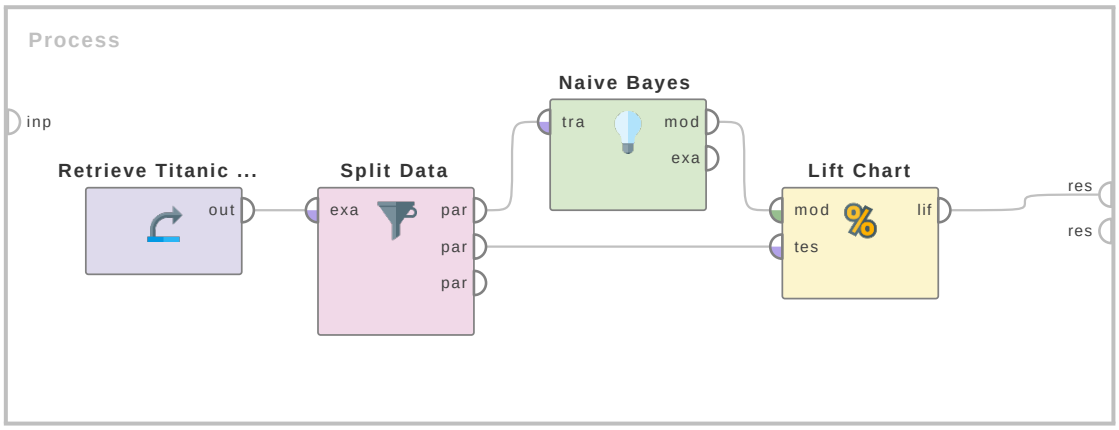
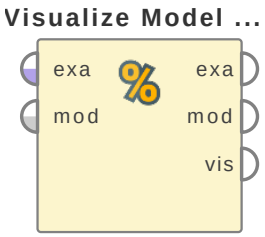



Figure 6.29: Tutorial process 'Lift Chart for Naive Bayes on Titanic'.

# Visualize Model by SOM




This operator generates a SOM plot (by transforming arbitrary number of dimensions to two) of the given ExampleSet and colorizes the landscape with the predictions of the given model.

## Description

The Visualize Model by SOM operator provides the visualization of arbitrary models with help of the dimensionality reduction via SOM of both the data set and the given model. A self-organizing map (SOM) or self-organizing feature map (SOFM) is a type of artificial neural network that is trained using unsupervised learning to produce a low-dimensional (typically two-dimensional), discretized representation of the input space of the training samples, called a map. Self-organizing maps are different from other artificial neural networks in the sense that they use a neighborhood function to preserve the topological properties of the input space. This makes SOMs useful for visualizing low-dimensional views of high-dimensional data, akin to multidimensional scaling. The model was first described as an artificial neural network by Teuvo Kohonen, and is sometimes called a Kohonen map.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Decision Tree operator in the attached Example Process. The output of other operators can also be used as input.

**model *(mod)*** This input port expects a model. It is the output of the Decision Tree operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**model *(mod)*** The model that was given as input is passed without changing to the output through this port. This is usually used to reuse the same model in further operators or to view the model in the Results Workspace.

**visualization *(vis)*** The SOM visualization is returned through this port.

## Tutorial Processes

### Visualizing the Decision Tree by SOM

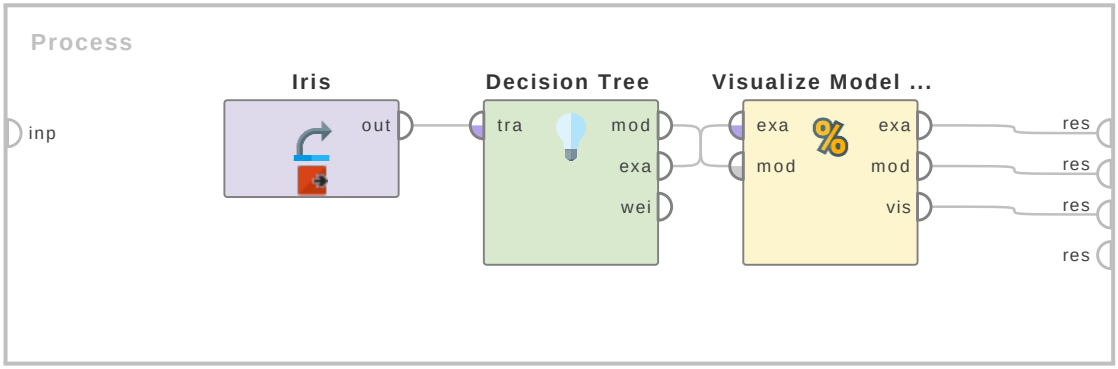
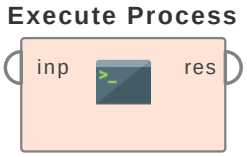



Figure 6.30: Tutorial process ‘Visualizing the Decision Tree by SOM’.

The ‘Iris’ data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. The Decision Tree operator is applied on this ExampleSet and the resultant model is provided as input model to the Visualize Model by SOM operator. The original ExampleSet is also provided as input. The ExampleSet, model and visualization returned by the Visualize Model by SOM operator are connected to the output and can be viewed in the Results Workspace.

# 7Utility

## Execute Process




This operator embeds a complete process (previously written into a file) into the current process.

### Description

This operator can be used to embed a complete process definition of a saved process into the current process definition. The saved process will be loaded and executed when the current process reaches this operator. Optionally, the input of this operator can be used as input for the embedded process. In both cases, the output of the saved process will be delivered as output of this operator. Please note that validation checks will not work for a process containing an operator of this type since the check cannot be performed without actually loading the process. The use of this operator can be easily understood by studying the attached Example Process.

### Input Ports

**input *(inp)*** The Execute Process operator can have multiple inputs. When one *input* port is connected, another *input* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *input* port of the Execute Process operator is available at the first *input* port of the embedded process. Don't forget to connect all inputs in correct order. Make sure that you have connected the right number of ports.

### Output Ports

**result *(res)*** The Execute Process operator can have multiple outputs. When one *result* port is connected, another *result* port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first *output* port of the embedded process is delivered at the first *result* port of the Execute Process operator. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports.

### Parameters

**process location** The location of the process to be embedded is provided here.

**use input *(boolean)*** This is an expert parameter. It indicates if the input of this operator should be used as input for the embedded process. This should always be set to true if you want to provide input to the embedded process through the current process.

**store output *(boolean)*** This is an expert parameter. It indicates if the operator output should be stored. This applies only if the context of the embedded process defines output locations.

**propagate metadata recursively *(boolean)*** This is an expert parameter. It determines whether meta data is propagated through the included process.

**cache process *(boolean)*** This is an expert parameter. It determines if the process should be loaded during execution. If it is checked, the process will not be loaded during the execution.

**macros** This is an expert parameter. It defines macros for this sub-process.

**fail for unknown macros** This is an expert parameter. It decides which macros you can define at the 'macros' list above. If checked, only macros defined in the embedded process' context can be defined in the 'macros' list above.

## Tutorial Processes

### The process to be used as an embedded process in the next Example Process

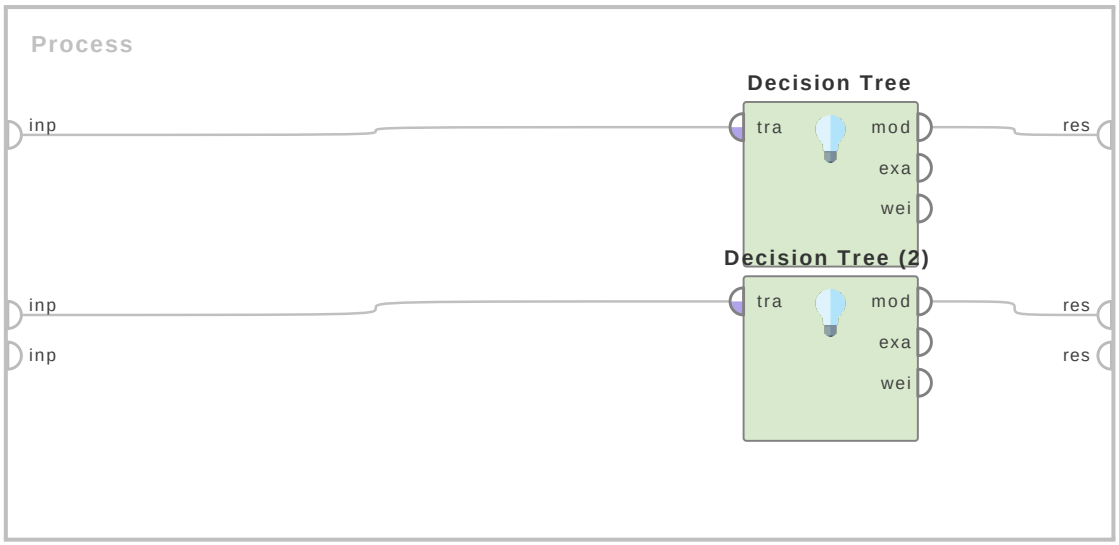



Figure 7.1: Tutorial process 'The process to be used as an embedded process in the next Example Process'.

This process does not use the Execute Process operator, rather it is used as an embedded process in the second Example Process . This process uses the Decision Tree operator twice. In both cases input is not provided to the operators in this process. The input ports of the operators are connected with the input ports of the process, thus these operators will receive inputs via another process as we shall see soon. Such a process cannot work at its own, because inputs are missing. Only the mod port is connected to the output in both operators. Always make sure that the input ports of this process are connected in the right way whenever you want this process to receive inputs from other processes.

### Executing an embedded process

The Execute Process operator is used in this process. The process location parameter supplies the location of the first Example Process. Make sure you provide the same location here that you

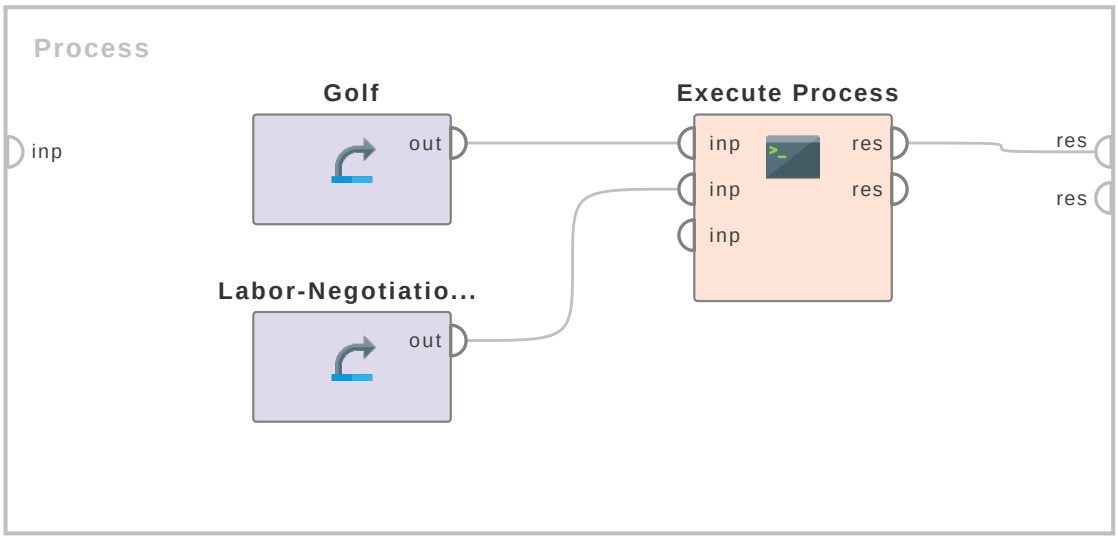
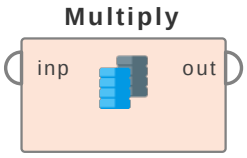



Figure 7.2: Tutorial process ‘Executing an embedded process’.

used to save the first Example Process. The Retrieve operator is used twice, first it loads the ‘Golf’ data set and then it is used to load the ‘Labor-Negotiations’ data set. These data sets are sent as input to the embedded process. The object connected to the first input port of the Execute Process operator is received at the first input port of the embedded process. Thus the ‘Golf’ data set is provided as input to the first Decision Tree operator. Similarly, the ‘Labor-negotiations’ data set is provided as input to the second Decision Tree operator. Passing input to the embedded process was possible because the use input parameter was checked. If you uncheck it and run the process again, you will get an error message. Outputs of the embedded process are delivered as outputs of the Execute Process operator in the current process. The order of outputs remains the same. The Decision Tree model of the Labor-negotiations data set is connected to the second res port of the first Example Process, thus it is available at the second res port of the Execute Process operator in the current process.

# Multiply




This Operator creates copies of a RapidMiner Object.

## Description

The Operator takes the RapidMiner Object from the input port and delivers copies of it to the output ports. Each connected port creates an independent copy. So changing one copy has no effect on other copies.

## Differentiation

Many Operators have an output port named *original* or *throughput*, which does not change the input. By chaining Operators to one or more original ports also copies of an object can be created. But such a layout can get complicated very fast. The Multiply Operator helps to better structure a RapidMiner Process.

## Input Ports

**input *(inp)*** The input that should be copied. It can be any RapidMiner Object.

## Output Ports

**output *(out)*** The copy of the input object. As one output port is connected, another output port is created for more copies. All ports deliver unchanged copies of the input object.

## Tutorial Processes

### Multiply an ExampleSet

In this tutorial Process the Multiply Operator creates two copies of the Titanic Example Set. All missing values of the first copy are replaced by the average value and then 100 random Examples are selected. The second copy is not modified. This illustrates that the copies are independent and changes on one copy are not applied on the other copies.

‘‘

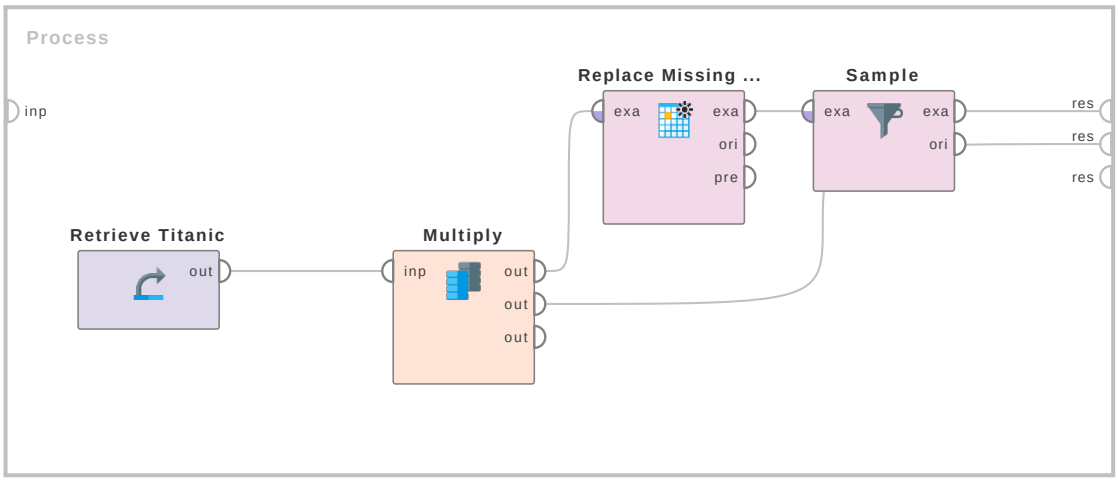
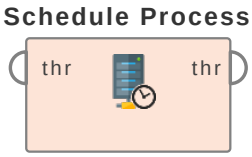



Figure 7.3: Tutorial process ‘Multiply an ExampleSet’.

# Schedule Process




This operator schedules a process on a RapidMiner Server with given input, output and macros.

## Description

This operator starts a process which is given as a repository location on a RapidMiner Server. The server instance of this repository will execute the process independently from the current process, so that an own process context is available. This context, i.e. the input and output of the process and the macros, can be defined as parameters as well.

Several different schedule modes are available which determine when the process will be started. The modes *now*, *once* and *offset* are used to start the process once at a given time, whereby *cron schedule* allows to execute the process periodically according to a given cron expression which can be easily created by the cron editor.

It is necessary that the repository of the process which will be scheduled is a repository of a RapidMiner Server instance. Although it is possible to schedule a process from a local RapidMiner instance on a remote server, it is not possible to execute several process within a local RapidMiner instance with this operator.

## Input Ports

**through *(thr)***

## Output Ports

**through *(thr)***

## Parameters

**process entry *(string)*** A valid path should be specified here in order to execute a process. This parameter references a process in a remote repository, i.e. a repository on a RapidMiner Server.

**process input *(menu)*** The process input which will be provided to the process which will be executed. This repository location has to be reachable by the process on the RapidMiner Server instance.

**process output *(menu)*** Indicates where the process output shall be stored. This repository location has to be reachable by the process on the RapidMiner Server instance.

**macros *(menu)*** The macros which will be accessible by the executed process.

**schedule mode *(selection)*** The mode defines the time when the selected process will be executed on the server.

- **now** The process will be executed immediately
- **once** The process will be executed at the date provided by the *date* parameter
- **offset** The process will be executed after a given offset provided by the *offset* and *time_unit* parameters
- **cron schedule** The process will be executed periodically given a *start* and end date and the *cron expression*

**date *(date)*** If the schedule mode *once* is selected this date parameter specifies the exact date and time when the process will be started.

**offset *(integer)*** The offset to specify for the schedule mode *offset*. Combined with the parameter *time_unit* this sets the time from now to the start of the process.

**time unit *(selection)*** For the *offset* mode this specifies the unit of time of the value defined in the *offset* parameter. So if the offset is set to 10, this parameter defines if the process will be started in 10 seconds, 10 minutes, 10 hours or 10 days.

**expression *(cron)*** If the schedule mode *cron schedule* is selected, the expression parameter offers a *cron* value which can be added by hand or via the cron editor. This allows to start the process periodically.

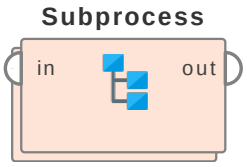
**set start date *(boolean)*** For a cron expression this parameter determines if a start date will be used. This start date specifies the time when the cron expression is activated to schedule a process.

**start date *(date)*** The start date which specifies the date to start the scheduling of a process via the cron expression defined in *expression*. This parameter is only in use if the *set_start_date* parameter is set to true.

**set end date *(boolean)*** For a cron expression this parameter determines if a end date will be used. This end date specifies the time until when the cron expression is activated to schedule a process.

**end date *(date)*** The end date which specifies the date until the cron expression will be used to schedule a process. This parameter is only in use if the *set_end_date* parameter is set to true.

## Subprocess




This operator introduces a process within a process. Whenever a Subprocess operator is reached during a process execution, first the entire subprocess is executed. Once the subprocess execution is complete, the flow is returned to the process (the parent process). A subprocess can be considered as a small unit of a process, like in process, all operators and combination of operators can be applied in a subprocess. That is why a subprocess can also be defined as a chain of operators that is subsequently applied.

### Description

Double click on the Subprocess operator to go inside the subprocess. The subprocess is then shown in the same Process View. To go back to the parent process, click the blue-colored up arrow button in the Process View toolbar. This works like files and folders work in operating systems. Subprocesses can have subprocesses in them just like folders can have folders in them. The order of execution in case of nested subprocesses is the same as a depth-first-search through a tree structure. When a Subprocess operator is reached, all operators inside it are executed and then the execution flow returns to the parent process and the operator that is located after the Subprocess operator (in the parent process) is executed. This description can be easily understood by studying the attached Example Process.

A subprocess can be considered as a simple operator chain which can have an arbitrary number of inner operators. The operators are subsequently applied and their output is used as input for the succeeding operators. The input of the Subprocess operator is used as input for the first operator in it and the output of the last operator in the subprocess is used as the output of the Subprocess operator. Subprocesses make a process more manageable but don't forget to connect all inputs and outputs in correct order. Also make sure that you have connected the right number of ports at all levels of the chain.

Subprocesses are useful in many ways. They give a structure to the entire process. Process complexity is reduced and they become easy to understand and modify. Many operators have a subprocess as their integral parts e.g. the X-Validation operator which is also shown in the attached Example Process. It should be noted that connecting the input of a Subprocess directly to its output without applying any operator in between or using an empty Subprocess gives no results.

### Input Ports

**input *(inp)*** The Subprocess operator can have multiple inputs. When one input is connected, another *input* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *input* port of the subprocess is available at the first *input* port of the nested chain (inside the subprocess). Subprocesses make a process more manageable but don't forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

### Output Ports

**output *(out)*** The Subprocess operator can have multiple outputs. When one output is connected, another *output* port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first *output* port

of the subprocess is delivered at the first *output* of the outer process. Subprocesses make a process more manageable but don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

## Tutorial Processes

### Using subprocesses to structure a process

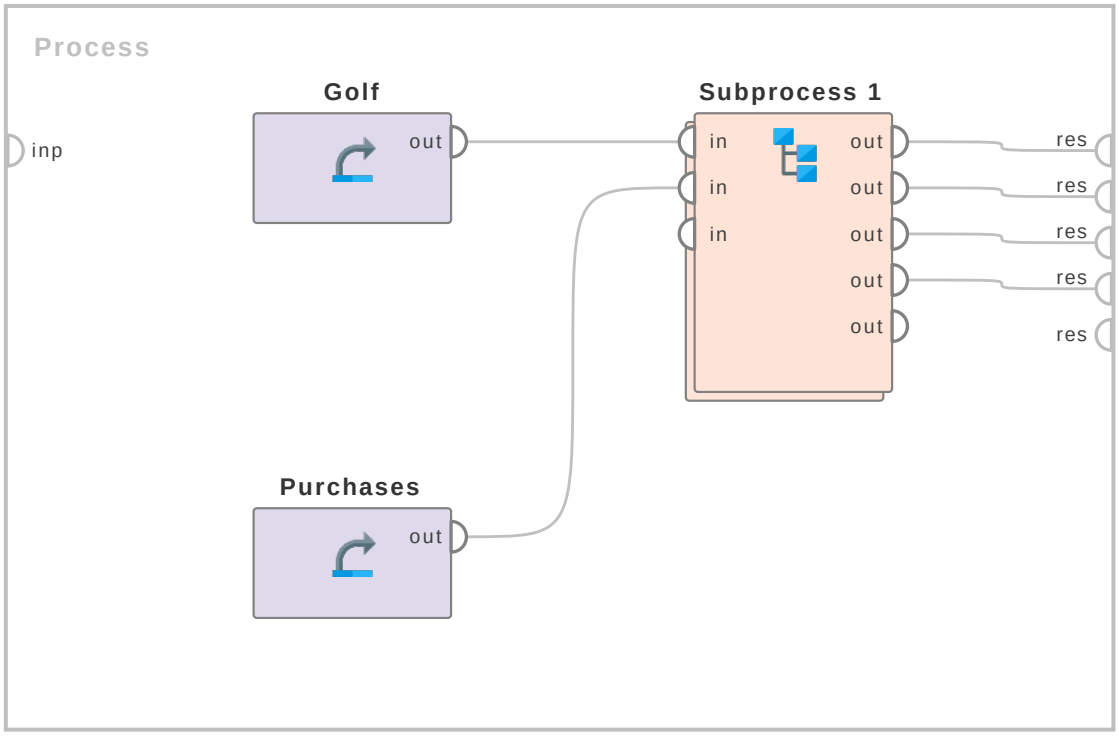



Figure 7.4: Tutorial process 'Using subprocesses to structure a process'.

The 'Golf' dataset is loaded using the Retrieve operator. It is attached to the first input of the Subprocess operator. Double click on the Subprocess operator to see what is inside this subprocess. The first input of the subprocess is attached with a Decision Tree operator. The output of the Decision Tree operator is given to the first output port. Now, go back to the main process. You will see that the first output port of the Subprocess operator is attached to the first result port. This explains the result 'Tree(decision tree(golf))' in the Results Workspace. This is how it works: The Golf data set enters the subprocess through the first input port, then the Decision Tree operator is applied on it in the subprocess, the resulting model is delivered to the results via the first output port of the subprocess.

During the main process, the Purchases data set is loaded using the Retrieve operator. It is attached to the second input port of the Subprocess operator. Double click on the Subprocess operator to see what is inside this subprocess. The second input port of the subprocess is attached directly to the second output port without applying any operator. Now, go back to the main process. You will see that the second output port of the Subprocess operator is attached to the second result port. But, as no operator is applied to the Purchases data set in the Subpro-

cess, it fails to produce any results (not even the result of the Retrieve operator is shown in the Results Workspace). This explains why we have three results in the Results Workspace despite the attachment of four outputs to the results ports in the main process.

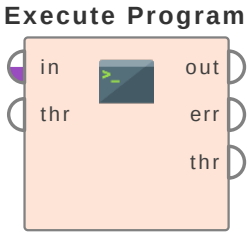
In the subprocess, the Iris data set is loaded using the Retrieve operator. It is connected to the Decision Tree operator and the resultant model is attached to the third output port of the subprocess, which is in turn attached to the third results port in the main process. This explains the result 'Tree (decision tree (Iris))' in the Results Workspace.

In the Subprocess, the Weighting data set is loaded using Retrieve operator. It is connected to the X-Validation operator and the resultant Performance Vector is attached to the forth output port of the Subprocess, which is in turn attached to the forth results port in the main process. This explains the result 'performanceVector (Performance)' in the Results Workspace. The X-Validation operator itself is composed of a subprocess; double click on the X-Validation operator and you will see the subprocess within this operator. Explanation of what is going on inside X-Validation would be a diversion here. This operator was added here just to show how various operators can be composed of a subprocess. To know more about the X-Validation operator you can read its description.

Note: This Example Process is just for highlighting different perspectives of Subprocess operator. It may not be very useful in real scenarios. The Example Process of Performance operator is also a good example of the usage of the Subprocess operator.

# 7.1 Scripting

## Execute Program




This operator simply executes a command in the shell of the underlying operating system. It can execute any system command or external program.

### Description

This operator executes a system command. The command and all its arguments are specified by the *command* parameter. Please note that the command is system dependent. The standard output stream of the process can be redirected to the log file by enabling the *log stdout* parameter. The standard error stream of the process can be redirected to the log file by enabling the *log stderr* parameter.

In Windows / MS DOS, simple commands should be preceded by 'cmd /c' call, e.g. 'cmd /c notepad'. Just writing 'notepad' in the *command* parameter will also work in case you are executing a program and not just a shell command. Then Windows opens a new shell, executes the command, and closes the shell again. However, Windows 7 may not open a new shell, it just executes the command. Another option would be to precede the command with 'cmd /c start' which opens the shell and keeps it open. The rest of the process will not be executed until the shell is closed by the user. Please study the attached Example Processes for more information.

CAUTION: Due to a Java bug on Windows / MS DOS the operator is only able to stop first level child processes. For example: When stopping the operator that was started with a command containing a preceeding 'cmd /c', only the direct child process (the shell) will be closed but processes started by these shell will still be running detached from RapidMiner.

The Java *ProcessBuilder* is used for building and executing the command. Characters that have special meaning on the shell e.g. the pipe symbol or brackets and braces do not have a special meaning to Java. Please note, that this Java method parses the string into tokens before it is executed. These tokens are not interpreted by a shell. If the desired command involves piping, redirection or other shell features, it is best to create a small shell script to handle this.

### Input Ports

**in *(in )*** A file object sent to this port will be piped to the standard input (stdin) of the process.

**through *(thr)*** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the Execute Program operator is available at the first *through* output port.

## Output Ports

**out *(out)*** If connected, the standard output stream (stdout) generated by this process will be delivered as a file object.

**err *(err)*** If connected, the standard error stream (stderr) generated by this process will be delivered as a file object.

**through *(thr)*** The objects that were given as input are passed without changing to the output through this port. It is not compulsory to connect this port to any other port, the command is executed even if this port is left without connections. The Execute Program operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Execute Program operator is delivered at the first *through* output port

## Parameters

**command *(string)*** This parameter specifies the command to be executed.

**log stdout *(boolean)*** If set to true, the *stdout* stream (standard output stream) of the command is redirected to the log file. Only available if *out* port is not connected.

**log stderr *(boolean)*** If set to true, the *stderr* stream (standard error stream) of the command is redirected to the log file. Only available if *err* port is not connected.

**working directory *(string)*** Defines the working directory for the command. If no working directory is defined the working directory of the current RapidMiner process is used.

**env variables *(list)*** Allows to set environment variables for the specified command. If an environment variable is defined multiple times, the last defined variable will be used.

## Tutorial Processes

### Introduction to the Execute Program operator

This Example Process uses the Execute Program operator to execute commands in the shell of Windows 7. Two Execute Program operators are used. The command parameter of the first Execute Program operator is set to 'cmd /c java -version'. The command parameter of the second Execute Program operator is set to 'cmd /c notepad'. When the process is executed, first the java version is described in the log window. Then the notepad is opened. The process waits for the notepad to close. The process proceeds when the notepad is closed by the user. Please note that setting the command parameter to just 'notepad' would have also worked here.

### Opening Internet Explorer by the Execute Program operator

This Example Process uses the Execute Program operator to open the Internet Explorer browser by using the shell commands of Windows 7. The command parameter of the Execute Program operator is set to 'cmd /c start C:\"Program Files"\"Internet Explorer"\"iexplore.exe"'. When the process is executed, the Internet Explorer browser opens. The process waits for the Internet Explorer browser to be closed by the user. The process proceeds when the Internet Explorer browser is closed.

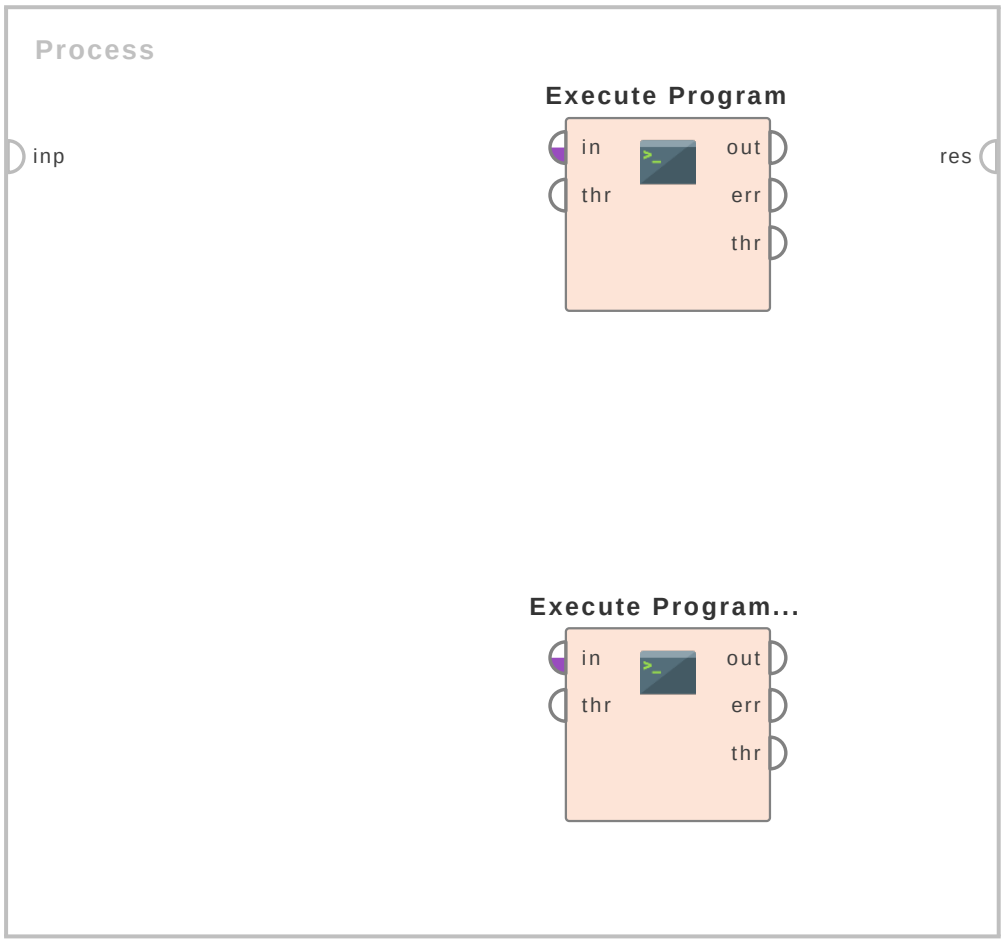



Figure 7.5: Tutorial process 'Introduction to the Execute Program operator'.

### Piping data through a shell command

This process demonstrates how data can be streamed into and out of the executed command. In particular, we open the Iris data set, write it as a CSV file into an in-memory File Object. This buffer is then passed to the Execute Program operator which executes the "sort" command. This sorts the input and returns it at the >out port. Another Read CSV operator parses the sorted output.

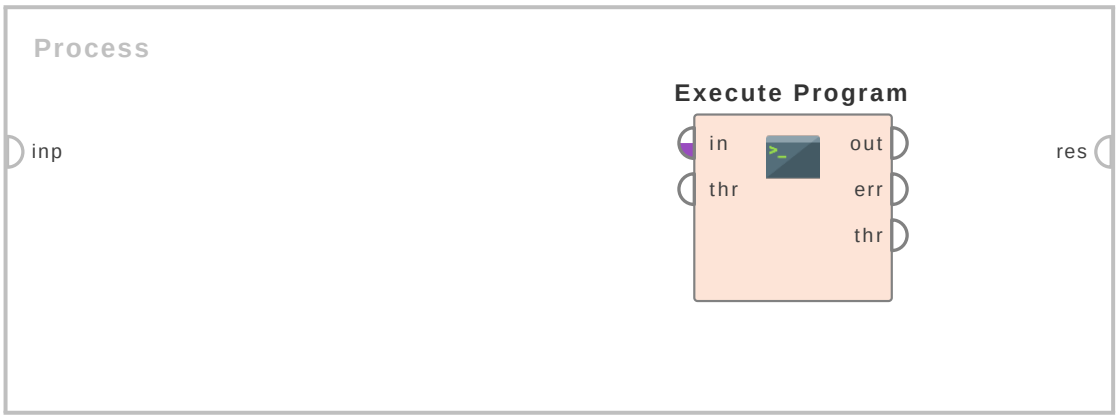



Figure 7.6: Tutorial process 'Opening Internet Explorer by the Execute Program operator'.

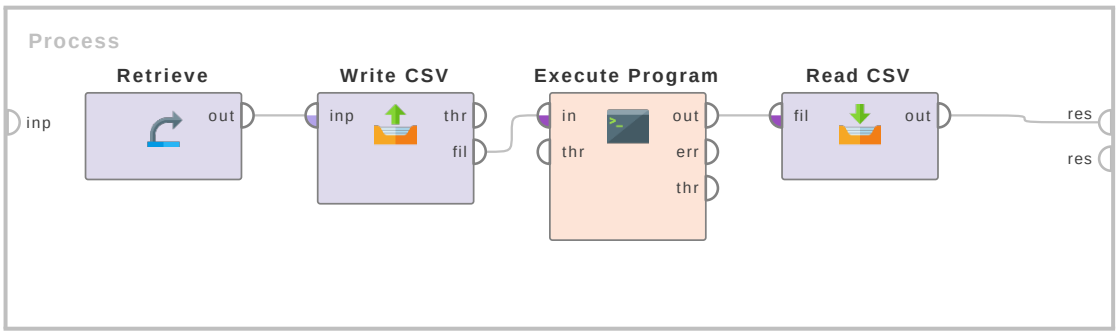
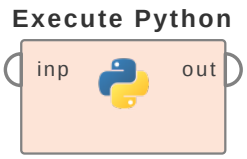



Figure 7.7: Tutorial process 'Piping data through a shell command'.

# Execute Python




Executes a Python script.

## Description

Before using this operator you need to specify the path to your Python installation under Tools -> Preferences -> Python Scripting. Your Python installation must include the pandas module since example sets get converted to pandas.DataFrames.

This operator executes the script specified as parameter. The arguments of the script correspond to the input ports, where example sets are converted to pandas.DataFrames. Analogously, the values returned by the script are delivered at the output ports of the operator, where pandas.DataFrames are converted to example sets.

The console output of Python is shown in the Log View (View -> Show View -> Log).

## Input Ports

**input *(inp)*** The Script operator can have multiple inputs. An input must be either an example set, a file object or a Python object which was generated by an 'Execute Python' operator.

## Output Ports

**output *(out)*** The Script operator can have multiple outputs. An output can be either an example set, a file object or a Python object generated by this operator.

## Parameters

**script *(text)*** The Python script to execute. Define a method with name 'rm_main' with as many arguments as connected input ports or alternatively a *args argument to use a dynamic number of attributes. The return values of the method 'rm_main' are delivered to the connected output ports. If the method returns a tuple then the single entries of the tuple are delivered to the output ports. Entries from the data type 'pandas.DataFrames' are converted to example sets; files are converted to File Objects, other Python objects are serialized and can be used by other 'Execute Python' operators or stored in the repository. Serialized Python objects have to be smaller than 2 GB.

If you pass an example set to your script through an input port, the meta data of the example set (types and roles) is available in the script. You can access it by reading the attribute rm_metadata of the associated pandas.DataFrame, in our example data.

data.rm_metadata is a dictionary from attribute names to a tuple of attribute type and attribute role.

You can influence the meta data of an example set that you return as a pandas.DataFrame by setting the attribute rm_metadata. If you don't specify attribute types in this dictionary, they will be determined using the data types in Python. You can specify your own roles or use the standard roles of RapidMiner like 'label'.

For more information about the meta data handling in a Python operator check the tutorial process 'Meta data handling' below.

## Tutorial Processes

### Clustering using Python

Random data is generated and then fed to the Python script. The script clusters the data in Python using as many clusters as are specified in the macro. The resulting ExampleSet contains the cluster in the 'cluster' attribute.

### Building a model and applying it using Python

This tutorial process uses the 'Execute Python' operators to first build a decision tree model using the 'Deals' data and then applying it to the 'Deals Testset' data. Before using the data, it the nominal values are converted to unique integers. The first Python scripting operator 'build model' builds the model and delivers it to its output port. The second Python scripting operator 'apply model' applies this model to the testset, adding a column called prediction. After specifying the 'label' and 'prediction' columns with 'Set Role', the result can be viewed.

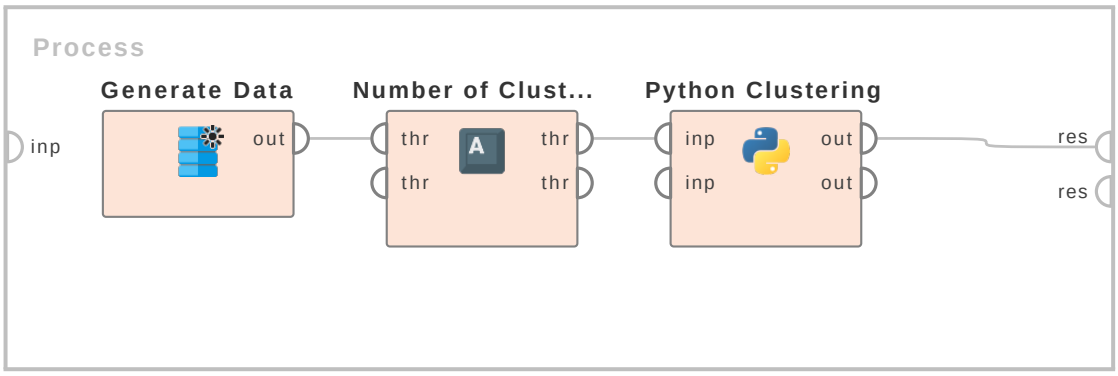



Figure 7.8: Tutorial process 'Clustering using Python'.

### Creating a plot using Python and storing it in your repository

This tutorial process uses the 'Execute Python' operator to first fetch example data, then create a plot and return both to the output ports. Please store the process in your repository. The data are shown as example set and the plot is stored in the repository as image.

### Reading an example set from a file using Python

This tutorial process uses the 'Execute Python' operator to save example data in a csv file. The second 'Execute Python' operator receives this file, reads the data and returns a part of the data to the output port. The result is an example set.

### Meta data handling

This tutorial process shows how to access the meta data of incoming example sets inside a 'Execute Python' operator. It also explains how to set the meta data for the outcoming example sets.

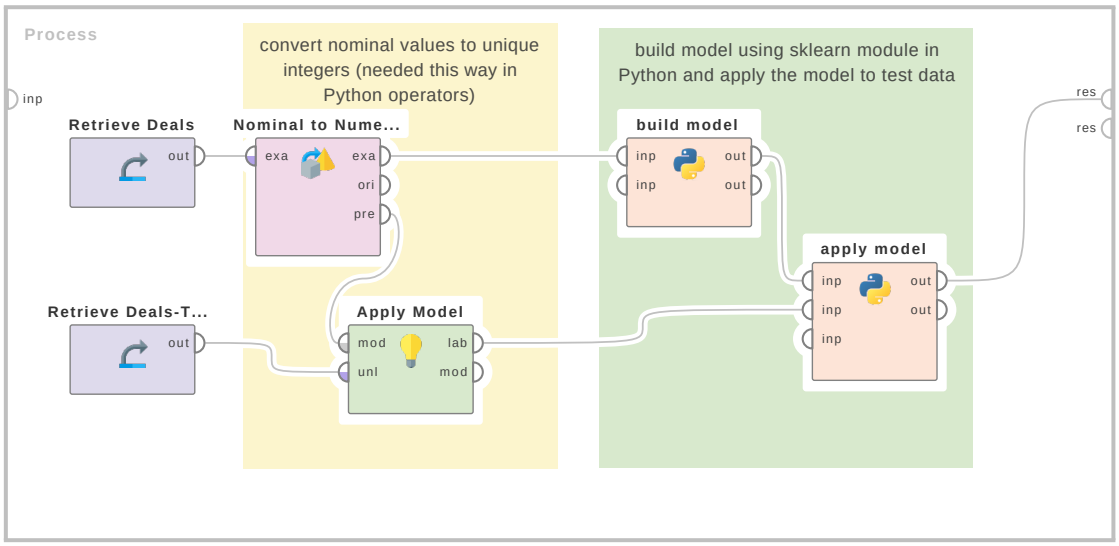
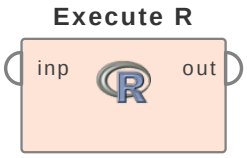



Figure 7.9: Tutorial process 'Building a model and applying it using Python'.

# Execute R




Executes a R script.

## Description

Before using this operator you need to specify the path to your R installation under Tools -> Preferences -> R Scripting. Your R installation has to include the 'data.table' package.

This operator executes the script specified as parameter. The arguments of the script correspond to the input ports, where example sets are converted to data frames. Analogously, the values returned by the script are delivered at the output ports of the operator, where data frames are converted to example sets.

The console output of R is shown in the Log View (View -> Show View -> Log).

## Input Ports

**input *(inp)*** The Script operator can have multiple inputs. An input must be either an example set, a file object or an R object which was generated by an 'Execute R' operator.

## Output Ports

**output *(out)*** The Script operator can have multiple outputs. An output can be either an example set, a file object or an R object generated by this operator.

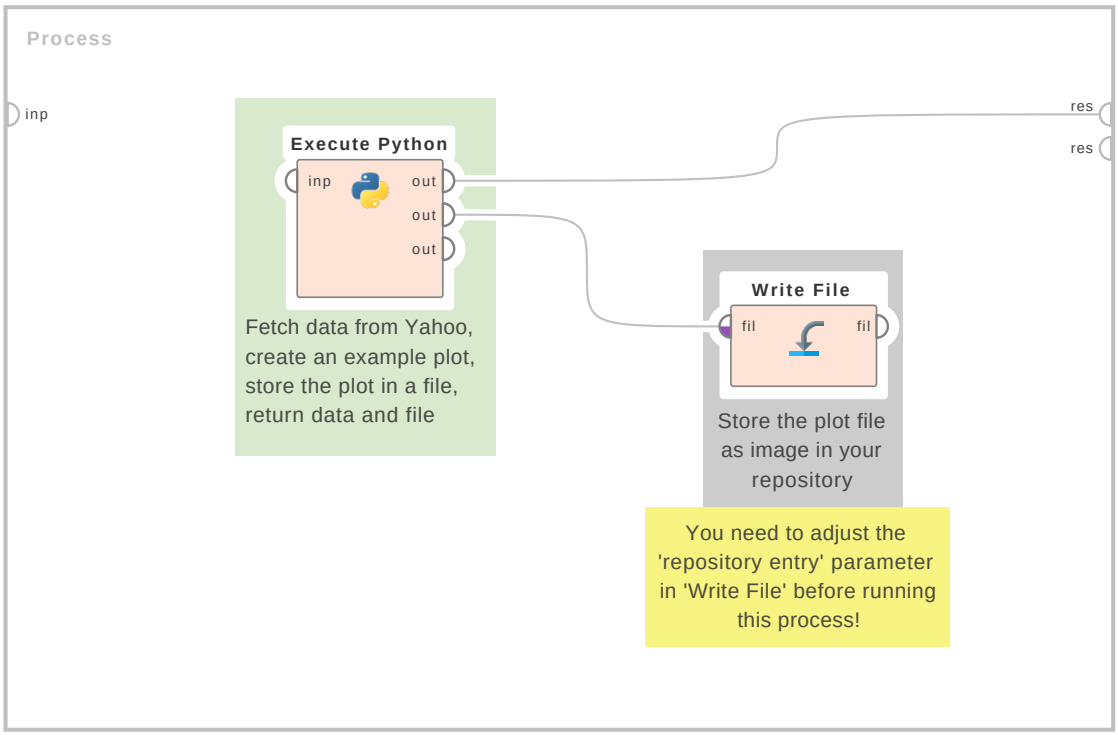



Figure 7.10: Tutorial process ‘Creating a plot using Python and storing it in your repository’.

## Parameters

**script** ***(text)*** The R script to execute. Define a function with name ‘rm_main’ with as many arguments as connected input ports or alternatively the ellipsis arguments (’...’) to use a dynamic number of attributes. The return values of the function ‘rm_main’ are delivered to the connected output ports. Entries from the data type ‘data frame’ are converted to example sets; files are converted to File Objects, other R objects are serialized and can be used by other ‘Execute R’ operators or stored in the repository. Serialized R objects have to be smaller than 2 GB.

If you pass an example set to your script through an input port, the meta data of the example set (types and roles) is available in the script. You can access it by the metaData list object in R. The names of top components in the list are identical to the arguments from the rm_main() function. Each component will contain the name of all attributes defined by that input argument and its type and role. To access or change a specific meta data entry use metaData$inputArgument$attributeName$type or metaData$inputArgument$attributeName$role. Please note that changes to the meta data have to be made with the ‘superassignment’ operator <<-.

For more information about the meta data handling in an R operator check the tutorial process ‘Meta data handling’ below.

## Tutorial Processes

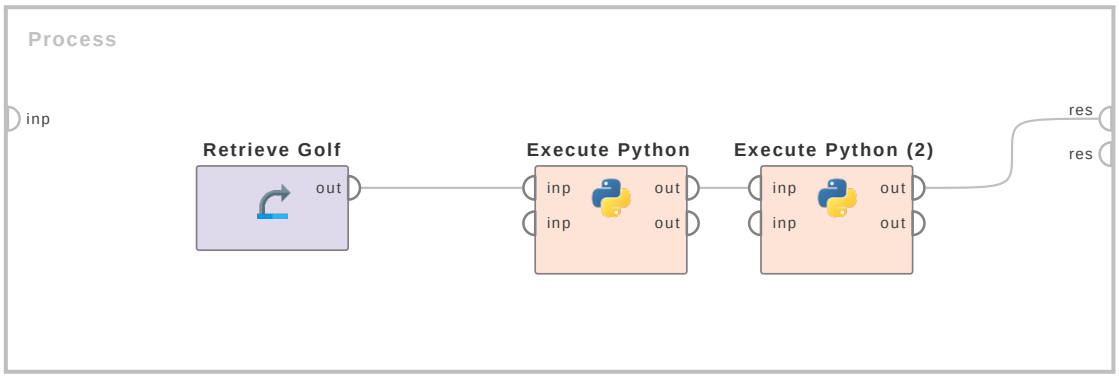



Figure 7.11: Tutorial process 'Reading an example set from a file using Python'.

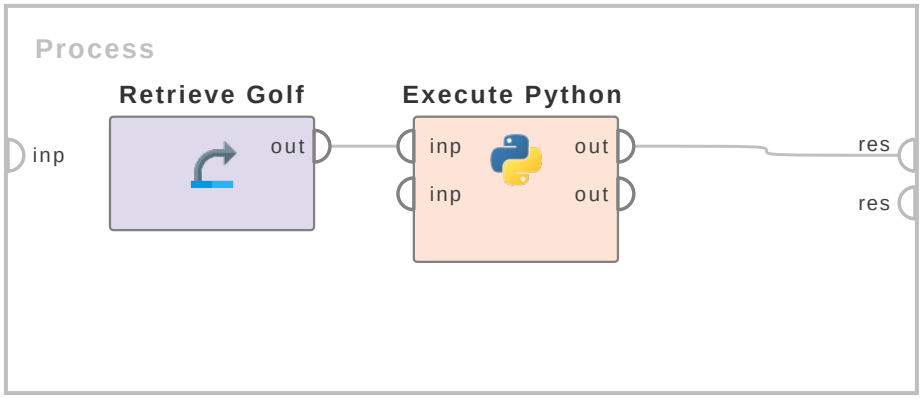



Figure 7.12: Tutorial process 'Meta data handling'.

### Training and applying a linear model in R

The polynomial data set is split in two parts. The first part is used by the 'Execute R' operator to train a linear model in R. The calculated model is passed on to the second 'Execute R' operator and applied there on the second part of the data set.

### Generating probability density functions for different probability functions in R

This script generates sample points for some statistical density functions and returns them as an example set.

### Reading an example set from a file using R

This tutorial process uses the 'Execute R' operator to save example data in a csv file. The second 'Execute R' operator receives this file, reads the data and returns a part of the data to the output port. The result is an example set.

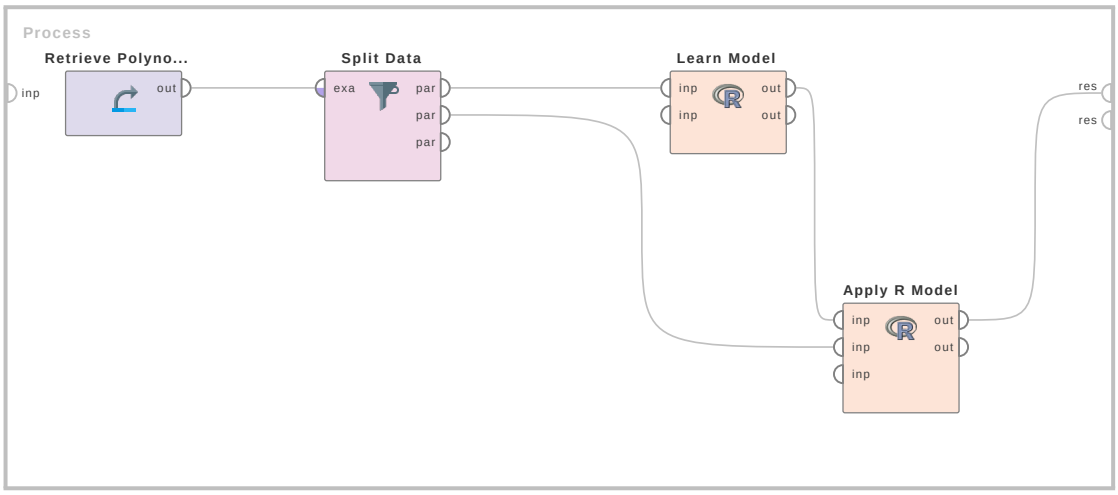



Figure 7.13: Tutorial process 'Training and applying a linear model in R'.

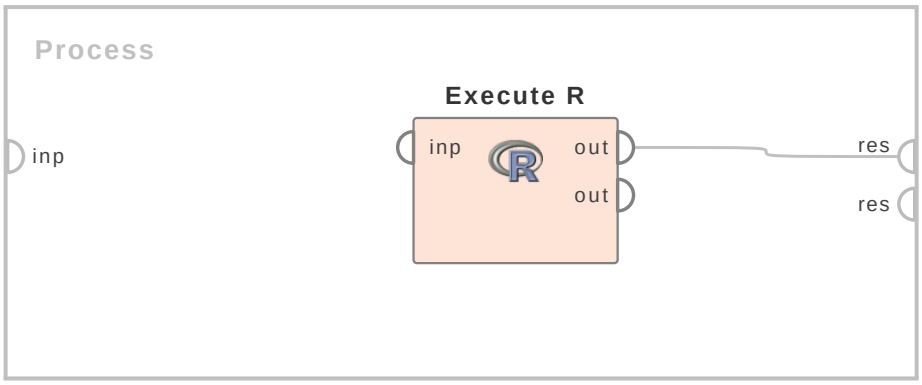



Figure 7.14: Tutorial process 'Generating probability density functions for different probability functions in R'.

### Meta data handling

This tutorial process shows how to access the meta data of incoming example sets inside a 'Execute R' operator. It also explains how to set the meta data for the outcoming example sets.

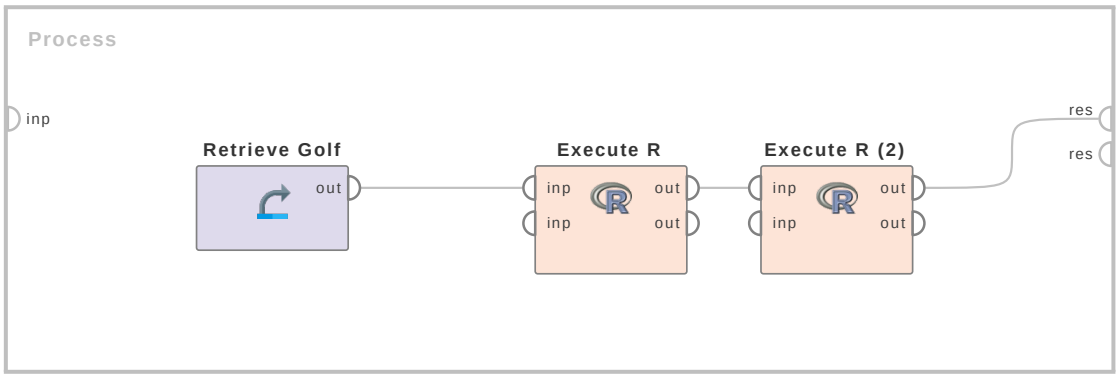



Figure 7.15: Tutorial process 'Reading an example set from a file using R'.

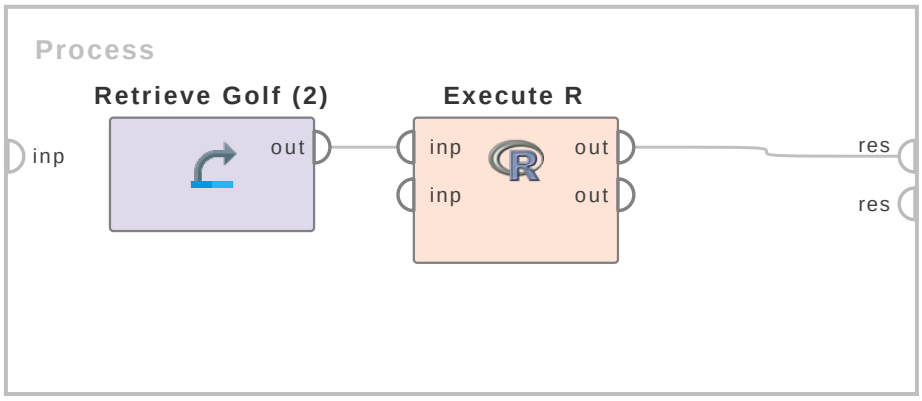
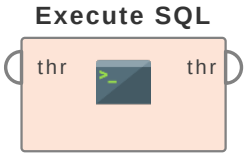



Figure 7.16: Tutorial process 'Meta data handling'.

# Execute SQL




This operator executes the specified SQL statement on the specified database.

## Description

The Execute SQL operator executes the specified SQL statement on the specified SQL database. The SQL query can be specified through the *query* parameter. If the SQL query is in a file then the path of that file can be specified through the *query file* parameter. Please note that this operator cannot be used for loading data from databases. It can be used for executing SQL statements like CREATE or ADD etc. In order to load data from an SQL database, please use the Read Database operator. You need to have at least a basic understanding of databases, database connections and queries in order to use this operator properly. Please go through the parameters and the attached Example Process to understand the working of this operator.

## Differentiation

- **Read Database** The Read Database operator is used for loading data from a database into RapidMiner. The Execute SQL operator cannot be used for loading data from databases. It can be used for executing SQL statements like CREATE or ADD etc on the database. See page 52 for details.

## Input Ports

**through *(thr)*** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the Execute SQL operator is available at the first *through* output port.

## Output Ports

**through *(thr)*** The objects that were given as input are passed without changing to the output through this port. It is not compulsory to connect this port to any other port; the SQL command is executed even if this port is left without connections. The Execute SQL operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Execute SQL operator is delivered at the first *through* output port

## Parameters

**define connection *(selection)*** This parameter indicates how the database connection should be specified. It gives you three options: predefined, url and jndi.

**connection *(selection)*** This parameter is only available when the *define connection* parameter is set to *predefined*. This parameter is used for connecting to a database using a predefined connection. You can have many predefined connections. You can choose one of them using the drop down list. You can add a new connections or modify previous connections using the button next to the drop down list. You may also accomplish this by clicking on *Manage Database Connections...* from the *Tools* menu in the main window. A new window appears. This window asks for several details e.g. *Host, Port, Database system, schema, username* and *password*. The *Test* button in this new window will allow you to check whether the connection can be made. Save the connection once the test is successful. After saving a new connection, it can be chosen from the drop down list of the *connection* parameter. You need to have a basic understanding of databases for configuring a connection.

**database system *(selection)*** This parameter is only available when the *define connection* parameter is set to *url*. This parameter is used for selecting the database system in use. It can have one of the following values: MySQL, PostgreSQL, Sybase, HSQLDB, ODBC Bridge (e.g. Access), Microsoft SQL Server (JTDS), Ingres, Oracle.

**database url *(string)*** This parameter is only available when the *define connection* parameter is set to *url*. This parameter is used for defining the URL connection string for the database, e.g. 'jdbc:mysql://foo.bar:portnr/database'.

**username** ***(string)*** This parameter is only available when the *define connection* parameter is set to *url*. This parameter is used for specifying the username of the database.

**password** ***(string)*** This parameter is only available when the *define connection* parameter is set to *url*. This parameter is used for specifying the password of the database.

**jndi name** ***(string)*** This parameter is only available when the *define connection* parameter is set to *jndi*. This parameter is used for specifying the JNDI name for a data source.

**query** ***(string)*** This parameter is used for specifying the SQL query which will be executed on the specified database.

**query file** ***(filename)*** This parameter is used for selecting the file that contains the SQL query which will be executed on the specified database. Long queries are usually stored in files. Storing queries in files can also enhance reusability.

**prepare statement** ***(boolean)*** If checked, the statement is prepared, and '?' can be filled in using the *parameters* parameter.

**parameters** ***(enumeration)*** This parameter specifies the Parameters to insert into '?' placeholders when the statement is prepared.

## Related Documents

- **Read Database** (page 52)

## Tutorial Processes

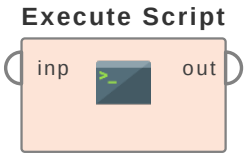
### Creating a new table in mySQL database


Figure 7.17: Tutorial process 'Creating a new table in mySQL database'.

The Execute SQL operator is used for creating a new table in an existing mySQL database. The define connection parameter is set to predefined. The define connection parameter was configured using the button next to the drop down list. The name of the connection was set to 'mySQL-conn'. The following values were set in the connection parameter's wizard. The Database system was set to 'mySQL'. The Host was set to 'localhost'. The Port was set to '3306'. The Database scheme was set to 'golf'; this is the name of the database. The User was set to 'root'. No password was provided. You will need a password if your database is password protected. Set all the values and test the connection. Make sure that the connection works.

The query parameter is set to the following query: 'CREATE TABLE Weather(Temperature INTEGER)'. This query creates a new table named Weather in the 'golf' database. This table has one integer attribute named Temperature. Run the process, you will not see any results in RapidMiner because this operator did not return anything. It simply executed the query on the specified database. So, in order to see the changes you can open the database and verify that a new table has been created.

# Execute Script




This operator executes Java code and/or Groovy scripts. This basically means that users can write their own operators directly within the process by specifying Java code and/or a Groovy script which will be interpreted and executed during the process runtime.

## Description

This is a very powerful operator because it allows you to write your own script. This operator should be used if the task you want to perform through your script cannot be performed by existing RapidMiner operators because writing scripts can be time-consuming and error-prone.

Groovy is an agile and dynamic language for the Java Virtual Machine. It builds upon the strengths of Java but has additional power features inspired by languages like Python, Ruby and Smalltalk. Groovy integrates well with all existing Java classes and libraries because it compiles straight to Java bytecode so you can use it anywhere you can use Java. For a complete reference of Groovy scripts please refer to http://groovy.codehaus.org/.

In addition to the usual scripting code elements from Groovy, the RapidMiner scripting operator defines some special scripting elements:

- If the *standard imports* parameter is set to true, all important types like Example, ExampleSet, Attribute, Operator etc as well as the most important Java types like collections etc are automatically imported and can directly be used within the script. Hence, there is no need for importing them in your script. However, you can import any other class you want and use it in your script.
- The current operator (the scripting operator for which you define the script) is referenced by *operator*.
  - Example: *operator.log(”text”)*
- All operator methods like *log* (see above) that access the input or the complete process can directly be used by writing a preceding *operator*.
  - Example: *operator.getProcess()*
- Input of the operator can be retrieved via the input method *getInput(Class)*of the surrounding *operator*.
  - Example: *ExampleSet exampleSet = operator.getInput(ExampleSet.class)*
- You can iterate over examples with the following construct:
  - *for (Example example : exampleSet) { ... }*
- You can retrieve example values with the shortcut:
  - In case of non-numeric values: *String value = example[”attribute_name”];*
  - In case of numeric values: *double value = example[”attribute_name”];*
- You can set example values with the shortcut:
  - In case of non-numeric values: *example[”attribute_name”] = “value”;*
  - In case of numeric values: *example[”attribute_name”] = 5.7;*

Please study the attached Example Processes for better understanding. Please note that Scripts written for this operator may access Java code. Scripts may hence become incompatible in future releases of RapidMiner.

## Input Ports

**input *(inp)*** The Script operator can have multiple inputs. When one input is connected, another *input* port becomes available which is ready to accept another input (if any).

## Output Ports

**output *(out)*** The Script operator can have multiple outputs. When one output is connected, another *output* port becomes available which is ready to deliver another output (if any).

## Parameters

**script** The script to be executed is specified through this parameter.

**standard imports *(boolean)*** If the *standard imports* parameter is set to true, all important types like Example, ExampleSet, Attribute, Operator etc as well as the most important Java types like collections etc are automatically imported and can directly be used within the script. Hence, there is no need for importing them in your script. However, you can import any other class you want and use it in your script.

## Tutorial Processes

### Iterating over attributes for changing the attribute names to lower case

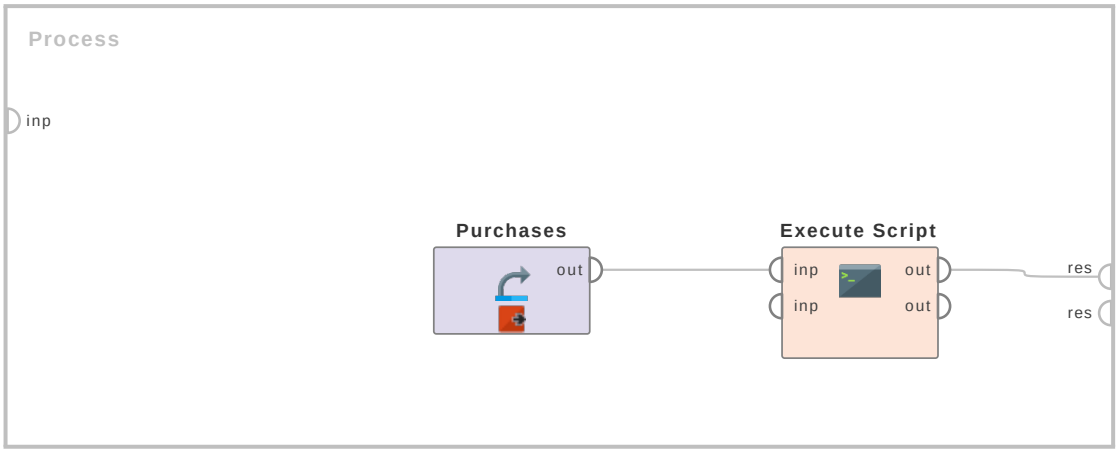



Figure 7.18: Tutorial process 'Iterating over attributes for changing the attribute names to lower case'.

The 'Purchases' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the ExampleSet. Note that the names of all attributes of the ExampleSet are in upper case letters. The Script operator is applied on the ExampleSet. The script changes the attribute names to lower case letters. This can be verified by viewing the results in the Results Workspace.

Here is a brief description of what happens in the script. First the input of the operator is retrieved via the input method getInput(Class). Then the for loop iterates for all attributes and

uses the toLowerCase() method to change the names of the attributes to lower case letters. At the end, the modified ExampleSet is returned.

Please note that this is a very simple script, it was included here just to introduce you with working of this operator. This operator can be used to perform very complex tasks.

### Iterating over all examples for changing the attribute values to upper case

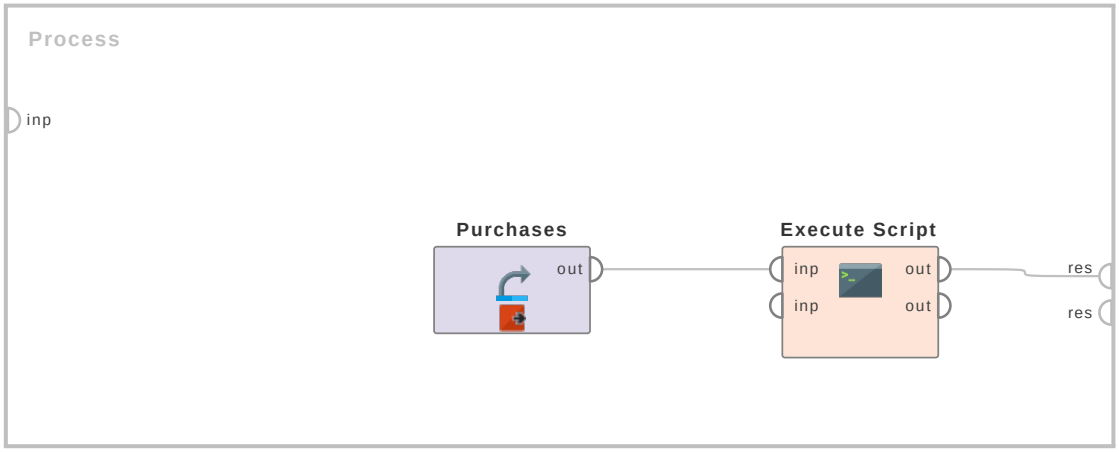



Figure 7.19: Tutorial process 'Iterating over all examples for changing the attribute values to upper case'.

The 'Purchases' data set is loaded using the Retrieve operator. A Breakpoint is inserted here so that you can view the ExampleSet. Note that the values of all attributes of the ExampleSet are in lower case letters. The Script operator is applied on the ExampleSet. The script changes the attribute values to upper case letters. This can be verified by viewing the results in the Results Workspace.

Here is a brief description of what happens in the script. First the input of the operator is retrieved via the input method getInput(Class). Then the outer for loop iterates for all attributes and stores the name of the current attribute in a string variable. Then the inner for loop iterates over all the examples of the current attribute and changes the values from lower to upper case using the toUpperCase() method. At the end, the modified ExampleSet is returned.

Please note that this is a very simple script, it was included here just to introduce you with working of this operator. This operator can be used to perform very complex tasks.

### Subtracting mean of numerical attributes from attribute values

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the ExampleSet. Note the values of the 'Temperature' and 'Humidity' attributes. The Script operator is applied on the ExampleSet. The script subtracts the mean of each numerical attribute from all values of that attribute. This can be verified by viewing the results in the Results Workspace.

Here is a brief description of what happens in the script. First the input of the operator is retrieved via the input method getInput(Class). Then the outer for loop iterates for all attributes and stores the name of the current attribute in a string variable and the mean of this attribute in a double type variable. Then the inner for loop iterates over all the examples of the current

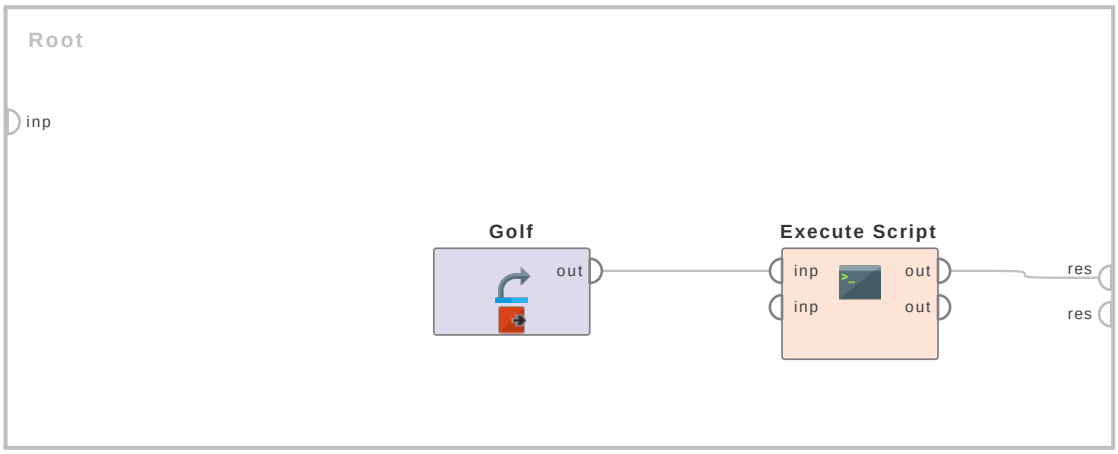



Figure 7.20: Tutorial process ‘Subtracting mean of numerical attributes from attribute values’.

attribute and subtracts the mean from the current value of the example. At the end, the modified ExampleSet is returned.

Please note that this is a very simple script, it was included here just to introduce you with working of this operator. This operator can be used to perform very complex tasks.

# 7.2 Process Control

## Publish to App


This operator stores the given object in the RapidMiner Server App or the current RapidMiner Studio session. The stored object can be retrieved by using the Recall from App operator.

### Description

The Publish to App operator can be used to store the input object in the RapidMiner Server App or the current RapidMiner Studio session. The name of the object is specified through the *name* parameter. The stored object can later be retrieved via the Recall from App operator by using the same name (i.e. the name that was used to store it with the Publish to App operator). When an object was stored by the Publish to App operator once, it can be recalled at any point in the RapidMiner Server App. But care should be taken that the execution order of operators is such that the Publish to App operator for an object is executed before the Recall from App operator for that object. The combination of these two operators can be used to build complex RapidMiner Server Apps, where an input object is stored once and used in completely different parts of the App later on.

### Differentiation

- **Recall from App** The Publish to App operator is always used in combination with the Recall from App operator. The Publish to App operator stores the required object in the RapidMiner Server App and the Recall from App operator retrieves the stored object when required. See page 846 for details.

### Input Ports

**store *(sto)*** Any object can be provided here. This object will be stored in the RapidMiner Server App or the current RapidMiner Studio session.

### Output Ports

**stored *(sto)*** The object that was given as input is passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the object will be cached even if this port is left without connections.

### Parameters

**name *(string)*** The name under which the input object is stored is specified through this parameter. The same name will be used for retrieving this object through the Recall from App operator.

### Related Documents

- **Recall from App** (page 846)

## Tutorial Processes

### Introduction to Publish to App and Recall from App operators

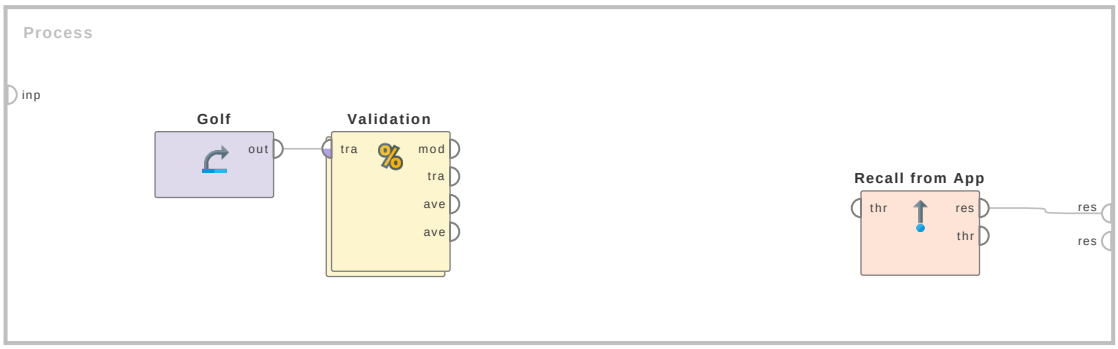



Figure 7.21: Tutorial process 'Introduction to Publish to App and Recall from App operators'.

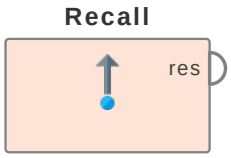
This process uses the combination of the Publish to App and Recall from App operators to display the testing data set of the Split Validation operator. The testing data set is present in the testing subprocess of the Split Validation operator but it is not available outside the Split Validation operator.

The 'Golf' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it. The test set size parameter is set to 5 and the training set size parameter is set to -1. Thus the test set in the testing subprocess will be composed of 5 examples. The Default Model operator is used in the training subprocess to train a model. The testing data set is available at the tes port of the testing subprocess. The Publish to App operator is used to store the testing data set in the RapidMiner Server App. The name and io object parameters are set to 'Testset' and 'ExampleSet' respectively. The Apply Model and Performance operator are applied in the testing subprocess later. In the main process, the Recall from App operator is used to retrieve the testing data set. The name and io object parameters of the Recall from App operator are set to 'Testset' and 'ExampleSet' respectively to retrieve the object that was cached by the Publish to App operator. The output of the Recall from App operator is connected to the result port of the process. Therefore the testing data set can be seen in the Results view.

The operator Recall from App cannot be used to retrieve objects stored with the Remember operator, and the operator Recall cannot be used to retrieve objects stored with the Publish to App operator. They do not use the same storage.

The difference to the operators Remember and Recall is that Publish to App does not only store an object for the process execution (as the Remember operator does), but for the lifetime of a RapidMiner Server App. This means that the Recall from App operator could also be part of another process in the same RapidMiner Server App and would retrieve the same object as long as the process, which remembers the object for the App, was executed beforehand.

# Recall




This operator retrieves the specified object from the object store of the process. The objects can be stored in the object store by using the Remember operator.

## Description

The Recall operator can be used for retrieving the specified object from the object store of the process. The name of the object is specified through the *name* parameter. The *io object* parameter specifies the class of the required object. The Recall operator is always used in combination with the operators like the Remember operator. For Recall operator to retrieve an object, first it is necessary that the object should be stored in the object store by using operators like the Remember operator. The name and class of the object are specified when the object is stored using the Remember operator. The same name (in *name* parameter) and class (in *io object* parameter) should be specified in the Recall operator to retrieve that object. The same stored object can be retrieved multiple number of times if the *remove from store* parameter of the Recall operator is not set to true. There is no scoping mechanism in RapidMiner processes therefore objects can be stored (using Remember operator) and retrieved (using Recall operator) at any nesting level. But care should be taken that the execution order of operators is such that the Remember operator for an object always executes before the Recall operator for that object. The combination of these two operators can be used to build complex processes where an input object is used in completely different parts or loops of the processes.

## Differentiation

- **Remember** The Recall operator is always used in combination with the Remember operator. The Remember operators stores the required object into the object store and the Recall operator retrieves the stored object when required.

  See page 849 for details.

## Output Ports

**result *(res)*** The specified object is retrieved from the object store of the process and is delivered through this output port.

## Parameters

**name *(string)*** The name of the required object is specified through this parameter. This name should be the same name that was used while storing the object in an earlier part of the process.

**io object *(selection)*** The class of the required object is selected through this parameter. This class should be the same class that was used while storing the object in an earlier part of the process.

**remove from store *(boolean)*** If this parameter is set to true, the specified object is removed from the object store after it has been retrieved. In such a case the object can be retrieved just once. If this parameter is set to false, the object remains in the object store even after

retrieval. Thus the object can be retrieved multiple number of times (by using the Recall operator multiple number of times).

## Related Documents

- **Remember** (page 849)

## Tutorial Processes

### Introduction to Remember and Recall operators

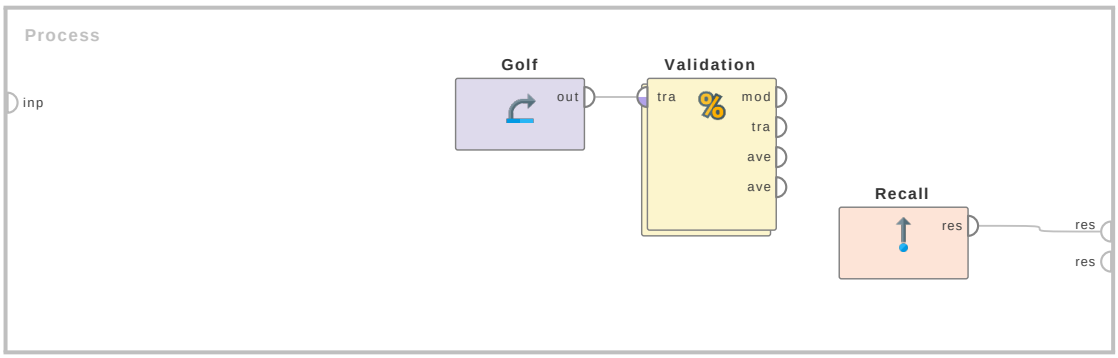
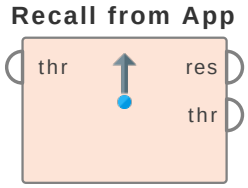



Figure 7.22: Tutorial process 'Introduction to Remember and Recall operators'.

This process uses the combination of the Remember and Recall operators to display the testing data set of the Split Validation operator. The testing data set is present in the testing subprocess of the Split Validation operator but it is not available outside the Split Validation operator.

The 'Golf'' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it. The test set size parameter is set to 5 and the training set size parameter is set to -1. Thus the test set in the testing subprocess will be composed of 5 examples. The Default Model operator is used in the training subprocess to train a model. The testing data set is available at the tes port of the testing subprocess. The Remember operator is used to store the testing data set into the object store of the process. The Apply Model and Performance operator are applied in the testing subprocess later. In the main process, the Recall operator is used to retrieve the testing data set. The name and io object parameters of the Recall operator are set to 'Testset' and 'ExampleSet' respectively to retrieve the object that was stored by the Remember operator. The output of the Recall operator is connect to the result port of the process. Therefore the testing data set can be seen in the Results Workspace.

# Recall from App




This operator retrieves the specified object from a RapidMiner Server App or the current RapidMiner Studio session. Ojects can be stored by using the Publish to App operator.

## Description

The Recall from App operator can be used for retrieving the specified object from a RapidMiner Server App or the current RapidMiner Studio session. The name of the object is specified through the *name* parameter. The Recall from App operator is always used in combination with the Publish to App operator. For the Recall from App operator to retrieve an object, first it is necessary that the object has to be stored by using the Publish to App operator. The name of the object is specified when the object is stored. The same name (in *name* parameter) should be specified in the Recall from App operator to retrieve that object. The same object can be retrieved multiple times, even from within another process, if the *remove from app* parameter of the Recall from App operator is not set to true. When an object was stored by the Publish to App operator once, it can be recalled at any point in the RapidMiner Server App or the current RapidMiner Studio session. But care should be taken that the execution order of operators is such that the Publish to App operator for an object always executes before the Recall from App operator for that object. The combination of these two operators can be used to build complex RapidMiner Server Apps where an input object is stored once and used in completely different parts of the App later on.

## Differentiation

- **Publish to App** The Recall from App operator is always used in combination with the Publish to App operator. The Publish to App operator stores the input object in the RapidMiner Server App and the Recall from App operator retrieves the stored object when required. See page **??** for details.

## Input Ports

**through *(thr)*** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the Recall from App operator is available at the first *through* output port.

## Output Ports

**result *(res)*** The specified object is retrieved from the RapidMiner Server App or the current RapidMiner Studio session and is delivered through this output port.

**through *(thr)*** The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the specified

object is retrieved from the RapidMiner Server App even if this port is left without connections. The Recall from App operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Recall from App operator is delivered at the first *through* output port

## Parameters

**name *(string)*** The name of the required object is specified through this parameter. This name should be the same name that was used while storing the object in an earlier call of the operator Publish to App.

**remove from dashboard *(boolean)*** If this parameter is set to true, the specified object is removed from the RapidMiner Server App or the current RapidMiner Studio session after it has been retrieved. In such a case the object can be retrieved just once. If this parameter is set to false, the object remains in the RapidMiner Server App even after retrieval. Thus the object can be retrieved by multiple Recall from App operators.

## Related Documents

- **Publish to App** (page **??**)

## Tutorial Processes

### Introduction to Publish to App and Recall from App operators

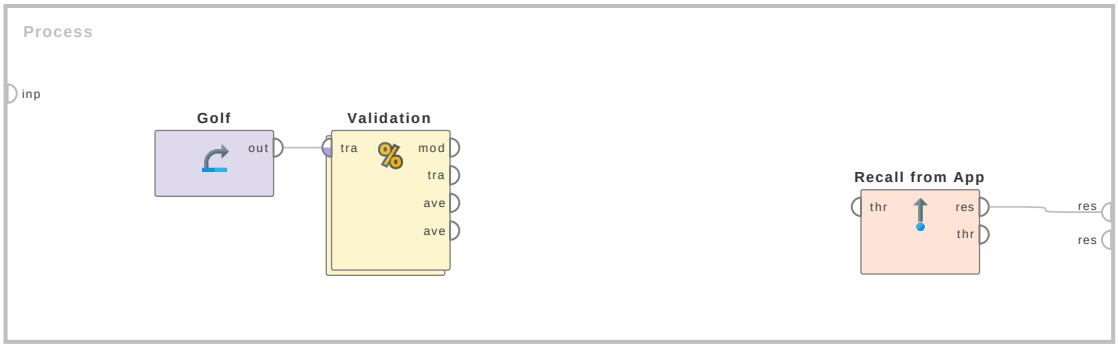



Figure 7.23: Tutorial process 'Introduction to Publish to App and Recall from App operators'.

This process uses the combination of the Publish to App and Recall from App operators to display the testing data set of the Split Validation operator. The testing data set is present in the testing subprocess of the Split Validation operator but it is not available outside the Split Validation operator.

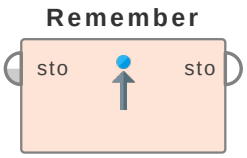
The 'Golf' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it. The test set size parameter is set to 5 and the training set size parameter is set to -1. Thus the test set in the testing subprocess will be composed of 5 examples. The Default Model operator is used in the training subprocess to train a model. The testing data set is available at

the tes port of the testing subprocess. The Publish to App operator is used to store the testing data set in the RapidMiner Server App. The name and io object parameters are set to 'Testset' and 'ExampleSet' respectively. The Apply Model and Performance operator are applied in the testing subprocess later. In the main process, the Recall from App operator is used to retrieve the testing data set. The name and io object parameters of the Recall from App operator are set to 'Testset' and 'ExampleSet' respectively to retrieve the object that was cached by the Publish to App operator. The output of the Recall from App operator is connected to the result port of the process. Therefore the testing data set can be seen in the Results view.

The operator Recall from App cannot be used to retrieve objects stored with the Remember operator, and the operator Recall cannot be used to retrieve objects stored with the Publish to App operator. They do not use the same storage.

The difference to the operators Remember and Recall is that Publish to App does not only store an object for the process execution (as the Remember operator does), but for the lifetime of a RapidMiner Server App. This means that the Recall from App operator could also be part of another process in the same RapidMiner Server App and would retrieve the same object as long as the process, which remembers the object for the App, was executed beforehand.

# Remember




This operator stores the given object in the object store of the process. The stored object can be retrieved from the store by using the Recall operator.

## Description

The Remember operator can be used to store the input object into the object store of the process under the specified name. The name of the object is specified through the *name* parameter. The *io object* parameter specifies the class of the object. The stored object can later be restored by the Recall operator by using the same name and class (i.e. the name and class that was used to store it using the Remember operator). There is no scoping mechanism in RapidMiner processes therefore objects can be stored (using Remember operator) and retrieved (using Recall operator) at any nesting level. But care should be taken that the execution order of operators is such that the Remember operator for an object always executes before the Recall operator for that object. The combination of these two operators can be used to build complex processes where an input object is used in completely different parts or loops of the processes.

## Differentiation

- **Recall** The Remember operator is always used in combination with the Recall operator. The Remember operators stores the required object into the object store and the Recall operator retrieves the stored object when required.

  See page 844 for details.

## Input Ports

**store *(sto)*** Any object can be provided here. This object will be stored in the object store of the process. It should be made sure that the class of this object is selected in the *io object* parameter.

## Output Ports

**stored *(sto)*** The object that was given as input is passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the object will be stored even if this port is left without connections.

## Parameters

**name *(string)*** The name under which the input object is stored is specified through this parameter. The same name will be used for retrieving this object through the Recall operator.

**io object *(selection)*** The class of the input object is selected through this parameter.

## Related Documents

- **Recall** (page 844)

## Tutorial Processes

### Introduction to Remember and Recall operators

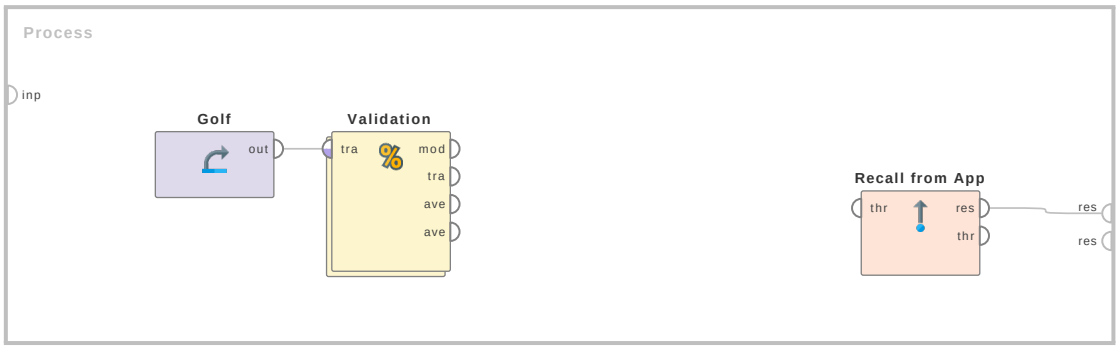


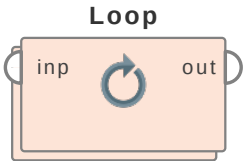

Figure 7.24: Tutorial process ‘Introduction to Remember and Recall operators’.

This process uses the combination of the Remember and Recall operators to display the testing data set of the Split Validation operator. The testing data set is present in the testing subprocess of the Split Validation operator but it is not available outside the Split Validation operator.

The ‘Golf’ data set is loaded using the Retrieve operator. The Split Validation operator is applied on it. The test set size parameter is set to 5 and the training set size parameter is set to -1. Thus the test set in the testing subprocess will be composed of 5 examples. The Default Model operator is used in the training subprocess to train a model. The testing data set is available at the tes port of the testing subprocess. The Remember operator is used to store the testing data set into the object store of the process. The name and io object parameters are set to ‘Testset’ and ‘ExampleSet’ respectively. The Apply Model and Performance operator are applied in the testing subprocess later. In the main process, the Recall operator is used to retrieve the testing data set. The name and io object parameters of the Recall operator are set to ‘Testset’ and ‘ExampleSet’ respectively to retrieve the object that was stored by the Remember operator. The output of the Recall operator is connected to the result port of the process. Therefore the testing data set can be seen in the Results Workspace.

## 7.2.1 Loops

### Loop




This operator loops over the subprocess as often as it is specified in the parameter *number of iterations*. The *iteration macro* returns the value of the current number of iteration.

#### Description

The operators in the subprocess are executed as many times as defined in the parameter *number of iterations*. By default, the input of each iteration will always be the original input data. This can be changed with the *reuse results* parameter, which changes the output of each iteration to be the input of the next iteration. For obvious reasons, this will limit the loop to run in a single thread and not make use of more CPU cores.

#### Input Ports

**input *(inp)*** This port receives an *IOObject* which is passed on to the inner process for each iteration. If the *reuse results* parameter is selected, all subsequent iterations after the first one will use the results of the *output* ports of the previous iteration.

#### Output Ports

**output *(out)*** This port collects every result that is provided by the inner process. If *reuse results* is selected, only the result of the last iteration will be returned. Otherwise, a collection of all results of each iteration will be returned. The order of the ports is the same inside and outside the operator.

#### Parameters

**number of iterations *(integer)*** The *number of iterations* specifies how often the subprocess will be executed.

**iteration macro *(string)*** The name of the iteration macro which can be accessed in the subprocess.

**reuse results *(boolean)*** Set whether to reuse the results of each iteration as the input of the next iteration. If set to true, the output of each iteration is used as input for the next iteration. For obvious reasons, this will limit the loop to run in a single thread and not make use of more CPU cores. If set to false, the input of each iteration will be the original input of the loop.

**enable parallel execution *(boolean)*** This parameter enables the parallel execution of the subprocess. Please disable the parallel execution if you run into memory problems.


Figure 7.25: Tutorial process ‘Using a basic loop’.

## Tutorial Processes

### Using a basic loop

The ‘Deals’ data set is loaded using the Retrieve operator and supplied to the Loop operator.

Inside the Loop operator, nothing else is done than returning the input as result after waiting for 1 second and adding an attribute based on the current iteration of the loop.

The result of the process is a collection of the results of each iteration.

### Reusing results in a loop

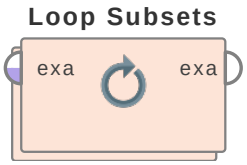

Figure 7.26: Tutorial process ‘Reusing results in a loop’.

The ‘Deals’ data set is loaded using the Retrieve operator and supplied to the Loop operator.

Inside the Loop operator, nothing else is done than returning the input as result after waiting for 1 second and adding an attribute based on the current iteration of the loop.

The results of each iteration are reused as the input of the next iteration, so the final result of the Loop operator is not a collection of each iteration result, but rather the result of the final iteration.

# Loop Attribute Subsets




This operator iterates over its subprocess for all possible combinations of regular attributes in the input ExampleSet. Optionally, the minimum and maximum number of attributes in a combination can be specified by the user.

## Description

The Loop Attribute Subsets operator is a nested operator i.e. it has a subprocess. The subprocess of the Loop Attribute Subsets operator executes *n* number of times, where *n* is the number of possible combinations of the regular attributes in the given ExampleSet. The user can specify the minimum and maximum number of attributes in a combination through the respective parameters; in this case the value of *n* will change accordingly. So, if an ExampleSet has three regular attributes say a, b and c. Then this operator will execute 7 times; once for each attribute combination. The combinations will be {a},{b},{c},{a,b},{a,c},{b,c} and {a,b,c}. Please study the attached Example Process for more information.

This operator can be useful in combination with the Log operator and, for example, a performance evaluation operator. In contrast to the brute force feature selection, which performs a similar task, this iterative approach needs much less memory and can be performed on larger data sets.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set *(exa)*** The ExampleSet that was given as input is delivered through this port without any modifications.

## Parameters

**use exact number *(boolean)*** If this parameter is set to true, then the subprocess will be executed only for combinations of a specified length i.e. specified number of attributes. The length of combinations is specified by the *exact number of attributes* parameter.

**exact number of attributes *(integer)*** This parameter determines the exact number of attributes to be used for the combinations.

**min number of attributes *(integer)*** This parameter determines the minimum number of attributes to be used for the combinations.

**limit max number *(boolean)*** If this parameter is set to true, then the subprocess will be executed only for combinations that have less than or equal to *m* number of attributes; where *m* is specified by the *max number of attributes* parameter.

**max number of attributes *(integer)*** This parameter determines the maximum number of attributes to be used for the combinations.

## Tutorial Processes

### Introduction to the Loop Attribute Subsets operator

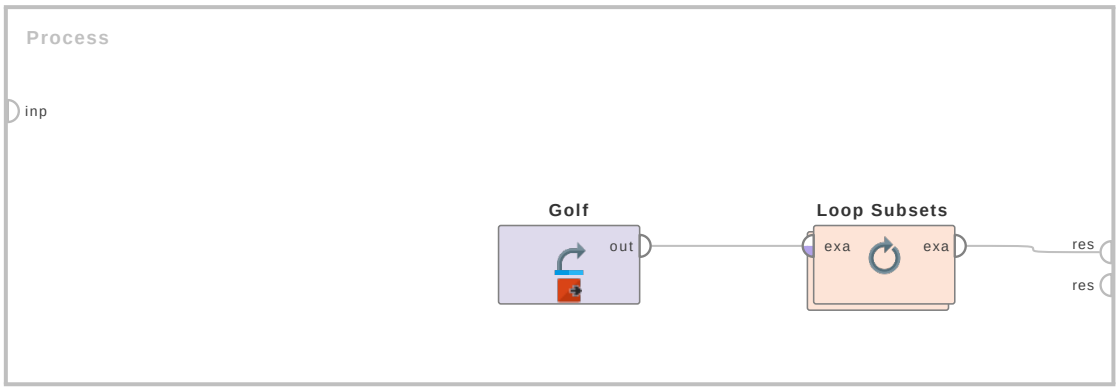
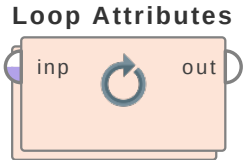



Figure 7.27: Tutorial process 'Introduction to the Loop Attribute Subsets operator'.

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet before the application of the Loop Attribute Subsets operator. You can see that the ExampleSet has four regular attributes. The Loop Attribute Subsets operator is applied on this ExampleSet with default values of all parameters. As no limit is applied on the minimum and maximum number of attributes in a combination, the subprocess of this operator will execute for all possible combinations of the four regular attributes. Have a look at the subprocess of the Loop Attribute Subsets operator. The Log operator is applied there to store the names of attributes of each iteration in the Log table. Execute the process and shift to the Results Workspace. Check the Table View of the Log results. You will see the names of attributes of each iteration. As there were 4 attributes there are 15 possible non-null combinations.

# Loop Attributes




This operator selects a subset (one or more attributes) of the input ExampleSet and iterates over its subprocess for all the selected attributes. The subprocess can access the attribute of current iteration by a macro.

## Description

The Loop Attributes operator has a number of parameters that allow you to select the required attributes of the input ExampleSet. Once the attributes are selected, the Loop Attributes operator applies its subprocess for each attribute i.e. the subprocess executes *n* number of times where *n* is the number of selected attributes. In all iterations the attribute of the current iteration can be accessed using the macro specified in the *iteration macro* parameter. You need to have basic understanding of macros in order to apply this operator. Please study the documentation of the Extract Macro operator for basic understanding of macros. The Extract Macro operator is also used in the attached Example Process. For more information regarding subprocesses please study the Subprocess operator.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set *(exa)*** The resultant ExampleSet, or Collection of ExampleSets is delivered through this port.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet, no attributes are removed. This is the default option.
- **single** This option allows the selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in the Parameters panel.
- **subset** This option allows the selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for the attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both

belong to the *numeric* type. The user should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameters panel.

- **block_type** This option is similar in working to the *value_type* option. This option allows the selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric_value_filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** ***(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *attribute* parameter if the meta data is known.

**attributes** ***(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list, which is the list of selected attributes that will make it to the output port; all other attributes will be removed.

**regular expression** ***(string)*** The attributes whose name match this expression will be selected. Regular expression is very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (except regular expression) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date, time.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. the *value type* parameter's value. One of the following types can be selected here: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date, time.

**block type** ***(selection)*** The Block type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: single_value, value_series, value_series_start, value_series_end, value_matrix, value_matrix_start, value_matrix_end, value_matrix_row_start.

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type. One of the following block types can be selected here: single_value, value_series, value_series_start, value_series_end, value_matrix, value_matrix_start, value_matrix_end, value_matrix_row_start.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is mention here. For example the numeric condition '> 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '> 6 && < 11' or '<= 5 || < 0'. But && and || cannot be used together in one numeric condition. Conditions like '(> 0 && < 2) || (>10 && < 12)' are not allowed because they use both && and ||. Use a blank space after '>', '=' and '<' e.g. '<5' will not work, so use '< 5' instead.

**invert selection** ***(boolean)*** If this parameter set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are removed and previously removed attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is removed prior to selection of this parameter. After selection of this parameter 'att1' will be removed and 'att2' will be selected.

**include special attributes** ***(boolean)*** Special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are delivered to the output port irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**attribute name macro** ***(string)*** This parameter specifies the name of the macro which holds the name of the current attribute in each iteration.

**reuse results** ***(boolean)*** Set whether to reuse the results of each iteration as the input of the next iteration. If set to true, the output of each iteration is used as input for the next iteration. For obvious reasons, this will limit the loop to run in a single thread and not make use of more CPU cores. If set to false, the input of each iteration will be the original input of the loop.

**enable parallel execution** ***(boolean)*** This parameter enables the parallel execution of the subprocess. Please disable the parallel execution if you run into memory problems.

## Tutorial Processes

### Generating new attributes in the Loop Attributes operator

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet before application of the Loop Attributes operator. Have

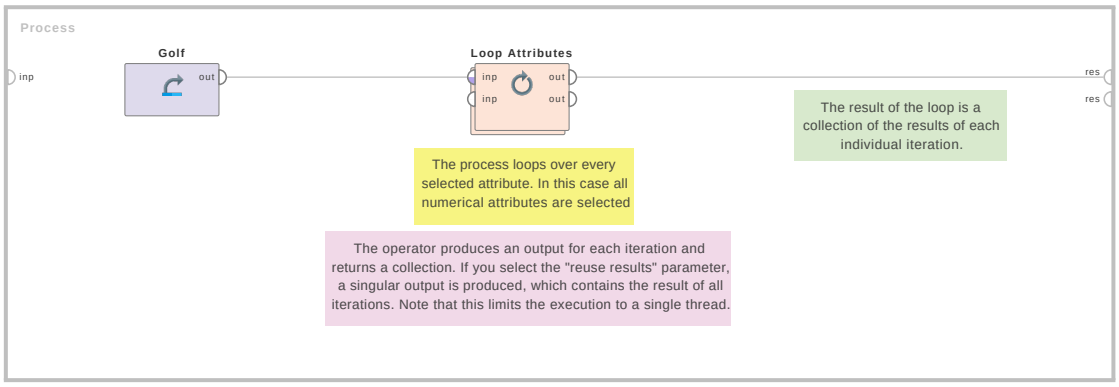
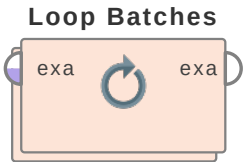



Figure 7.28: Tutorial process ‘Generating new attributes in the Loop Attributes operator’.

a look at the parameters of the Loop Attributes operator. The attribute filter type parameter is set to ‘value type’ and the value type parameter is set to ‘numeric’ and the include special attributes parameter is set to true. Thus all numeric attributes are selected from the ‘Golf’ data set i.e. the Temperature and Humidity attributes are selected. Therefore the subprocess of the Loop Attributes operator will iterate twice. In each iteration the current attribute can be accessed by the ‘loop_attribute’ macro defined by the iteration macro parameter. Now have a look at the subprocess of the Loop Attributes operator. The Extract Macro operator is applied first. The parameters of the Extract Macro operator are adjusted such that the ‘avg’ macro holds the average or mean of the attribute of the current iteration. Please note how the ‘loop_attribute’ macro is used in parameters of the Extract Macro operator. Next, the Generate Attributes operator is applied. It generates a new attribute from the attribute of the current iteration. The new attribute holds the deviation of examples from the mean of that attribute. The mean was stored in the ‘avg’ macro. Please note carefully the use of macros in the function descriptions parameter of the Generate Attributes operator.

Thus the subprocess of the Loop Attributes operator executes twice, once for each value of selected attributes. In the first iteration a new attribute is created with the name ‘Deviation(Temperature)’ which holds the deviations of the Temperature values from the mean of the Temperature attribute. In the second iteration a new attribute is created with the name ‘Deviation(Humidity)’ which holds the deviations of the Humidity values from the mean of the Humidity attribute.

# Loop Batches




This operator creates batches from the input ExampleSet and executes its subprocess on each of these batches. This can be useful for applying operators on very large data sets that are in a database.

## Description

This operator groups the examples of the input ExampleSet into batches of the specified size and executes the inner operators on all batches subsequently. This can be useful for very large data sets which cannot be loaded into memory and must be handled in the database. In such cases, preprocessing methods or model applications and other tasks can be performed on each batch and the results can be written into the database table (by using the Write Database or Update Database operators). Note that the output of this operator is not composed of the results of the subprocess. In fact the subprocess does not need to deliver any output since it operates on a subset view of the input ExampleSet. Thus this operator returns the input ExampleSet without any modifications. The results of the subprocess are not directly accessible, they can be written into a database or a file during the execution of this process. The results of the last batch can be accessed using Remember/Recall operators.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set *(exa)*** The ExampleSet that was given as input is delivered through this port without any modifications.

## Parameters

**batch size *(integer)*** This parameter specifies the number of examples in a batch.

## Tutorial Processes

### Introduction to the Loop Batches operator

The 'Iris' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet before the application of the Loop Batches operator. You can see that the ExampleSet has 150 examples. The Loop Batches operator is applied on this ExampleSet. The batch size parameter is set to 50. Given that there are 150 examples and the batch size is 50, there will be 3 (i.e. 150/50) iterations of this operator. Have a look at the subprocess of the Loop Batches operator. The Remember operator is applied there to store the examples of each iteration into the object table as an ExampleSet. A breakpoint is inserted before the Remember operator so that you can have a look at the examples of each iteration. On execution of process, you will see three iterations of the Loop Batches operator. You can see that the first

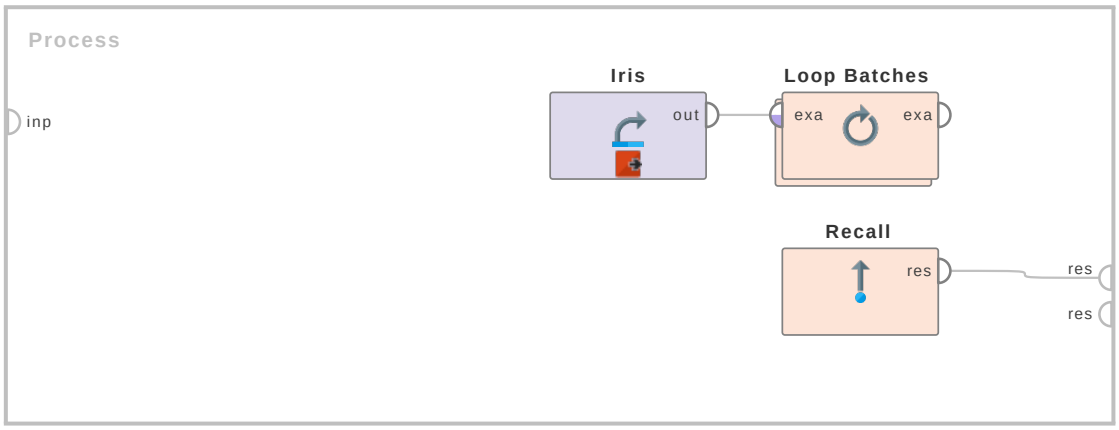
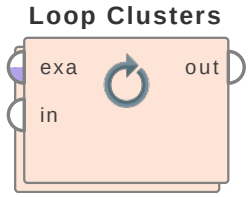



Figure 7.29: Tutorial process ‘Introduction to the Loop Batches operator’.

iteration has examples from id_1 to id_50. Similarly the consequent iterations have examples id_51 to id_100 and id_101 to id_150. At the end, the Recall operator is used for fetching the objects stored by the Remember operator. The Recall operator can only fetch the examples of the last batch because the previous batches were overridden by the consequent batches.

# Loop Clusters




This operator iterates over its subprocess for each cluster in the input ExampleSet. In each iteration the subprocess receives examples belonging to the cluster of that iteration.

## Description

The Loop Clusters operator is a nested operator i.e. it has a subprocess. The subprocess of the Loop Clusters operator executes *n* number of times, where *n* is the number of clusters in the given ExampleSet. It is compulsory that the given ExampleSet should have a cluster attribute. Numerous clustering operators are available in RapidMiner that generate a cluster attribute e.g. the K-Means operator. The subprocess executes on examples of one cluster in an iteration, on examples of the next cluster in next iteration and so on. Please study the attached Example Process for better understanding.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is compulsory that the ExampleSet should have an attribute with cluster role. It is output of the K-Means operator in the attached Example Process.

**in *(in )*** This operator can have multiple *in* input ports. When one input is connected, another *in* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object delivered at first *in* port of the operator is available at first *in* port of the subprocess. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

## Output Ports

**out *(out)*** This operator can have multiple *out* output ports. When one output is connected, another *out* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at first *out* port of subprocess is delivered at first *out* output port of the outer process. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

## Tutorial Processes

### Introduction to the Loop Clusters operator

The 'Ripley-Set' data set is loaded using the Retrieve operator. The K-Means operator is applied on it for generating a cluster attribute. A breakpoint is inserted here so that you can have a look at the clustered ExampleSet. You can see that there is an attribute with cluster role. It has two possible values i.e. cluster_0 and cluster_1. This means that there are two clusters in the ExampleSet. The Loop Clusters operator is applied next. The subprocess of the Loop Clusters operator executes twice; once for each cluster. Have a look at the subprocess of the Loop Clusters

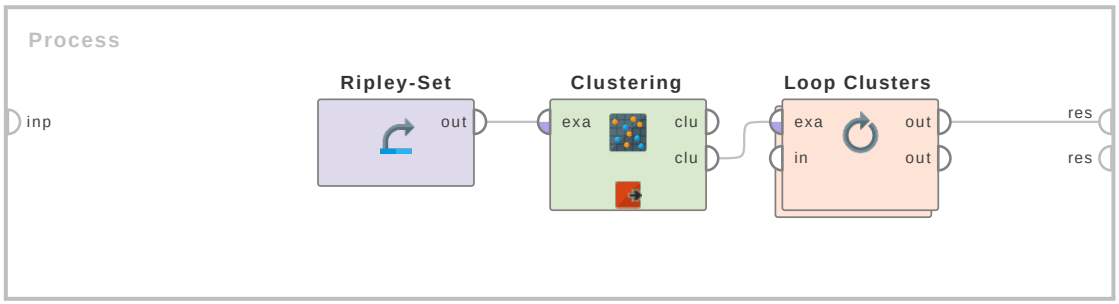
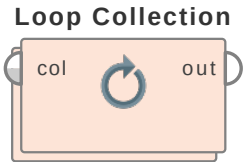



Figure 7.30: Tutorial process ‘Introduction to the Loop Clusters operator’.

operator. The Log operator is applied in the subprocess. A breakpoint is inserted before the Log operator so that you can see the examples of each iteration. In the first iteration you will see that all examples belong to cluster_1 while in the second iteration all examples belong to cluster_0.

# Loop Collection




This operator iterates over a collection of objects. It is a nested operator and its subprocess executes once for each object of the given collection.

## Description

Objects can be grouped into a collection using the Collect operator. In the Process View, collections are indicated by double lines. The Loop Collection operator loops over its subprocess once for every object in the input collection. The output of this operator is also a collection, any additional results of the subprocess can also be delivered through its output ports (as collections). If the *unfold* parameter is set to true then the output will be the union of all elements of the input collections.

Collections can be useful when you want to apply the same operations on a number of objects. The Collect operator will allow you to collect the required objects into a single collection, the Loop Collection operator will allow you to iterate over all collections and finally you can separate the input objects from collection by individually selecting the required element by using the Select operator.

## Input Ports

**collection *(col)*** This input port expects a collection. It is the output of the Collect operator in the attached Example Process.

## Output Ports

**output *(out)*** This operator can have multiple outputs. When one output is connected, another *output* port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object supplied at the first *output* port of the subprocess of the Loop Collection operator is delivered through the first output port of this operator. The objects are delivered as collections.

## Parameters

**set iteration macro *(boolean)*** This parameter specifies if a macro should be defined for the loop. The macro value will increment after every iteration. The name and start value of the macro can be specified by the *macro name* and *macro start value* parameters respectively.

**macro name *(string)*** This parameter is only available when the *set iteration macro* parameter is set to true. This parameter specifies the name of the macro.

**macro start value *(integer)*** This parameter is only available when the *set iteration macro* parameter is set to true. This parameter specifies the starting value of the macro. The value of the macro increments after every iteration of the loop.

**unfold *(boolean)*** This parameter specifies whether collections received at the input ports should be unfolded. If the *unfold* parameter is set to true then the output will be the union of all elements of the input collections.

## Tutorial Processes

### Introduction to collections

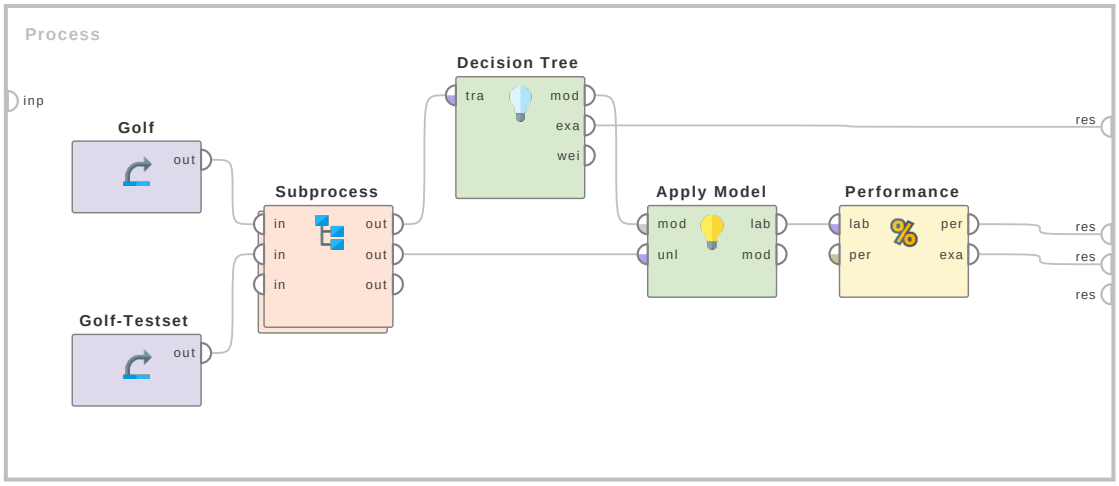


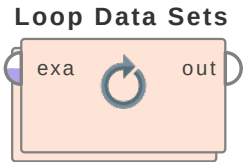

Figure 7.31: Tutorial process ‘Introduction to collections’.

This Example Process explains a number of important ideas related to collections. This Example Process shows how objects can be collected into a collection, then some preprocessing is applied on the collection and finally individual elements of the collection are separated as required.

The ‘Golf’ and ‘Golf-Testset’ data sets are loaded using the Retrieve operator. Both ExampleSets are provided as inputs to the Subprocess operator. The subprocess performs some preprocessing on the ExampleSets and then returns them through its output ports. The first output port returns the preprocessed ‘Golf’ data set which is then used as training set for the Decision Tree operator. The second output port delivers the preprocessed ‘Golf-Testset’ data set which is used as testing set for the Apply Model operator which applies the Decision Tree model. The performance of this model is measured and it is connected to the results port. The training and testing ExampleSets can also be seen in the Results Workspace.

Now have a look at the subprocess of the Subprocess operator. First of all, the Collect operator combines the two ExampleSets into a single collection. Note the double line output of the Collect operator which indicates that the result is a collection. Then the Loop Collection operator is applied on the collection. The Loop Collection operator iterates over the elements of the collection and performs some preprocessing (renaming an attribute in this case) on them. You can see in the subprocess of the Loop Collection operator that the Rename operator is used for changing the name of the Temperature attribute to ‘New Temperature’. It is important to note that this renaming is performed on both ExampleSets of the collection. The resultant collection is supplied to the Multiply operator which generates two copies of the collection. The first copy is used by the Select operator (with index parameter = 1) to select the first element of collection i.e. the preprocessed ‘Golf’ data set. The second copy is used by the second Select operator (with index parameter = 2) to select the second element of the collection i.e. the preprocessed ‘Golf-Testset’ data set.

# Loop Data Sets




This operator iterates over its subprocess for every ExampleSet given at its input ports.

## Description

The subprocess of the Loop Data Sets operator executes *n* number of times where *n* is the number of ExampleSets provided as input to this operator. You must have basic understanding of Subprocesses in order to understand this operator. For more information regarding subprocesses please study the Subprocess operator. For each input ExampleSet the Loop Data Sets operator executes the inner operators of the subprocess like an operator chain. This operator can be used to conduct a process consecutively on a number of different data sets. If the *only best* parameter is set to true then only the results generated during the iteration with best performance are delivered as output. For this option it is compulsory to attach a performance vector to the *performance* port in the subprocess of this operator. The Loop Data Sets operator uses this performance vector to select the iteration with best performance.

## Input Ports

**example set *(exa)*** This operator can have multiple inputs. When one input is connected, another *input* port becomes available which is ready to accept another ExampleSet (if any). The order of inputs remains the same. The ExampleSet supplied at the first *input* port of this operator is available at the first *input* port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at the subprocess level.

## Output Ports

**output *(out)*** The Loop Data Sets operator can have multiple *output* ports. When one output is connected, another *output* port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first *output* port of the subprocess is delivered at the first *output* of the outer process. Do not forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

## Parameters

**only best *(boolean)*** If the *only best* parameter is set to true then only the results generated during the iteration with the best performance are delivered as output. For this option it is compulsory to attach a performance vector to the *performance* port in the subprocess of this operator. The Loop Data Sets operator uses this performance vector to select the iteration with the best performance.

## Tutorial Processes

### Selecting the ExampleSet with best performance

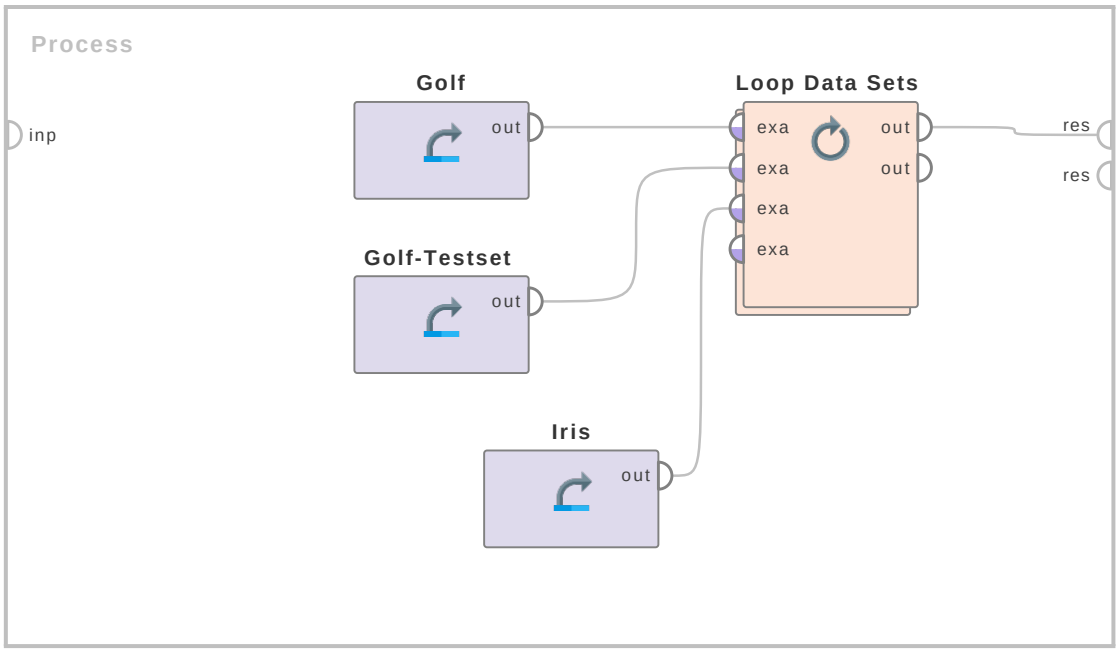
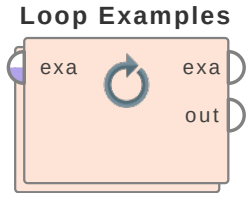



Figure 7.32: Tutorial process 'Selecting the ExampleSet with best performance'.

This Example Process explains the usage of the only best parameter of the Loop Data Sets operator. The 'Golf', 'Golf-Testset' and 'Iris' data sets are loaded using the Retrieve operator. All these ExampleSets are provided as input to the Loop Data Sets operator. Have a look at the subprocess of the Loop Data Sets operator. The Split Validation operator is used for training and testing a K-NN model on the given ExampleSet. The Split Validation operator returns the performance vector of the model. This performance vector is used by the Loop Data Sets operator for finding the iteration with the best performance. The results of the iteration with the best performance are delivered because the only best parameter is set to true.

When this process is executed, the 'Iris' data set is delivered as result. This is because the iteration with the 'Iris' data set had the best performance vector. If you insert a breakpoint after the Split Validation operator and run the process again, you can see that the 'Golf', 'Golf-Testset' and 'Iris' data sets have 25%, 50% and 93.33% accuracy respectively. As the iteration with 'Iris' data set had the best performance its results are returned by this operator (remember only best parameter is set to true). This operator can also return other objects like a model etc.

# Loop Examples




This operator iterates over its subprocess for all the examples of the given ExampleSet. The subprocess can access the index of the example of the current iteration by a macro.

## Description

The subprocess of the Loop Examples operator executes *n* number of times where *n* is the total number of examples in the given ExampleSet. In all iterations, the index of the example of the current iteration can be accessed using the macro specified in the *iteration macro* parameter. You need to have a basic understanding of macros in order to apply this operator. Please study the documentation of the Extract Macro operator for a basic understanding of macros. The Extract Macro operator is also used in the attached Example Process. For more information regarding subprocesses please study the Subprocess operator.

One important thing to note about this operator is the behavior of the *example set* output port of its subprocess. The subprocess is given the ExampleSet provided at the outer *example set* input port in the first iteration. If the *example set* output port of the subprocess is connected the ExampleSet delivered here in the last iteration will be used as input for the following iteration. If it is not connected the original ExampleSet will be delivered in all iterations.

It is important to note that the subprocess of the Loop Examples operator is executed for all examples of the given ExampleSet. If you want to iterate for possible values of a particular attribute please use the Loop Values operator. The subprocess of the Loop Values operator is executed for all possible values of the selected attribute. Suppose the selected attribute has three possible values and the ExampleSet has 100 examples. The Loop Values operator will iterate only three times (not 100 times); once for each possible value of the selected attribute. The Loop Examples operator, on the other hand, will iterate 100 times on this ExampleSet.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Extract Macro operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set *(exa)*** The ExampleSet that is connected at the *example set* output port of the inner subprocess is delivered through this port. If no ExampleSet is connected then the original ExampleSet is delivered.

**output *(out)*** The Loop Examples operator can have multiple *output* ports. When one output is connected, another *output* port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first *output* port of the subprocess is delivered at the first *output* of the outer process. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

## Parameters

**iteration macro *(string)*** This parameter specifies the name of the macro which holds the index of the example of the current iteration in each iteration.

## Tutorial Processes

### Subtracting the average of an attribute from all examples

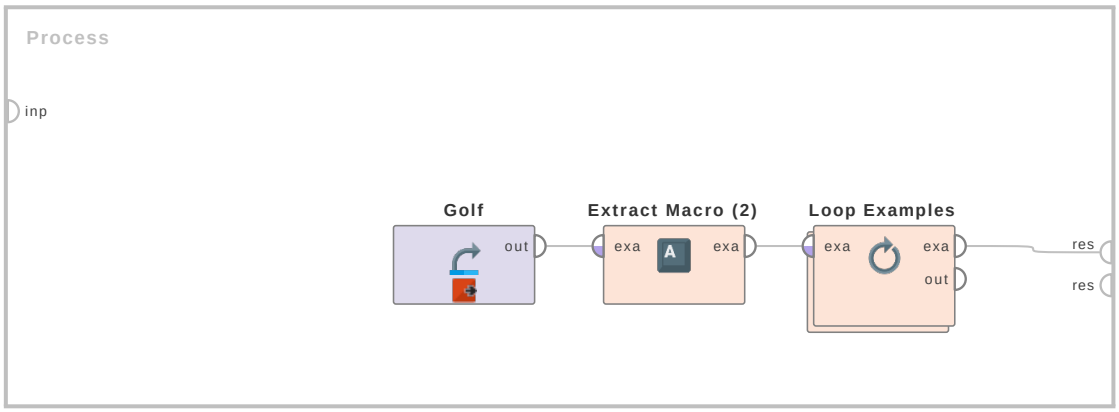


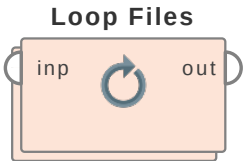

Figure 7.33: Tutorial process ‘Subtracting the average of an attribute from all examples’.

The ‘Golf’ data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. The Extract Macro operator is applied on it. The Extract Macro operator generates a macro named ‘avg_temp’ which stores the average value of the Temperature attribute of the ‘Golf’ data set. The Loop Examples operator is applied next. The Loop Examples operator iterates over all the examples of the ‘Golf’ data set and sets the value of the Temperature attribute in each example to the difference of the average value (stored in ‘avg_temp’ macro) from the existing value of the Temperature attribute. The iteration macro parameter of the Loop Examples operator is set to ‘example’. Thus in each iteration of the subprocess the index of the current example can be accessed by the ‘example’ macro.

Have a look at the subprocess of the Loop Examples operator. The Extract Macro operator is applied first to store the value of the Temperature attribute of the example of the current iteration in a macro named ‘temp’. Then the Generate Macro operator is applied to generate a new macro from the ‘temp’ macro. The name of the new macro is ‘new_temp’ and it stores the difference of the current temperature value (stored in ‘temp’ macro) and the average temperature value (stored in ‘avg_temp’ macro). Finally the Set Data operator is applied. The example index parameter is set to ‘%{example}’ to access the example of the current iteration. The value parameter is set to ‘%{new_temp}’ to store the value of the ‘new_temp’ macro in the current example.

This subprocess is executed once for each example and in every iteration it replaces value of the Temperature attribute of the example of that iteration with the value of ‘new_temp’ macro. The resultant ExampleSet can be seen in the Results Workspace. You can see that all values of the Temperature attribute have been replaced with new values.

# Loop Files




This operator executes the inner process tasks on every selected file.

## Description

With this operator you can select and filter files of a directory and execute the inner process on every selected file. Macros can be used to extract the file name, file path and file type.

In contrast to the core operator this advanced implementation allows the parallel execution of the inner processes. You can activate the *Background Processes* Panel and change the *Concurrency Level* or use the operator *Set Concurrency Level*.

## Input Ports

**in *(in )*** This port expects an *IOObject* which is passed on to the inner process without being altered. It will be reproduced if used. If you want to execute an inner loop, you have to connect the operators of the subprocess with the inner *loop* input port. In the first iteration it delivers the *IOObject* of the *in* input port.

## Output Ports

**output collector *(out)*** This port collects every result of the inner process. It will be reproduced if used.

## Parameters

**directory *(string)*** Select the directory from where to start scanning for files.

This parameter is only available if the *file set* port is not connected.

**filter type *(selection)*** Specifies how to filter file names. You can either use standard, command shell like glob filtering or a regular expression.

**filter by glob *(string)*** "Specifies a glob expression which is used as filter for the file and directory names.

Here is a short overview:

- * : any number of characters
- **: same as '*', but crosses directory boundaries. Useful to match complete paths.
- ? : matches exactly one char
- {}: contains collections that are separated by ','. The glob filter will try to match the string to any of the strings in the collection.

  : contains a range of chars or a single char (e.g.[a-z]).
- String(*): *
- String(?): \?
- String(**): **

**filter by regex** ***(string)*** ”Specifies a regular expression which is used as filter for the file and directory names, e.g. ‘a.*b’ for all files starting with ‘a’ and ending with ‘b’. Ignored if empty.”,

**recursive** ***(boolean)*** Set whether to recursively search every directory. If set to true, the operator will include files inside sub-directories (and sub-sub-directories ...) of the selected directory.

**enable macros** ***(boolean)*** If this parameter is enabled, you can name and extract three macros (for file name, file type and file folder)and use them in your subprocess.

**macro for file name** ***(string)*** If filled, a macro with this name will be set to the name of the current entry. To get access on the full path including the containing directory, combine this with the folder macro. Can be left blank.

**macro for file type** ***(string)*** Will be set to the file’s extension. Can be left blank.

**macro for file folder** ***(string)*** If filled, a macro with this name will be set to the containing folder of the current file. To get access on the full path you can combine this with the name macro. Can be left blank.

**reuse results** ***(boolean)*** Set whether to reuse the results of each iteration as the input of the next iteration. If set to true, the output of each iteration is used as input for the next iteration. Enabling this parameter will force the operator to NOT run in a parallel fashion. If set to false, the input of each iteration will be the original input.”,

**enable parallel execution** ***(boolean)*** This parameter enables the parallel execution of the inner processes. Please disable the parallel execution if you either run into memory problems or if you need an inner loop. The end result will be propagated to the outside process and can be used in the usual way.

## Tutorial Processes

### Generating an ExampleSet with names of all files in a directory

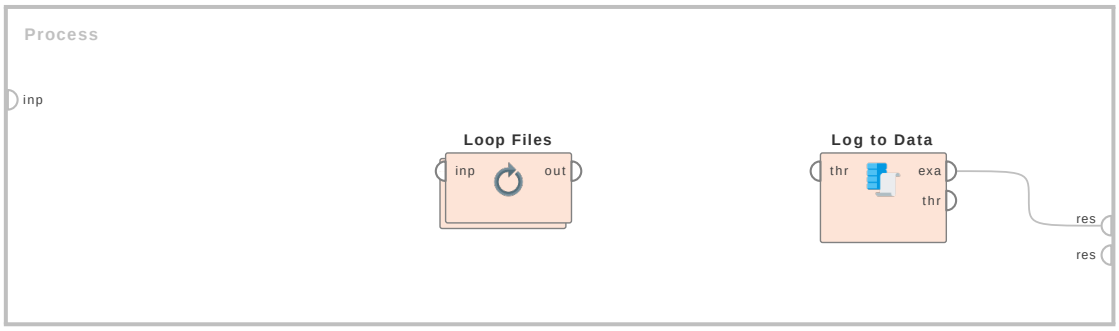
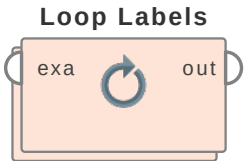



Figure 7.34: Tutorial process ‘Generating an ExampleSet with names of all files in a directory’.

This Example Process shows how the Loop Files operator can be used for iterating over files in a directory. You need to have at least a basic understanding of macros and logs in order to understand this process completely. The goal of this process is to simply provide the list of all

files in the specified directory in form of an ExampleSet. This process starts with the Loop Files operator. All parameters are used with default values. The Log operator is used in the subprocess of the Loop Files operator to store the name of the files in the log table in every iteration. The Provide Macro As Log Value operator is used before the Log operator to make the file name macro of the Loop Files operator available to the Log operator. The name of the file name macro (specified through the file name macro parameter) is 'file_name', therefore the macro name parameter of the Provide Macro As Log Value operator is set to 'file_name'. After the execution of the Loop Files operator, names of all the files in the specified directory are stored in the log table. To convert this data into an ExampleSet, the Log to Data operator is applied. The resultant ExampleSet is connected to the result port of the process and it can be viewed in the Results Workspace. As the path of the RapidMiner repository was specified in the directory parameter of the Loop Files parameter, the ExampleSet has the names of all the files in your RapidMiner repository. You can see names of all the files including files with '.properties' and '.rmp' extensions. If you want only the file names with '.rmp' extension, you can set the filter parameter to '.*rmp'.

# Loop Labels




This operator iterates over its subprocess for each attribute with label role in the input ExampleSet.

## Description

The Loop Labels operator is a nested operator i.e. it has a subprocess. The subprocess of the Loop Labels operator executes *n* number of times, where *n* is the number of attributes with label role in the given ExampleSet. The important thing to note here is that one ExampleSet cannot have more than one attributes with label role. The trick is that this operator executes for each attribute whose 'role name' starts with the string 'label'. So, if an ExampleSet has attributes with role label, label2 and label3, then the subprocess of this operator will execute three times on it. The result of this operator is a collection of objects. Please study the attached Example Process for better understanding.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is output of the Set Role operator in the attached Example Process.

## Output Ports

**out *(out)*** This operator can have multiple *out* output ports. When one output is connected, another *out* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *out* port of subprocess is delivered at the first *out* output port of the outer process. Do not forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

## Tutorial Processes

### Introduction to the Loop Labels operator

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has an attribute with label role i.e. the 'Play' attribute. The Set Role operator is applied on this ExampleSet to change the roles of the 'Wind' and 'Outlook' attributes to 'label2' and 'label3' respectively. A breakpoint is inserted here so that you can have a look at the ExampleSet before application of the Loop Labels operator. The Loop Labels operator is applied on the ExampleSet. There are three attributes with label roles therefore the subprocess of the Loop Labels operator will be executed three times; once for each label attribute. The results of each iteration will be merged into a collection which will be delivered as the result of this operator.

Have a look at the subprocess of the Loop Labels operator. The Decision Tree operator is applied there with default values of all parameters. In each iteration a Decision Tree will be generated for the current label. These trees will be returned in form of a collection by the Loop Labels operator. The resultant collection can be seen in the Results Workspace.

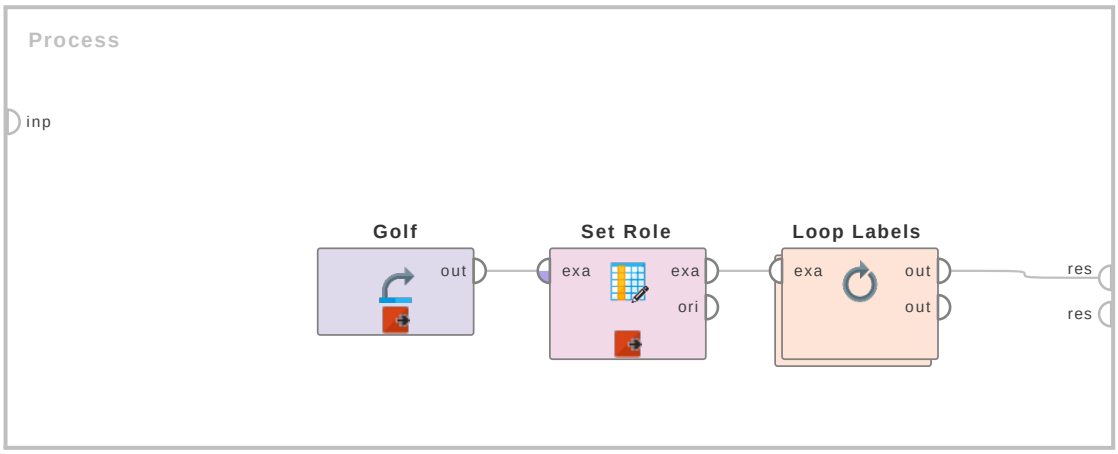
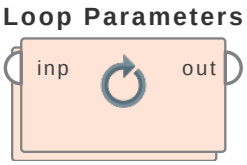



Figure 7.35: Tutorial process 'Introduction to the Loop Labels operator'.

# Loop Parameters




This Operator iterates over its subprocess for all the defined parameter combinations. The parameter combinations can be set by the wizard provided in parameters.

## Description

The Loop Parameters Operator is a nested Operator. It executes the subprocess for all combinations of selected values of the parameters. This can be very useful for plotting or logging purposes and sometimes for simply configuring the parameters for the inner Operators as a sort of meta step. Any results of the subprocess are delivered through the *output* ports. This Operator can be run in parallel.

The entire configuration of this Operator is done through the *edit parameter settings* parameter. Complete description of this parameter can be found in the parameters section.

The inner performance port can be used to log the performance of the inner subprocess. When it is connected, a log is created automatically to capture the number of the run, the parameter settings and the main criterion or all criteria of the delivered performance vector, depending on the parameter *log all criteria*. Please note that if no results are delivered at the end of a process, the log tables still can be seen in the Results View even if it is not automatically shown.

Please note that this Operator has two modes: synchronized and non-synchronized. They depend on the setting of the *synchronize* parameter. In the latter, all parameter combinations are generated and the subprocess is executed for each combination. In the synchronized mode, no combinations are created but the parameter values are treated as a list of combinations. For the iteration over a single parameter there is no difference between both modes. Please note that the number of parameter possibilities must be the same for all parameters in the synchronized mode. As an Example, having two boolean parameters A and B (both with true(t)/false(f) as possible parameter settings) will produce four combinations in non-synchronized mode (t/t, f/t, t/f, f/f) and two combinations in synchronized mode (t/t, f/f).

If the *synchronize* parameter is not set to true, selecting a large number of parameters and/or large number of steps (or possible values of parameters) results in a huge number of combinations. For example, if you select 3 parameters and 25 steps for each parameter then the total number of combinations would be above 17576 (i.e. 26 x 26 x 26). The subprocess is executed for all possible combinations. Running a subprocess for such a huge number of iterations will take a lot of time. So always carefully limit the parameters and their steps.

## Differentiation

- **Optimize Parameters (Grid)**

  The Optimize Parameters (Grid) Operator executes the subprocess for all combinations of selected values of the parameters and then delivers the optimal parameter values. The Loop Parameters Operator, in contrast to the optimization Operators, simply iterates through all parameter combinations. This might be especially useful for plotting purposes.

  See page **??** for details.

## Tutorial Processes

### Iterating through the parameters of the SVM Operator

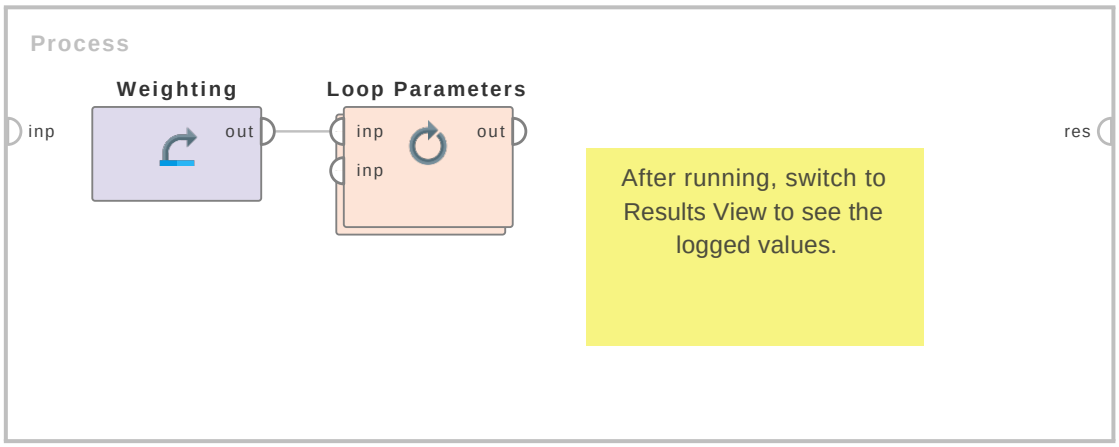



Figure 7.36: Tutorial process 'Iterating through the parameters of the SVM Operator'.

The 'Weighting' data set is loaded using the Retrieve Operator. The Loop Parameters Operator is applied on it. Have a look at the Edit Parameter Settings parameter of the Loop Parameters Operator. You can see in the Selected Parameters window that the C and gamma parameters of the SVM Operator are selected. Click on the SVM.C parameter in the Selected Parameters window, you will see that the range of the C parameter is set from 0.001 to 100000. 11 values are selected (in 10 steps) logarithmically. Now, click on the SVM.gamma parameter in the Selected Parameters window, you will see that the range of the gamma parameter is set from 0.001 to 1.5. 11 values are selected (in 10 steps) logarithmically. There are 11 possible values of two parameters, thus there are 121 ( i.e. 11 x 11) combinations. The subprocess will be executed for all combinations of these values because the synchronize parameter is set to false, thus it will iterate 121 times. In every iteration, the value of the C and/or gamma parameters of the

SVM(LibSVM) Operator is changed. The value of the C parameter is 0.001 in the first iteration. The value is increased logarithmically until it reaches 100000 in the last iteration. Similarly, the value of the gamma parameter is 0.001 in the first iteration. The value is increased logarithmically until it reaches 1.5 in the last iteration.

Have a look at the subprocess of the Loop Parameters Operator. First the data is split into two equal partitions using the Split Data Operator. The SVM (LibSVM) Operator is applied on one partition. The resultant classification model is applied using an Apply Model Operator on the other partition. The statistical performance of the SVM model on the testing partition is measured using the Performance (Classification) Operator. At the end the Loop Parameters Operator automatically logs the parameter settings and performance.

The log contains the following four things:

The iteration numbers of the Loop Parameters Operator are counted. This is stored in a column named 'Iteration'.

The classification error of the performance of the testing partition is logged in a column named 'classification error'.

The value of the C parameter of the SVM (LibSVM) Operator is stored in a column named 'SVM.C'.

The value of the gamma parameter of the SVM (LibSVM) Operator is stored in a column named 'SVM.gamma'.

Run the process and turn to the Results View. Now have a look at the values logged by the Loop Parameters Operator.

## Parameters

**edit parameter settings *(menu)*** The parameters are selected through the *edit parameter settings* menu. You can select the parameters and their possible values through this menu. This menu has an *Operators* window which lists all the operators in the subprocess of this Operator. When you click on any Operator in the *Operators* window, all parameters of that Operator are listed in the *Parameters* window. You can select any parameter through the arrow keys of the menu. The selected parameters are listed in the *Selected Parameters* window. Only those parameters should be selected for which you want to iterate the subprocess. This Operator iterates through parameter values in the specified range. The range of every selected parameter should be specified. When you click on any selected parameter (parameter in *Selected Parameters* window), the *Grid/Range* and *Value List* option is enabled. These options allow you to specify the range of values of the selected parameters. The *Min* and *Max* fields are for specifying the lower and upper bounds of the range respectively. As all values within this range cannot be checked, the *steps* field allows you to specify the number of values to be checked from the specified range. Finally the *scale* option allows you to select the pattern of these values. You can also specify the values in form of a list.

**error handling *(selection)*** This parameter allows you to select the method for handling errors occurring during the execution of the inner process. It has the following options:

- **fail_on_error** In case an error occurs, the execution of the process will fail with an error message.
- **ignore_error** In case an error occurs, the error will be ignored and the execution of the process will continue with the next iteration.

**log performance *(boolean)*** This parameter will only be visible if the inner performance port is connected. If it is connected, the main criterion of the performance vector will be automatically logged with the parameter set if this parameter is set to true.

**log all criteria *(boolean)*** This parameter allows for more logging. If set to true, all performance criteria will be logged.

**synchronize *(boolean)*** This Operator has two modes: synchronized and non-synchronized. They depend on the setting of this parameter. If it is set to false, all parameter combinations are generated and the inner Operators are applied for each combination. If it is set to true, no combinations are created but the parameter values are treated as a list of combinations. For the iteration over a single parameter there is no difference between both modes. Please note that the number of parameter possibilities must be the same for all parameters in the synchronized mode.

**enable parallel execution *(boolean)*** This parameter enables the parallel execution of the subprocess. Please disable the parallel execution if you run into memory problems.

## Input Ports

**input *(inp)*** This Operator can have multiple inputs. When one input is connected, another *input* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *input* port of this Operator is available at the first *input* port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at the subprocess level.

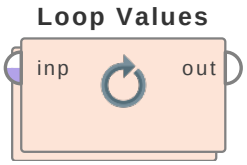
## Output Ports

**output *(out)*** Any results of the subprocess are delivered through the *output* ports. This Operator can have multiple outputs. When one *output* port is connected, another *output* port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first *output* port of the subprocess is delivered at the first *output*port of the Operator. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports.

## Related Documents

- **Optimize Parameters (Grid)** (page **??**)
- **Optimize Parameters (Quadratic)** (page 677)
- **Optimize Parameters (Evolutionary)** (page 669)

# Loop Values




This operator iterates over its subprocess for all the possible values of the selected attribute. The subprocess can access the attribute value of the current iteration by a macro.

## Description

The Loop Values operator has a parameter named *attribute* that allows you to select the required attribute of the input ExampleSet. Once the attribute is selected, the Loop Values operator applies its subprocess for each possible value of the selected attribute i.e. the subprocess executes *n* number of times where *n* is the number of possible values of the selected attribute. In all iterations the attribute value of the current iteration can be accessed using the macro specified in the *iteration macro* parameter. You need to have basic understanding of macros in order to apply this operator. Please study the documentation of the Extract Macro operator for basic understanding of macros. The Extract Macro operator is also used in the attached Example Process. For more information regarding subprocesses please study the Subprocess operator.

It is important to note that the subprocess of the Loop Values operator executes for all possible values of the selected attribute. Suppose the selected attribute has three possible values and the ExampleSet has 100 examples. The Loop Values operator will iterate only three times (not 100 times); once for each possible value of the selected attribute. This operator is usually applied on nominal attributes.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**output *(out)*** The Loop Values operator can have multiple outputs. When one output is connected, another *output* port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at first *output* port of subprocess is delivered at first *output* of the outer process. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

## Parameters

**attribute *(string)*** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *attribute* parameter if the meta data is known.

**iteration macro *(string)*** This parameter specifies the name of the macro which holds the current value of the selected attribute in each iteration.

**reuse results *(boolean)*** Set whether to reuse the results of each iteration as the input of the next iteration. If set to true, the output of each iteration is used as input for the next iteration. For obvious reasons, this will limit the loop to run in a single thread and not make

use of more CPU cores. If set to false, the input of each iteration will be the original input of the loop.

**enable parallel execution *(boolean)*** This parameter enables the parallel execution of the subprocess. Please disable the parallel execution if you run into memory problems.

## Tutorial Processes

### The use of the Loop Values operator in complex preprocessing

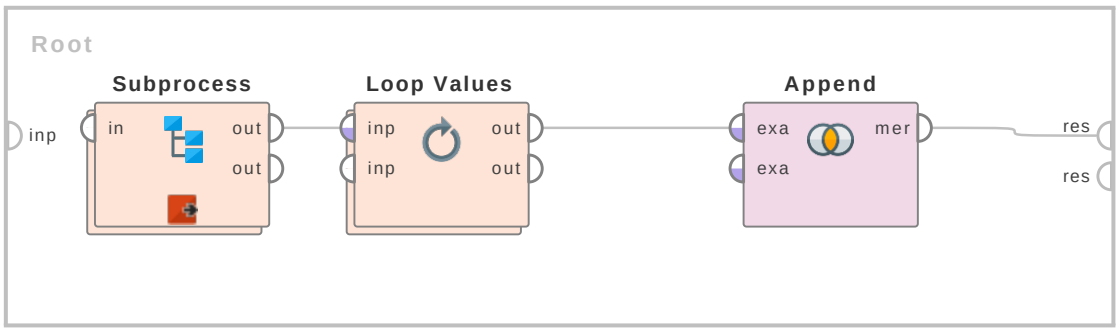



Figure 7.37: Tutorial process 'The use of the Loop Values operator in complex preprocessing'.

This Tutorial Process will cover a number of concepts of macros including redefining macros, the macro of the Loop Values operator and the use of the Extract Macro operator. This process starts with a subprocess which is used to generate data. What is happening inside this subprocess is not relevant to the use of macros, so it is not discussed here. A breakpoint is inserted after this subprocess so that you can view the ExampleSet. You can see that the ExampleSet has 12 examples and 2 attributes: 'att1' and 'att2'. 'att1' is nominal and has 3 possible values: 'range1', 'range2' and 'range3'. 'att2' has real values.

The Loop Values operator is applied on the ExampleSet. The attribute parameter is set to 'att1' therefore the Loop Values operator iterates over the values of the specified attribute (i.e. att1) and applies the inner operators on the given ExampleSet while the current value can be accessed via the macro defined by the iteration macro parameter which is set to 'loop_value', thus the current value can be accessed by specifying %{loop_value} in the parameter values. As att1 has 3 possible values, Loop Values will iterate 3 times, once for each possible value of att1.

Here is an explanation of what happens inside the Loop Values operator. It is provided with an ExampleSet as input. The Filter Examples operator is applied on it. The condition class parameter is set to 'attribute value filter' and the parameter string is set to 'att1 = %{loop_value}'. Note the use of the loop_value macro here. Only those examples are selected where the value of att1 is equal to the value of the loop_value macro. A breakpoint is inserted here so that you can view the selected examples. Then the Aggregation operator is applied on the selected examples. It is configured to take the average of the att2 values of the selected examples. This average value is stored in a new ExampleSet in the attribute named 'average(att2)'. A breakpoint is inserted here so that you can see the average of the att2 values of the selected examples. The Extract Macro operator is applied on this new ExampleSet to store this average value in a macro named 'current_average'. The originally selected examples are passed to the Generate Attributes operator that generates a new attribute named 'att2_abs_avg' which is defined by the expression 'abs(att2 - %{current_average})'. Note the use of the current_average macro here. Its value is

subtracted from all values of att2 and stored in a new attribute named ‘att2_abs_avg’. The Resultant ExampleSet is delivered at the output of the Loop Values operator. A breakpoint is inserted here so that you can see the ExampleSet with the ‘att2_abs_avg’ attribute. This output is fed to the Append operator in the main process. It merges the results of all the iterations into a single ExampleSet which is visible at the end of this process in the Results Workspace.

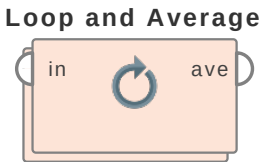
Here is what you see when you run the process.

ExampleSet generated by the first Subprocess operator. Then the process enters the Loop Value operator and iterates 3 times.

Iteration 1: ExampleSet where the ‘att1’ value is equal to the current value of the loop_value macro i.e. ‘range1’Average of ‘att2’ values for the selected examples. The average is -1.161.ExampleSet with ‘att2_abs_avg’ attribute for iteration 1. Iteration 2: ExampleSet where the ‘att1’ value is equal to the current value of the loop_value macro i.e. ‘range2’Average of ‘att2’ values for the selected examples. The average is -1.656.ExampleSet with ‘att2_abs_avg’ attribute for iteration 2.

Iteration 3: ExampleSet where the ‘att1’ value is equal to the current value of the loop_value macro i.e. ‘range3’Average of ‘att2’ values for the selected examples. The average is 1.340.ExampleSet with ‘att2_abs_avg attribute’ for iteration 3. Now the process comes out of the Loop Values operator and the Append operator merges the final ExampleSets of all three iterations into a single ExampleSet that you can see in the Results Workspace.

# Loop and Average




This operator iterates over its subprocess the specified number of times and delivers the average of the inner results.

## Description

The Loop and Average operator is a nested operator i.e. it has a subprocess. The subprocess of the Loop and Average operator executes *n* number of times, where *n* is the value of the *iterations* parameter specified by the user. The subprocess of this operator must always return a performance vector. These performance vectors are averaged and returned as result of this operator. For more information regarding subprocesses please study the Subprocess operator.

## Differentiation

- **Loop and Deliver Best** This operator iterates over its subprocess the specified number of times and delivers the results of the iteration that has the best performance. See page 882 for details.

## Input Ports

**in *(in )*** This operator can have multiple inputs. When one input is connected, another *in* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *in* port of this operator is available at the first *in* port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at the subprocess level.

## Output Ports

**averagable *(ave)*** This operator can have multiple *averagable* output ports. When one output is connected, another *averagable* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Average Vector delivered at the first *averagable* port of the subprocess is delivered at the first *averagable* output port of the outer process. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

## Parameters

**iterations *(integer)*** This parameter specifies the number of iterations of the subprocess of this operator.

**average performances only *(boolean)*** This parameter indicates if only performance vectors or all types of averagable result vectors should be averaged.

## Related Documents

- **Loop and Deliver Best** (page 882)

## Tutorial Processes

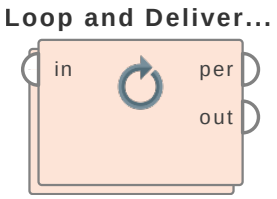
### Taking average of performance vectors


Figure 7.38: Tutorial process 'Taking average of performance vectors'.

The 'Golf' data set is loaded using the Retrieve operator. The Loop And Average operator is applied on it. The iterations parameter is set to 3; thus the subprocess of the Loop And Average operator will be executed three times. A performance vector will be generated in every iteration and the average of these performance vectors will be delivered as the result of this operator.

Have a look at the subprocess of the Loop And Average operator. The Split Validation operator is used for training and testing a Naive Bayes model. A breakpoint is inserted after the Split Validation operator so that the performance vector can be seen in each iteration. Run the process. You will see the performance vector of the first iteration. It has 25% accuracy. Keep continuing the process; you will see the performance vectors of the second and third iterations (with 75% and 100% accuracy respectively). The Loop and Average operator takes the average of these three results and delivers it through its output port. The average of these three results is 66.67% (i.e. (25% + 75% + 100%) / 3). The resultant average vector can be seen in the Results Workspace.

# Loop and Deliver Best




This operator iterates over its subprocess the specified number of times and delivers the results of the iteration that has the best performance.

## Description

The Loop and Deliver Best operator is a nested operator i.e. it has a subprocess. The subprocess of the Loop and Deliver Best operator executes *n* number of times, where *n* is the value of the *iterations* parameter specified by the user. The subprocess of this operator must always return a performance vector. The best performance vector from these performance vectors is returned as result of this operator. For more information regarding subprocesses please study the Subprocess operator.

## Differentiation

- **Loop and Average** This operator iterates over its subprocess the specified number of times and delivers the average of the inner results. See page 880 for details.

## Input Ports

**in *(in )*** This operator can have multiple inputs. When one input is connected, another *in* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *in* port of this operator is available at the first *in* port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at the subprocess level.

## Output Ports

**performance *(per)*** The best performance vector of all the performance vectors is returned from this port.

**out *(out)*** This operator can have multiple *out* output ports. When one output is connected, another *out* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *out* port of the subprocess is delivered at the first *out* port of the outer process. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

## Parameters

**iterations *(integer)*** This parameter specifies the number of iterations of the subprocess of this operator.

**enable timeout *(boolean)*** This parameter specifies if the processing should be aborted after the time specified in the *timeout* parameter. Please note that the processing is aborted after completion of the running iteration.

**timeout *(integer)*** The timeout (in minutes) is specified through this parameter.

## Related Documents

- **Loop and Average** (page 880)

## Tutorial Processes

### Delivering the best performance vector

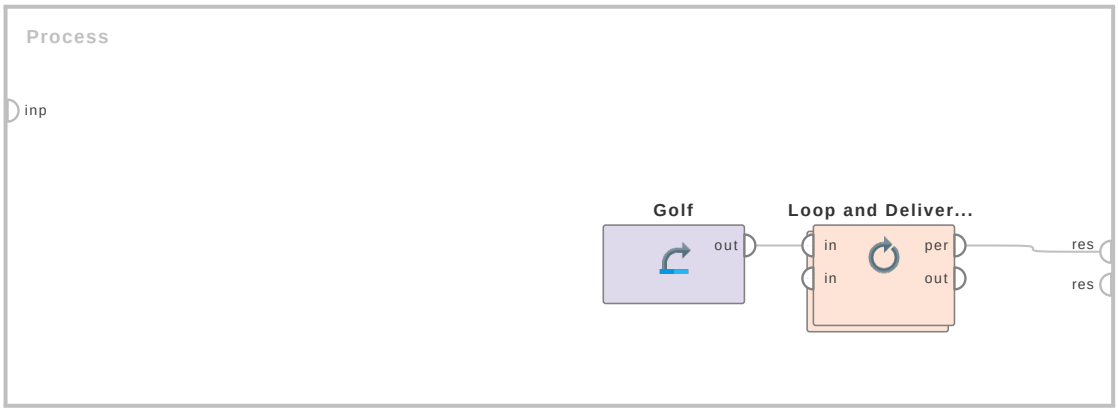


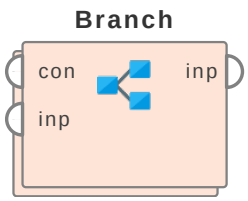

Figure 7.39: Tutorial process 'Delivering the best performance vector'.

The 'Golf' data set is loaded using the Retrieve operator. The Loop and Deliver Best operator is applied on it. The iterations parameter is set to 3; thus the subprocess of the Loop and Deliver Best operator will be executed three times. A performance vector will be generated in every iteration and the best of these performance vectors will be delivered as the result of this operator.

Have a look at the subprocess of the Loop and Deliver Best operator. The Split Validation operator is used for training and testing a Naive Bayes model. A breakpoint is inserted after the Split Validation operator so that the performance vector can be seen in each iteration. Run the process. You will see that the performance vector of the first iteration has 25% accuracy. Keep continuing the process; you will see the performance vectors of the second and third iterations (with 75% and 100% accuracy respectively). The Loop and Deliver Best operator returns the best of these three results through its output port. The best of these three results is the one with 100% accuracy. The resultant performance vector can be seen in the Results Workspace.

# 7.2.2 Branches

## Branch




This operator consists of two subprocesses but it executes only one subprocess at a time depending upon the condition. This operator is similar to the 'if-then-else' statement, where one of the two options is selected depending upon the results of the specified condition. It is important to have understanding of subprocesses in order to use this operator effectively.

### Description

The Branch operator tests the condition specified in the parameters (mostly through the *condition type* and *condition value* parameters) on the object supplied at the *condition* input port. If the condition is satisfied, the first subprocess i.e. the 'Then' subprocess is executed otherwise the second subprocess i.e. the 'Else' subprocess is executed.

It is very important to have a good understanding of use of subprocesses in RapidMiner to understand this operator completely. A subprocess introduces a process within a process. Whenever a subprocess is reached during a process execution, first the entire subprocess is executed. Once the subprocess execution is complete, the flow is returned to the process (the parent process). A subprocess can be considered as a small unit of a process, like in a process, all operators and combination of operators can be applied in a subprocess. That is why a subprocess can also be defined as a chain of operators that is subsequently applied. For more detail about subprocesses please study the Subprocess operator.

Double-click on the Branch operator to go inside and view the subprocesses. The subprocesses are then shown in the same Process View. Here you can see two subprocesses: 'Then' and 'Else' subprocesses. The 'Then' subprocess is executed if the condition specified in the parameters results true. The 'Else' subprocess is executed if the condition specified in the parameters results false. To go back to the parent process, click the blue-colored up arrow button in the Process View toolbar. This works like files and folders work in operating systems. Subprocesses can have subprocesses in them just like folders can have folders in them.

The Branch operator is similar to the Select Subprocess operator because they both have multiple subprocesses but only one subprocess is executed at a time. The Select Subprocess operator can have more than two subprocesses and the subprocess to be executed is specified in the parameters. On the contrary, The Branch operator has only two subprocesses and the subprocess to be executed depends upon the result of the condition specified in the parameters. The condition is specified through the *condition type* and *condition value* parameters. Macros can be provided in the *condition value* parameter. Thus the subprocess to be executed can be controlled by using macros. If this operator is placed in any Loop operator this operator will be executed multiple number of times. True power of this operator comes into play when it is used with other operators like various Macro and Loop operators. For example, if this operator is placed in any Loop operator and the *condition value* parameter is controlled by a macro then this operator can be used to dynamically change the process setup. This might be useful in order to test different layouts.

### Input Ports

**condition *(con)*** Any object can be supplied at this port. The condition specified in the parameters is tested on this object. If the condition is satisfied the 'Then' subprocess is executed otherwise the 'Else' subprocess is executed

**input *(inp)*** The Branch operator can have multiple inputs. When one input is connected, another *input* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *input* port of the Branch operator is available at the first *input* port of the nested chain (inside the subprocess).Don't forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

## Output Ports

**input *(inp)*** The Branch operator can have multiple outputs. When one output is connected, another *input* port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first *input* port of subprocess is delivered at the first *input* of the Branch operator. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

## Parameters

**condition type *(selection)*** The type of condition is selected through this parameter.

**condition value** The value of the selected condition type is specified through this parameter. The *condition type* and *condition value* parameters together specify the condition statement. This condition will be tested on the object provided at the *condition* input port.

**io object *(selection)*** This parameter is only available when the *condition type* parameter is set to 'input exists'. This parameter specifies the class of the object which should be checked for existence.

**return inner output *(boolean)*** This parameter indicates if the outputs of the inner subprocess should be delivered through this operator.

## Tutorial Processes

### Applying different subprocesses on Golf data set depending upon the performance value

The 'Golf' data set is loaded using the Retrieve operator. The Default Model operator is applied on it. The resultant model is applied on the 'Golf-Testset' data set through the Apply Model operator. The performance of this model is measured by the Performance operator. A breakpoint is inserted here so that you can have a look at this performance vector. You can see that its accuracy value is 64.29%. It is provided at the condition port of the Branch operator. Thus the condition specified in the parameters of the Branch operator will be tested on this performance vector. The 'Golf' data set is also provided to the Branch operator (through the first input port).

Now have a look at the subprocesses of the Branch operator. The 'Then' subprocess simply connects the condition port to the input port without applying any operator. Thus If the condition specified in the parameters is true, the condition object i.e. the performance vector will be delivered by the Branch operator. The 'Else' subprocess does not use the object at the condition port. Instead, it applies the K-NN operator on the object at the first input port i.e. the 'Golf' data set. Thus If the condition specified in the parameters is false, the K-NN operator will be applied on the object at the first input port i.e. the 'Golf' data set and the resultant model will be delivered by the Branch operator.

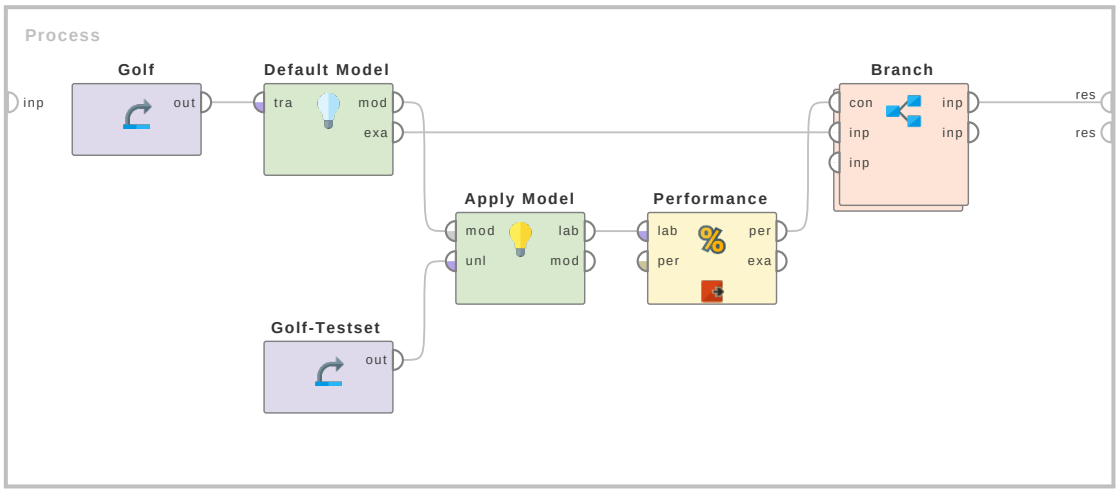
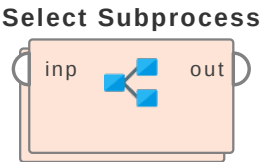



Figure 7.40: Tutorial process ‘Applying different subprocesses on Golf data set depending upon the performance value’.

Now have a look at the parameters of the Branch operator. The condition type parameter is set to ‘min performance value’ and the condition value parameter is set to 70. Thus if the performance of the performance vector is greater than 70, the condition will be true.

Overall in this process, The Default Model is trained on the ‘Golf’ data set, if its performance on the ‘Golf-Testset’ data set is more that 70% the performance vector will be delivered otherwise the K-NN model trained on the ‘Golf’ data set will be delivered.

# Select Subprocess




This operator consists of multiple subprocesses but it executes only one subprocess at a time. This operator is similar to a switch, where numerous options exist but only one option is selected at a time. It is important to have a good understanding of subprocesses in order to use this operator effectively.

## Description

It is very important to have a good understanding of the use of subprocesses in RapidMiner to understand this operator completely. A subprocess introduces a process within a process. Whenever a subprocess is reached during a process execution, first the entire subprocess is executed. Once the subprocess execution is complete, flow is returned to the process (the parent process). A subprocess can be considered as a small unit of a process, like in a process, all operators and combination of operators can be applied in a subprocess. That is why a subprocess can also be defined as a chain of operators that is subsequently applied. For more details about subprocesses please study the Subprocess operator.

Double-click on the Select Subprocess operator to go inside and view the subprocesses. The subprocesses are then shown in the same Process View. Here you can see the options to add or remove subprocesses. To go back to the parent process, click the blue-colored up arrow button in the Process View toolbar. This works like files and folders work in operating systems. Subprocesses can have subprocesses in them just like folders can have folders in them.

The Select Subprocess operator consists of multiple subprocesses but it executes only one subprocess at a time. The number of subprocesses can be easily controlled. You can easily add or remove subprocesses. The process to be executed is selected by the *select which* parameter. Macros can be provided in the *select which* parameter. Thus the subprocess to be executed can be controlled by using macros. If this operator is placed in any Loop operator this operator will be executed multiple number of times. The true power of this operator comes into play when it is used with other operators like various Macro and Loop operators. For example, if this operator is placed in any Loop operator and the *select which* parameter is controlled by a macro then this operator can be used to dynamically change the process setup. This might be useful in order to test different layouts, e.g. the gain by using different preprocessing steps or the quality of a certain learner.

## Input Ports

**input *(inp)*** The Select Subprocess operator can have multiple inputs. When one input is connected, another *input* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *input* port of the Select Subprocess operator is available at the first *input* port of the nested chain (inside the subprocess).Don't forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

## Output Ports

**output *(out)*** The Select Subprocess operator can have multiple outputs. When one output is connected, another *output* port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first *output* port of subprocess is delivered at the first *output* of the Select Subprocess operator. Don't

forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

## Parameters

**select which *(integer)*** This parameter indicates which subprocess should be applied. True power of this operator comes into play when the *select which* parameter is specified through a macro.

## Tutorial Processes

### Applying different classification operators on Golf data set using the Select Subprocess operator

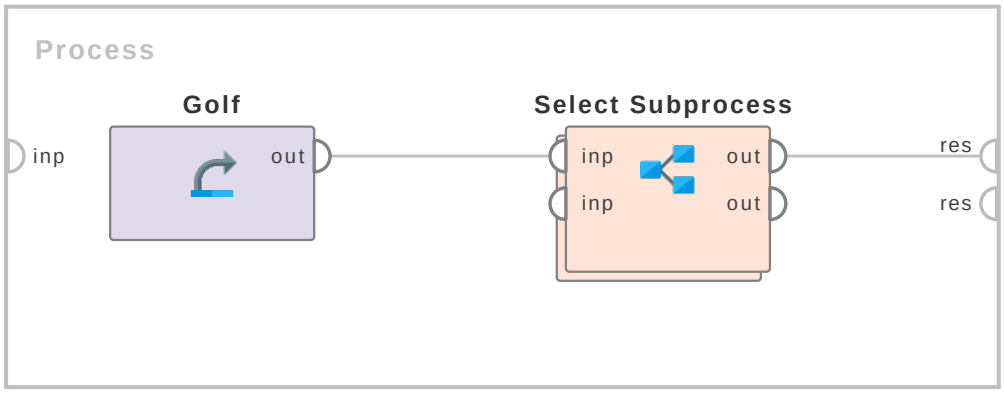



Figure 7.41: Tutorial process 'Applying different classification operators on Golf data set using the Select Subprocess operator'.

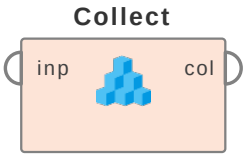
The 'Golf' data set is loaded using the Retrieve operator. The Select Subprocess operator is applied on it. Double-click on the Select Subprocess operator to see the subprocesses in it. As you can see, there are four subprocesses:

Subprocess 1: The k-NN operator is applied on the input and the resulting model is passed to the output. Subprocess 2: The Naive Bayes operator is applied on the input and the resulting model is passed to the output. Subprocess 3: The Decision Tree operator is applied on the input and the resulting model is passed to the output. Subprocess 4: The input is directly connected to the output.

Only one of these subprocesses can be executed at a time. The subprocess to be executed can be controlled by the select which parameter. The select which parameter is set to 1, thus the first subprocess will be executed. When you run the process you will see the model created by the k-NN operator in the Results workspace. To execute the second subprocess set the select which parameter to 2 and run the process again. You will see the model generated by the Naive Bayes operator in the Results Workspace. To execute the third subprocess set the select which parameter to 3 and run the process again. You will see the model generated by the Decision Tree operator in the Results Workspace. To execute the fourth subprocess set the select which parameter to 4 and run the process again. Now you will see the 'Golf' data set in the Results Workspace because no operator was applied in the fourth subprocess.

# 7.2.3 Collections

## Collect




This operator combines multiple input objects into a single collection.

### Description

The Collect operator combines a variable number of input objects into a single collection. It is important to know that all input objects should be of the same IOObject class. In the Process View, collections are indicated by double lines. If the input objects are collections themselves then the output of this operator would be a collection of collections. However if the *unfold* parameter is set to true then the output will be the union of all elements of the input collections. After combining objects into a collection, the Loop Collection operator can be used to iterate over this collection. The Select operator can be used to retrieve the required element of the collection.

Collections can be useful when you want to apply the same operations on a number of objects. The Collect operator will allow you to collect the required objects into a single collection, the Loop Collection operator will allow you to iterate over all collections and finally you can separate the input objects from collection by individually selecting the required element by using the Select operator.

### Input Ports

**input *(inp)*** This operator can have multiple inputs. When one input is connected, another *input* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *input* port of the Collect operator becomes the first element of the resultant collection. It is important to note that all input objects should be of the same IOObject class.

### Output Ports

**collection *(col)*** All the input objects are combined into a single collection and the resultant collection is delivered through this port.

### Parameters

**unfold *(boolean)*** This parameter is only applicable when the input objects are collections. This parameter specifies whether collections received at the input ports should be unfolded. If the input objects are collections themselves and the *unfold* parameter is set to false, then the output of this operator would be a collection of collections. However if the *unfold* parameter is set to true then the output will be the union of all elements of the input collections.

# Tutorial Processes

## Introduction to collections

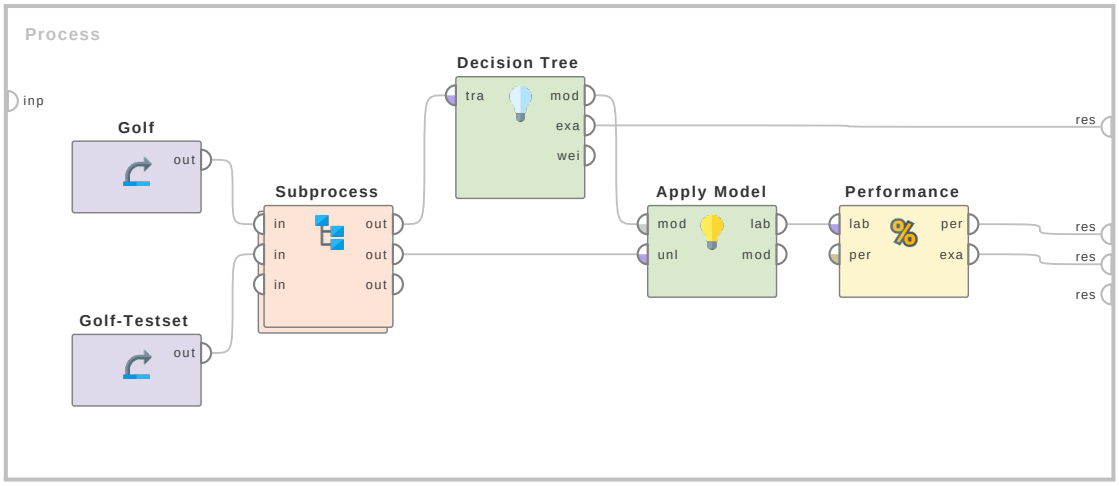


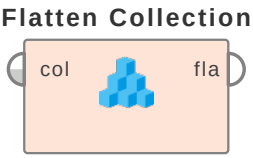

Figure 7.42: Tutorial process ‘Introduction to collections’.

This Example Process explains a number of important ideas related to collections. This Example Process shows how objects can be collected into a collection, then some preprocessing is applied on the collection and finally individual elements of the collection are separated as required.

The ‘Golf’ and ‘Golf-Testset’ data sets are loaded using the Retrieve operator. Both ExampleSets are provided as inputs to the Subprocess operator. The subprocess performs some preprocessing on the ExampleSets and then returns them through its output ports. The first output port returns the preprocessed ‘Golf’ data set which is then used as training set for the Decision Tree operator. The second output port delivers the preprocessed ‘Golf-Testset’ data set which is used as testing set for the Apply Model operator which applies the Decision Tree model. The performance of this model is measured and it is connected to the results port. The training and testing ExampleSets can also be seen in the Results Workspace.

Now have a look at the subprocess of the Subprocess operator. First of all, the Collect operator combines the two ExampleSets into a single collection. Note the double line output of the Collect operator which indicates that the result is a collection. Then the Loop Collection operator is applied on the collection. The Loop Collection operator iterates over the elements of the collection and performs some preprocessing (renaming an attribute in this case) on them. You can see in the subprocess of the Loop Collection operator that the Rename operator is used for changing the name of the Temperature attribute to ‘New Temperature’. It is important to note that this renaming is performed on both ExampleSets of the collection. The resultant collection is supplied to the Multiply operator which generates two copies of the collection. The first copy is used by the Select operator (with index parameter = 1) to select the first element of collection i.e. the preprocessed ‘Golf’ data set. The second copy is used by the second Select operator (with index parameter = 2) to select the second element of the collection i.e. the preprocessed ‘Golf-Testset’ data set.

# Flatten Collection




This operator receives a 'collection of collections' and unions all content into a single collection.

## Description

The Flatten Collection operator receives a 'collection of collections' and unions the content of each collection into a single collection. For example, if your collection object contains three collections which contain 5 objects each; this operator will return a single collection with 15 objects. Operators like the Collect operator can produce collections and 'collections of collections'. In the Process View, collections are indicated by double lines.

Collections can be useful when you want to apply the same operations on a number of objects. The Collect operator allows you to collect the required objects into a single collection, the Loop Collection operator allows you to iterate over all collections and finally you can separate the input objects from a collection by individually selecting the required element by using the Select operator.

## Input Ports

**collection *(col)*** This port expects a collection object. This operator can be useful when this object is a collection of collections.

## Output Ports

**flat *(fla)*** The given collection object is flattened and delivered through this port.

## Tutorial Processes

### Introduction to Flatten Collection operator

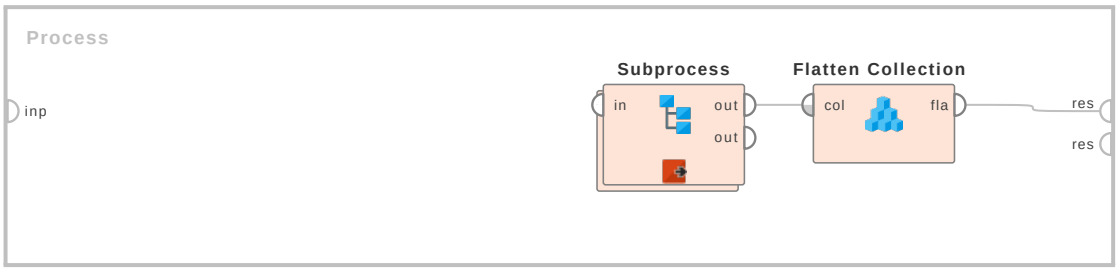
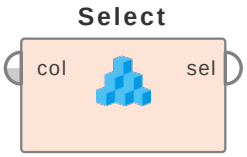



Figure 7.43: Tutorial process 'Introduction to Flatten Collection operator'.

This Example Process starts with the Subprocess operator which delivers a collection of collections. A breakpoint is inserted here so that you can have a look at this object. You can see that this object has two collections. These collections have 3 ExampleSets each. The Flatten

Collection operator is applied on this object to flatten the collection into a single collection of 6 ExampleSets. The resultant flattened collection object can be seen in the Results Workspace.

# Select




This operator returns the specified single object from the given collection of objects.

## Description

Operators like the Collect operator combine a variable number of input objects into a single collection. In the Process View, collections are indicated by double lines. The Select operator can be used for selecting an individual object from this collection. The *index* parameter specifies the index of the required object. If the objects of the given collection are collections themselves the output of this operator would be the collection at the specified index. However if the *unfold* parameter is set to true the index refers to the index in the flattened list, i.e. the list obtained from the input list by replacing all nested collections by their elements.

Collections can be useful when you want to apply the same operations on a number of objects. The Collect operator will allow you to collect the required objects into a single collection, the Loop Collection operator will allow you to iterate over all collections and finally you can separate the input objects from a collection by individually selecting the required element by using the Select operator.

## Input Ports

**collection *(col)*** This port expects a collection of objects as input. Operators like the Collect operator combine a variable number of input objects into a single collection.

## Output Ports

**selected *(sel)*** The object at the index specified by the *index* parameter is returned through this port.

## Parameters

**index *(integer)*** This parameter specifies the index of the required object within the collection of objects.

**unfold *(boolean)*** This parameter is only applicable when the objects of the given collection are collections themselves. This parameter specifies whether collections received at the input ports should be considered unfolded for selection. If the input objects are collections themselves and the *unfold* parameter is set to false, then the *index* parameter will refer to the collection at the specified index. However if the *unfold* parameter is set to true then the index refers to the index in the flattened list, i.e. the list obtained from the input list by replacing all nested collections by their elements.

## Tutorial Processes

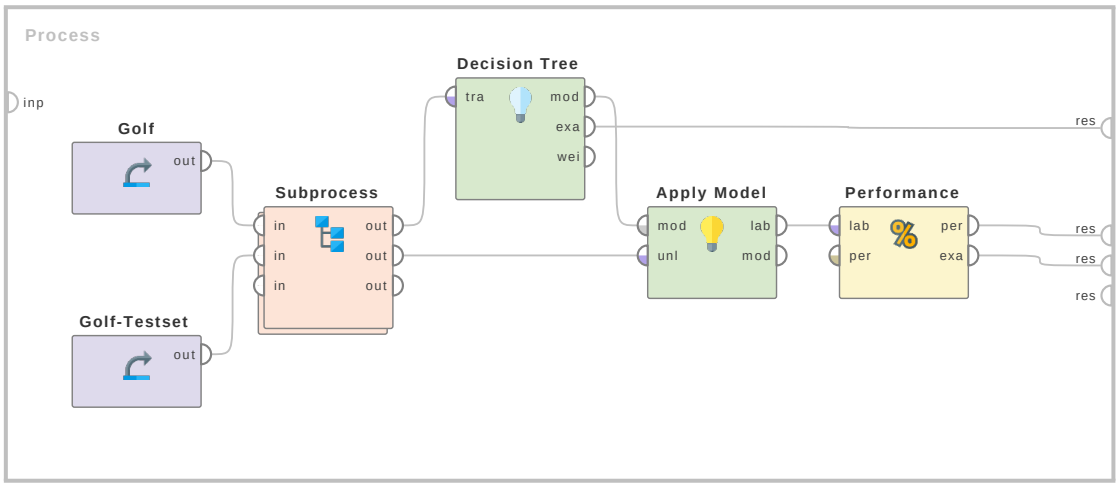



Figure 7.44: Tutorial process 'Introduction to collections'.

### Introduction to collections

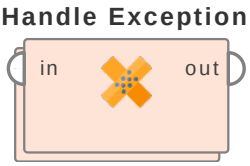
This Example Process explains a number of important ideas related to collections. It shows how objects can be collected into a collection, then some preprocessing is applied on the collection and finally individual elements of the collection are separated as required.

The 'Golf' and 'Golf-Testset' data sets are loaded using the Retrieve operator. Both ExampleSets are provided as inputs to the Subprocess operator. The subprocess performs some preprocessing on the ExampleSets and then returns them through its output ports. The first output port returns the preprocessed 'Golf' data set which is then used as training set for the Decision Tree operator. The second output port delivers the preprocessed 'Golf-Testset' data set which is used as testing set for the Apply Model operator which applies the Decision Tree model. The performance of this model is measured and it is connected to the results port. The training and testing ExampleSets can also be seen in the Results Workspace.

Now have a look at the subprocess of the Subprocess operator. First of all, the Collect operator combines the two ExampleSets into a single collection. Note the double line output of the Collect operator which indicates that the result is a collection. Then the Loop Collection operator is applied on the collection. The Loop Collection operator iterates over the elements of the collection and performs some preprocessing (renaming an attribute in this case) on them. You can see in the subprocess of the Loop Collection operator that the Rename operator is used for changing the name of the Temperature attribute to 'New Temperature'. It is important to note that this renaming is performed on both ExampleSets of the collection. The resultant collection is supplied to the Multiply operator which generates two copies of the collection. The first copy is used by the Select operator (with index parameter = 1) to select the first element of the collection i.e. the preprocessed 'Golf' data set. The second copy is used by the second Select operator (with index parameter = 2) to select the second element of the collection i.e. the preprocessed 'Golf-Testset' data set.

# 7.2.4 Exceptions

## Handle Exception




This is a nested operator that is used for exception handling. This operator executes the operators in its *Try* subprocess and returns its results if there is no error. If there is an error in the *Try* subprocess, the *Catch* subprocess is executed instead and its results are returned.

### Description

The Handle Exception operator is a nested operator i.e. it has two subprocesses: *Try* and *Catch*. This operator first tries to execute the *Try* subprocess. If there is no error i.e. the execution is successful, then this operator returns the results of the *Try* subprocess. In case there is any error the process is not stopped. Instead, the *Catch* subprocess is executed and its results are delivered by this operator. The error message can be saved using the *exception macro* parameter. This Try/Catch concept is like the exception handling construct used in many programming languages. You need to have a basic understanding of macros if you want to save the error message using the *exception macro*. Please study the documentation of the Extract Macro operator for basic understanding of macros. For more information regarding subprocesses please study the Subprocess operator. Please use this operator with care since it will also cover unexpected errors.

### Input Ports

**in *(in )*** This operator can have multiple inputs. When one input is connected, another *in* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *in* port of this operator is available at the first *in* port of the nested chain (inside the subprocess). Do not forget to connect all inputs in the correct order. Make sure that you have connected the right number of ports at the subprocess level.

### Output Ports

**out *(out)*** This operator can have multiple *out* ports. When one output is connected, another *out* port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first *out* port of the subprocess is delivered at the first *out* port of the outer process. Don't forget to connect all outputs in the correct order. Make sure that you have connected the right number of ports at all levels of the chain.

### Parameters

**exception macro *(string)*** This parameter specifies the name of the macro which will store the error message (if any). This macro can be accessed by other operators by using '%{macro_name}' syntax.

## Tutorial Processes

### Using Try/Catch for exception handling

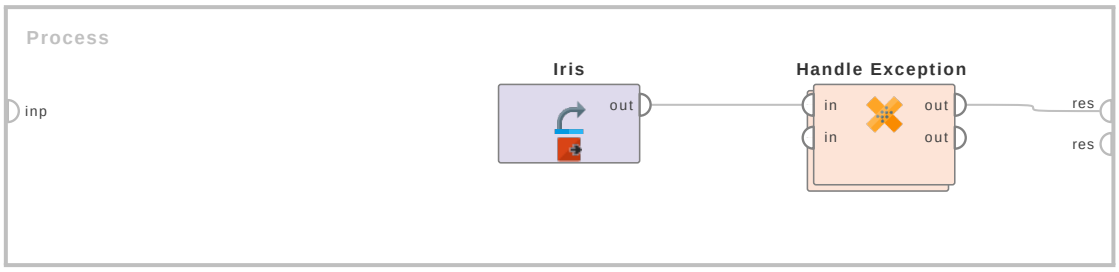



Figure 7.45: Tutorial process 'Using Try/Catch for exception handling'.

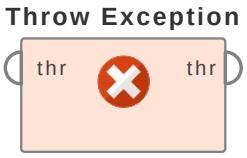
The goal of this Example Process is to deliver the 'Iris' data set after renaming its attributes. In case there is an error, the original 'Iris' data set should be delivered along with the error message.

The 'Iris' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that there are four regular attributes a1, a2, a3 and a4. The Handle Exception operator is applied next. Have a look at the subprocesses of this operator. The Rename operator is applied in the Try subprocess for renaming the attributes. The old name and new name parameters of the Rename operator are deliberately left empty which is an error because these are mandatory parameters. The Catch subprocess takes the original ExampleSet and applies the Log operator and delivers the ExampleSet without modifications to the output. The Log operator is used for recording the error message in case there is an error.

Execute the process and switch to the Results Workspace. You can see that the attributes of the ExampleSet have not been renamed. This is because there was an error in the Try subprocess, therefore it was not executed. The error message can be seen in the log which says that values for mandatory parameters of the Rename operator were not provided. The original ExampleSet can be seen in the Results Workspace because when the Handle Exception operator encountered an error in the Try subprocess, it stopped its execution and executed the Catch subprocess instead and delivered its results.

Now set the old name and new name parameters of the Rename operator to 'label' and 'new_label' respectively and run the process again. This time you can see that the attributes have been renamed and there is no error message. This is so because the Handle Exception operator first tried to execute the Try subprocess. Since there was no error in it, its results were delivered and the Catch subprocess was not executed.

# Throw Exception




This operator throws an exception every time it is executed.

## Description

The Throw Exception operator will throw an exception with an user-defined message as soon as it is executed. This will cause the process to fail. It can be useful if e. g. a certain result is equal to a failure.

## Input Ports

**through *(thr)*** Data delivered to this port will be passed to the output port without any modification. Whenever an input port gets occupied, two new input and output ports become available. The order remains the same. Data delivered at the first input port is available at the first output port. It's not necessary to connect this port, the exception will be thrown anyway.

## Output Ports

**through *(thr)*** Provides the data which was delivered at the corresponding input port without any changes. Whenever an input port gets occupied, two new input and output ports become available. The order remains the same. Data delivered at the first input port is available at the first output port. It's not necessary to connect this port, the exception will be thrown anyway.

## Parameters

**message *(string)*** The error message that should be shown/logged is specified through this parameter.

## Tutorial Processes

### Throw exeption if no examples passed

The 'Iris' data set is loaded with the Retrieve operator. The Filter Examples operator is applied on the data and filters examples of the attribute a1 that have a value greater than 10.

The ExampleSet is passed on to a Branch operator. If there is at least one example, the data is passed on without changing. If there are zero examples, the Throw Exception operator makes the process fail with the entered message. Because the filter condition applies on no example of the data set, the process will fail i.e. the Throw Exeption operator will be executed.

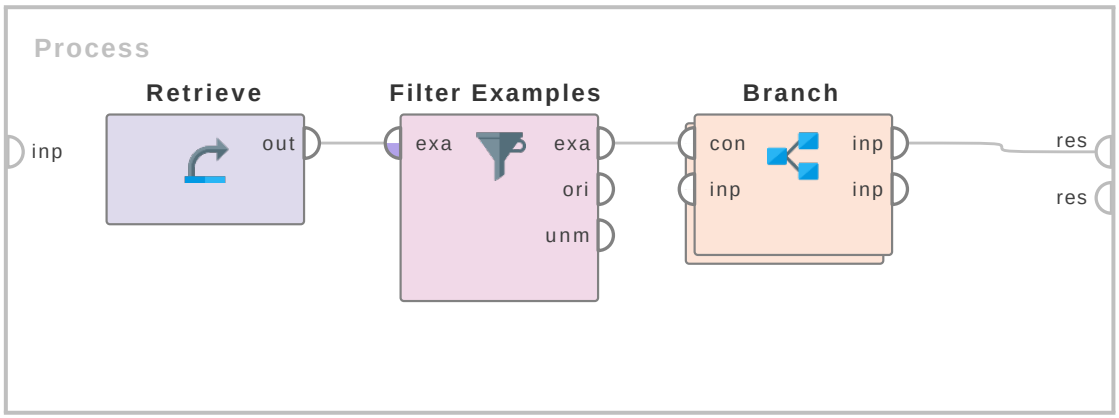
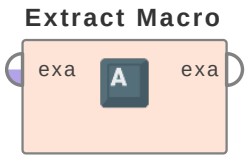



Figure 7.46: Tutorial process 'Throw exeption if no examples passed'.

# 7.3 Macros

## Extract Macro




This operator can be used to define a single macro which can be used by *%{macro_name}* in parameter values of succeeding operators of the current process. The macro value will be derived from the input ExampleSet. A macro can be considered as a value that can be used by all operators of the current process that come after the macro has been defined. This operator can also be used to re-define an existing macro.

### Description

This operator can be used to define a single macro which can be used in parameter values of succeeding operators of the current process. Once the macro has been defined, the value of that macro can be used as parameter values in coming operators by writing the macro name in *%{macro_name}* format in the parameter value where 'macro_name' is the name of the macro specified when it was defined. In the Extract Macro operator the macro name is specified by the *macro* parameter. The macro will be replaced in the value strings of parameters by the macro's value. This operator can also be used to re-define an existing macro.

This operator sets the value of a single macro from properties of a given input ExampleSet. This includes properties like the number of examples or number of attributes of the input ExampleSet. Specific data value of the input ExampleSet can also be used to set the value of the macro which can be set using various statistical properties of the input ExampleSet e.g. average, min or max value of an attribute. All these options can be understood by studying the parameters and the attached Example Processes. The Set Macro operator can also be used to define a macro but it does not set the value of the macro from properties of a given input ExampleSet.

#### Macros

A macro can be considered as a value that can be used by all operators of the current process that come after it has been defined. Whenever using macros, make sure that the operators are

in the correct sequence. It is compulsory that the macro should be defined before it can be used in parameter values. The macro is one of the advanced topics of RapidMiner, please study the attached Example Processes to develop a better understanding of macros.

There are also some predefined macros:

- *%{process_name}*: will be replaced by the name of the process (without path and extension)
- *%{process_file}*: will be replaced by the file name of the process (with extension)
- *%{process_path}*: will be replaced by the complete absolute path of the process file
- Several other short macros also exist, e.g. *%{a}* for the number of times the current operator was applied.

Please note that other operators like many of the loop operators (e.g. Loop Values , Loop Attributes) also add specific macros.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. The macro value will be extracted from this ExampleSet

## Output Ports

**example set output *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators. It is not compulsory to attach this port to any other port, Macro value is set even if this port is left without connections.

## Parameters

**macro *(string)*** This parameter is used to name the macro and can be accessed in succeeding operators of current process by writing the macro name in *%{macro_name}* format, where 'macro_name' is the name of the macro specified by this parameter.

**macro type** This parameter indicates the way the input ExampleSet should be used to define the macro.

- **number_of_examples** If this option is selected, the macro value is set to the total number of examples in the input ExampleSet.
- **number_of_attributes** If this option is selected, the macro value is set to the total number of attributes in the input ExampleSet.
- **data_value** If this option is selected, the macro value is set to the value of the specified attribute at the specified index. The attribute is specified using the *attribute name* parameter and the index is specified using the *example index* parameter.
- **statistics** If this option is selected, the macro value is set to the value obtained by applying the selected statistical operation on the specified attribute. The attribute is specified using the *attribute name* parameter and the statistical operator is selected using the *statistics* parameter.

**statistics** This parameter is only available when the *macro type* parameter is set to 'statistics'. This parameter allows you to select the statistical operator to be applied on the attribute specified by the *attribute name* parameter.

**attribute name** ***(string)*** This parameter is only available when the *macro type* parameter is set to 'statistics' or 'data value'. This parameter allows you to select the required attribute.

**attribute value** ***(string)*** This parameter is only available when the *macro type* parameter is set to 'statistics' and the *statistics* parameter is set to 'count'. This parameter is used to specify a particular value of the specified attribute. The macro value will be set to the number of occurrences of this value in the specified attribute. The attribute is specified by the *attribute name* parameter.

**example index** ***(integer)*** This parameter is only available when the *macro type* parameter is set to 'data value'. This parameter allows you to select the index of the required example of the attribute specified by the *attribute name* parameter and the optional *additional macros* parameter.

**additional macros** This parameter is only available when the *macro type* parameter is set to 'data value'. This optional parameter allows you to add an unlimited amount of additional macros. Note that the value for the *example index* parameter is used for all macros in this list.

## Related Documents

- **Extract Macro from Annotation** (page **??**)

## Tutorial Processes

### Introduction to the Extract Macro operator

This is a very basic process that demonstrates the use of macros and the Extract Macro operator. The 'Golf' data set is loaded using the Retrieve operator. The Extract Macro operator is applied on it. The macro is named 'avg_temp'. The macro type parameter is set to 'statistics', the statistics parameter is set to 'average' and the attribute name parameter is set to 'Temperature'. Thus the value of the avg_temp macro is set to the average of values of the 'Golf' data set's Temperature attribute. Which in all 14 examples of the 'Golf' data set is 73.571. Thus the value of the avg_temp macro is set to 73.571. In this process, wherever %{avg_temp} is used in parameter values, it will be replaced by the value of the avg_temp macro i.e. 73.571. Note that the output port of the Extract Macro operator is not connected to any other operator but still the avg_temp macro has been created.

The 'Golf-Testset' data set is loaded using the Retrieve operator. The Filter Examples operator is applied on it. The condition class parameter is set to 'attribute value filter'. The parameter string parameter is set to 'Temperature $>$ %{avg_temp}'. Note the use of the avg_temp macro. When this process is run, %{avg_temp} will be replaced by the value of the avg_temp macro i.e. 73.571. Thus only those examples of the Golf-Testset data set will make it to the output port where the value of the Temperature attribute is greater than average value of the Temperature attribute values of the Golf data set (i.e. 73.571). You can clearly verify this by seeing the results in the Results Workspace.

### Redefining a macro using the Extract Macro operator

The focus of this Example Process is to show how macros can be redefined using the Extract Macro operator. This process is almost the same as the first Example Process. The only difference is that after the avg_temp macro has been defined, the same macro is redefined using the 'Golf' data set and the Extract Macro operator. The 'Golf' data set is loaded again and it is provided to

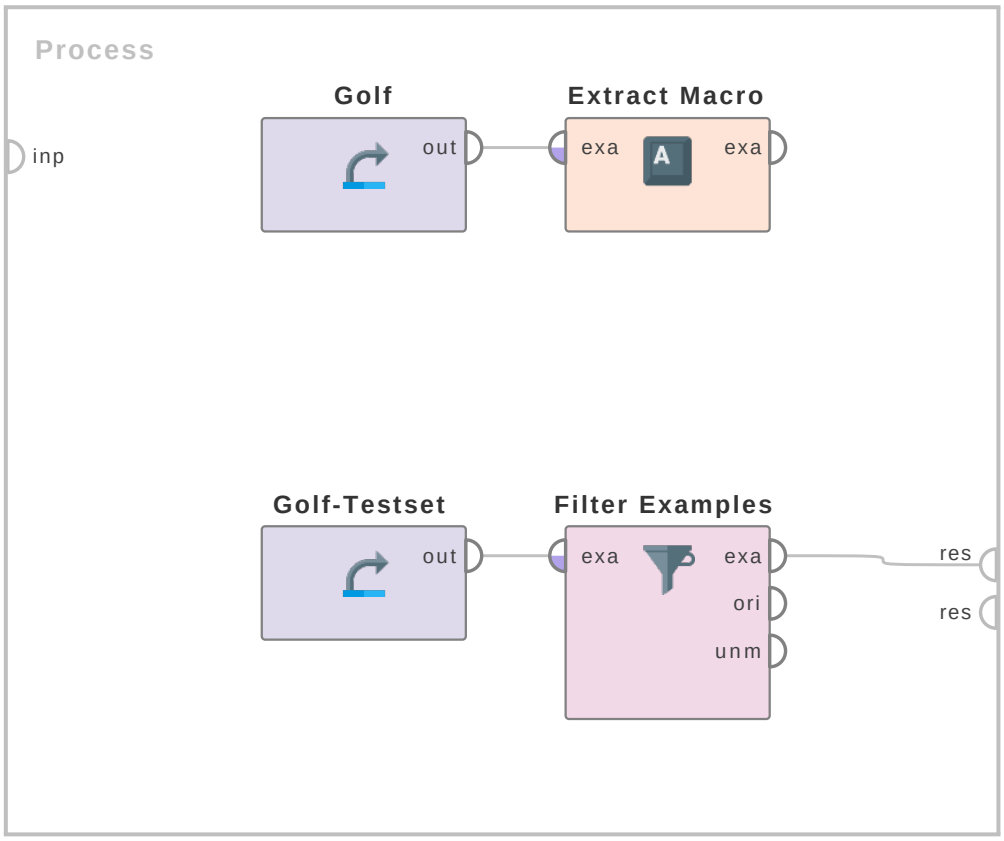



Figure 7.47: Tutorial process 'Introduction to the Extract Macro operator'.

the second Extract Macro operator. In this Extract Macro operator the macro parameter is set to 'avg_temp' and the macro type parameter is set to 'number of examples'. As the avg_temp macro already exists, no new macro is created; the already existing macro is redefined. As the number of examples in the 'Golf' data set is 14, avg_temp is redefined as 14. Thus in the Filter Examples operator the value of the Temperature attribute of the 'Golf-Testset' data set is compared with 14 instead of 73.571. This can be verified by seeing the results in the Results workspace. Please note that macros are redefined depending on their order of execution.

### use of Extract Macro in complex preprocessing

This Example Process is also part of the RapidMiner tutorial. It is included here to show the usage of the Extract Macro operator in complex preprocessing. This process will cover a number of concepts of macros including redefining macros, the macro of the Loop Values operator and the use of the Extract Macro operator. This process starts with a subprocess which is used to generate data. What is happening inside this subprocess is not relevant to the use of macros, so it is not discussed here. A breakpoint is inserted after this subprocess so that you can view the ExampleSet. You can see that the ExampleSet has 12 examples and 2 attributes: 'att1' and 'att2'. 'att1' is nominal and has 3 possible values: 'range1', 'range2' and 'range3'. 'att2' has real values.

The Loop Values operator is applied on the ExampleSet and iterates over the values of the specified attribute (i.e. att1) and applies the inner operators on the given ExampleSet while the

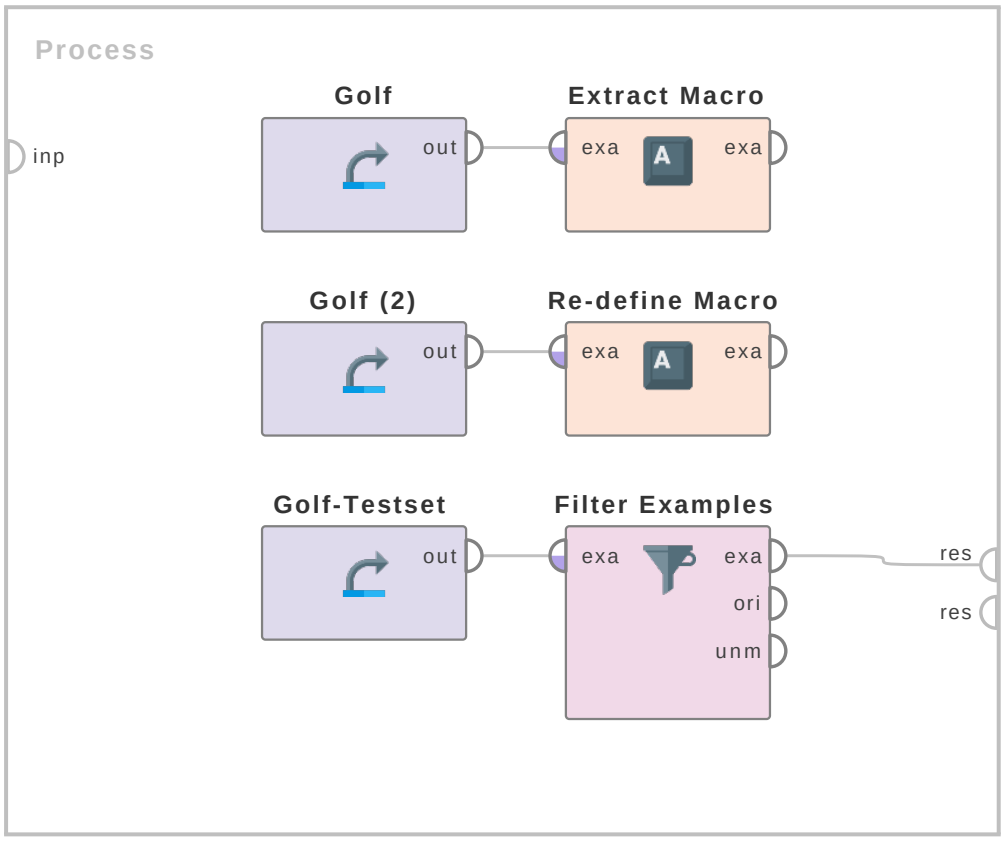



Figure 7.48: Tutorial process ‘Redefining a macro using the Extract Macro operator’.

current value can be accessed via the macro defined by the iteration macro parameter which is set to ‘loop_value’, thus the current value can be accessed by specifying %{loop_value} in the parameter values. As att1 has 3 possible values, Loop Values will iterate 3 times, once for each possible value of att1.

Here is an explanation of what happens inside the Loop Values operator. It is provided with an ExampleSet as input. The Filter Examples operator is applied on it. The condition class parameter is set to ‘attribute value filter’ and the parameter string is set to ‘att1 = %{loop_value}’. Note the use of the loop_value macro here. Only those examples are selected where the value of att1 is equal to the value of the loop_value macro. A breakpoint is inserted here so that you can view the selected examples. Then the Aggregation operator is applied on the selected examples. It is configured to take the average of the att2 values of the selected examples. This average value is stored in a new ExampleSet in the attribute named ‘average(att2)’. A breakpoint is inserted here so that you can see the average of the att2 values of the selected examples. The Extract Macro operator is applied on this new ExampleSet to store this average value in a macro named ‘current_average’. The originally selected examples are passed to the Generate Attributes operator that generates a new attribute named ‘att2_abs_avg’ which is defined by the expression ‘abs(att2 - %{current_average})’. Note the use of the current_average macro here. Its value is subtracted from all values of att2 and stored in a new attribute named ‘att2_abs_avg’. The Resultant ExampleSet is delivered at the output of the Loop Values operator. A breakpoint is inserted here so that you can see the ExampleSet with the ‘att2_abs_avg’ attribute. This output is fed to the Append operator in the main process. It merges the results of all the iterations into a single

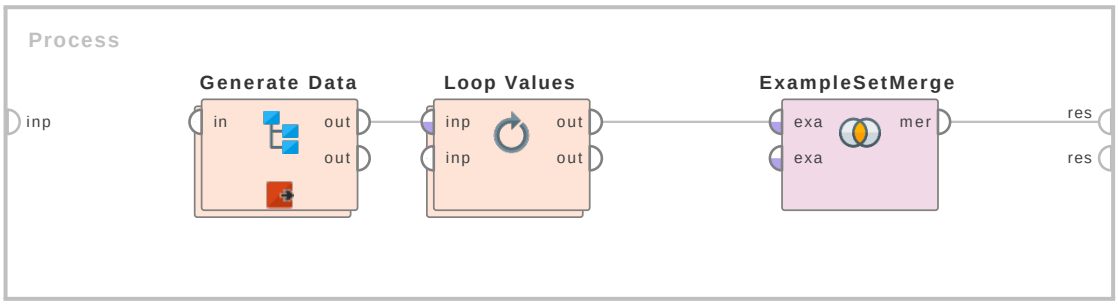



Figure 7.49: Tutorial process ‘use of Extract Macro in complex preprocessing’.

ExampleSet which is visible at the end of this process in the Results Workspace.

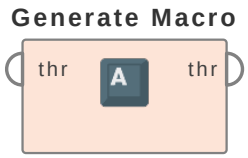
Here is what you see when you run the process.

ExampleSet generated by the Generate Data subprocess. Then the process enters the Loop Value operator and iterates 3 times.

Iteration 1: ExampleSet where the ‘att1’ value is equal to the current value of the loop_value macro i.e. ‘range1’Average of ‘att2’ values for the selected examples. The average is -1.161.ExampleSet with ‘att2_abs_avg’ attribute for iteration 1. Iteration 2: ExampleSet where the ‘att1’ value is equal to the current value of the loop_value macro i.e. ‘range2’Average of ‘att2’ values for the selected examples. The average is -1.656.ExampleSet with ‘att2_abs_avg’ attribute for iteration 2.

Iteration 3: ExampleSet where the ‘att1’ value is equal to the current value of the loop_value macro i.e. ‘range3’Average of ‘att2’ values for the selected examples. The average is 1.340.ExampleSet with ‘att2_abs_avg attribute’ for iteration 3. Now the process comes out of the Loop Values operator and the Append operator merges the final ExampleSets of all three iterations into a single ExampleSet that you can see in the Results Workspace.

# Generate Macro




This operator can be used to calculate new macros from existing macros. A macro can be used by writing *%{macro_name}* in parameter values of succeeding operators of the current process. A Macro can be considered as a value that can be used by all operators of the current process that come after the macro has been defined. This operator can also be used to re-define existing macros.

## Description

This operator can be used to define macros which can be used in parameter values of succeeding operators of the current process. Once the macro has been defined, the value of that macro can be used as parameter value in coming operators by writing the macro name in *%{macro_name}* format in the parameter value where 'macro_name' is the name of the macro specified when it was defined. In Generate Macro operator macro names and function descriptions are specified in the *function descriptions* parameters. The macro will be replaced in the value strings of parameters by the macro's value. This operator can also be used to re-define existing macros by specifying the name of that macro as name in the *function descriptions* parameter.

A large number of operations and functions is supported, which allows you to write rich expressions. For a list of operations and functions and their descriptions open the Edit Expression dialog. Complicated expressions can be created by using multiple operations and functions. Parenthesis can be used to nest operations. Since RapidMiner 6.0.3 the operator will fail if an expression is not valid so that you can correct it. The description of all operations follows this format:

This operator also supports various constants (for example 'INFINITY', 'PI' and 'e'). Again you can find a complete list in the Edit Expression dialog. You can also use strings in operations but the string values should be enclosed in double quotes (").

The functions available in the Generate Macro operator behave analogously to the functions of the Generate Attributes operator. Please study the Example Process of the Generate Attributes operator to understand the use of these functions.

### Macros

Macro can be considered as a value that can be used by all operators of the current process that come after the macro has been defined. Whenever using macros, make sure that the operators are in the correct sequence. It is compulsory that the macro should be defined before it can be used in parameter values. Macro is one of the advanced topics of RapidMiner, please study the attached Example Processes to develop a better understanding of macros.

There are also some predefined macros:

- *%{process_name}*: will be replaced by the name of the process (without path and extension)
- *%{process_file}*: will be replaced by the file name of the process (with extension)
- *%{process_path}*: will be replaced by the complete absolute path of the process file
- *%{execution_count}*: will be replaced by the number of times the current operator was applied.
- *%{operator_name}*: will be replaced by the name of the current operator.

Please note that other operators like many of the loop operators (e.g. Loop Values , Loop Attributes) also add specific macros.

## Input Ports

**through *(thr)*** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at first *through* input port of the Generate Macro operator is available at the first *through* output port.

## Output Ports

**through *(thr)*** The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the macro value is calculated even if this port is left without connections. The Generate Macro operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Generate Macro operator is delivered at the first *through* output port

## Parameters

**function descriptions** The list of macro names together with the expressions which define the new macros are specified through this parameter.

## Tutorial Processes

### Generating a new macro from an already existing macro

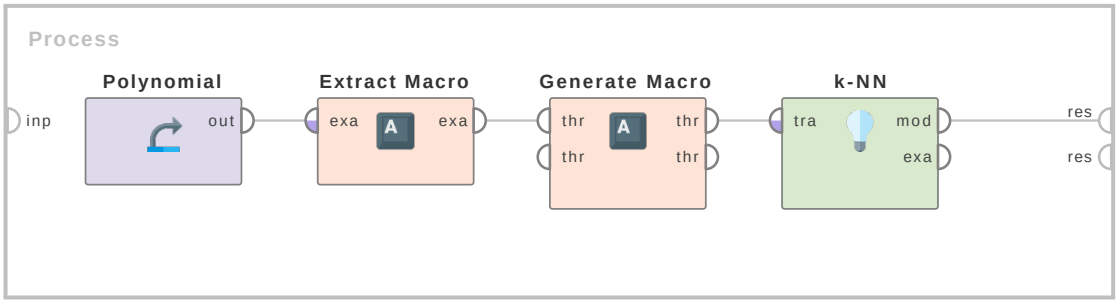


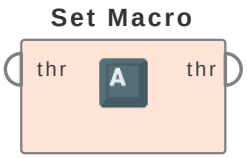

Figure 7.50: Tutorial process ‘Generating a new macro from an already existing macro’.

This Example Process discusses an imaginary scenario to explain how the Generate macro operator can be used to define a new macro from an existing macro. Suppose we want to apply the K-NN operator on an ExampleSet such that the value of the k parameter of the K-NN operator is set dynamically to some ratio of number of examples in the input ExampleSet. Let us assume that the required ratio is k = n * 0.025 where n is the number of examples in the input ExampleSet.

The Polynomial data set is loaded using the Retrieve operator. The Extract Macro operator is applied on it to create a new macro. The macro parameter is set to ‘example_count’ and the macro type parameter is set to ‘number of examples’. Thus a macro is created with the name

'example_count'. The value of this macro is equal to the number of examples in the Polynomial data set. Then the Generate Macro operator is applied. Only one macro is defined in the function descriptions parameter. The macro is named 'value_k' and it is defined by this expression: ceil(eval(%{example_count}) * 0.025). Note the use of the 'example_count' macro here. When this process is executed '%{example_count}' is replaced by the value of the 'example_count' macro which is equal to the number of examples in the input ExampleSet which is 200. Since all macros are strings we need to use the eval function to obtain 200 as number. Thus at run-time the expression that defines 'value_k' macro will be evaluated as ceil( 200 * 0.025 ). The result of this expression is 5. The K-NN operator is applied on the Polynomial data set with the parameter k set to '%{value_k}'. At run-time '%{value_k}' will be replaced by the actual value of the 'value_k' macro which is 5. Thus K-NN operator will be applied on the Polynomial data set with the parameter k set to 5. This can be verified by running the process and viewing the results in the Results Workspace.

# Set Macro




This operator can be used to define a macro which can be used by *%{macro_name}* in parameter values of succeeding operators of the current process. The macro value will NOT be derived from any ExampleSet. A macro can be considered as a value that can be used by all operators of the current process that come after the macro has been defined. This operator can also be used to re-define an existing macro.

## Description

This operator can be used to define a macro which can be used in parameter values of succeeding operators of the current process. Once the macro has been defined, the value of that macro can be used as parameter values in coming operators by writing the macro name in *%{macro_name}* format in the parameter value where 'macro_name' is the name of the macro specified when the macro was defined. In the Set Macro operator, the macro name is specified by the *macro* parameter and the macro value is specified by the *value* parameter. The macro will be replaced in the value strings of parameters by the macro's value. This operator can also be used to re-define an existing macro.

This operator sets the value of a macro irrespective of any ExampleSet. That is why this operator can also exist on its own i.e. without being connected to any other operator. If you want to create a single macro from properties of a given input ExampleSet, the Extract Macro operator is the right operator.

### Macros

A macro can be considered as a value that can be used by all operators of the current process that come after the macro has been defined. Whenever using macros, make sure that the operators are in the correct sequence. It is compulsory that the macro should be defined before it can be used in parameter values. The macro is one of the advanced topics of RapidMiner, please study the attached Example Process to develop a better understanding of macros. The Example Processes of the Extract Macro operator are also useful for understanding the concepts related to the macros.

There are also some predefined macros:

- *%{process_name}*: will be replaced by the name of the process (without path and extension)
- *%{process_file}*: will be replaced by the file name of the process (with extension)
- *%{process_path}*: will be replaced by the complete absolute path of the process file
- Several other short macros also exist, e.g. *%{a}* for the number of times the current operator was applied.

Please note that other operators like many of the loop operators (e.g. Loop Values , Loop Attributes) also add specific macros.

During the runtime the defined macros can be observed in the macro viewer.

## Differentiation

- **Set Macros** The Set Macros operator is like the Set Macro operator with only one difference. The Set Macros operator can be used for setting values of multiple macros whereas

the Set Macro operator can be used for setting value of just a single macro. See page 910 for details.

## Input Ports

**through *(thr)*** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the Set Macro operator is available at the first *through* output port.

## Output Ports

**through *(thr)*** Objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the macro value is set even if this port is left without connections. The Set Macro operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Set Macro operator is delivered at the first *through* output port

## Parameters

**macro** This parameter is used to specify the name of the macro. The macro can be accessed in succeeding operators of the current process by writing the macro's name in *%{macro_name}* format, where 'macro_name' is the name of the macro specified in this parameter.

**value** This parameter is used to specify the value of the macro. When the macro is accessed in succeeding operators of the current process by writing the macro's name in *%{macro_name}* format, it is replaced by the value of the macro specified by this parameter.

## Related Documents

- **Set Macros** (page 910)

## Tutorial Processes

### Introduction to the Set Macro operator

This is a very basic process that demonstrates the use of the Set Macro operator. The Set Macro operator is used first of all. One macro is defined using the macro and the value parameter. The macro is named 'number' and it is given the value 1. Note that this operator is not connected to any other operator; it can exist at its own. Always make sure that the macro is defined before it is used in the parameter values.

The 'Golf' data set is loaded using the Retrieve operator. The Select Subprocess operator is applied on it. Double-click on the Select Subprocess operator to see the subprocesses in it. As you can see, there are four subprocesses:

Subprocess 1: The k-NN operator is applied on the input and the resulting model is passed to the output. Subprocess 2: The Naive Bayes operator is applied on the input and the resulting model is passed to the output. Subprocess 3: The Decision Tree operator is applied on the input

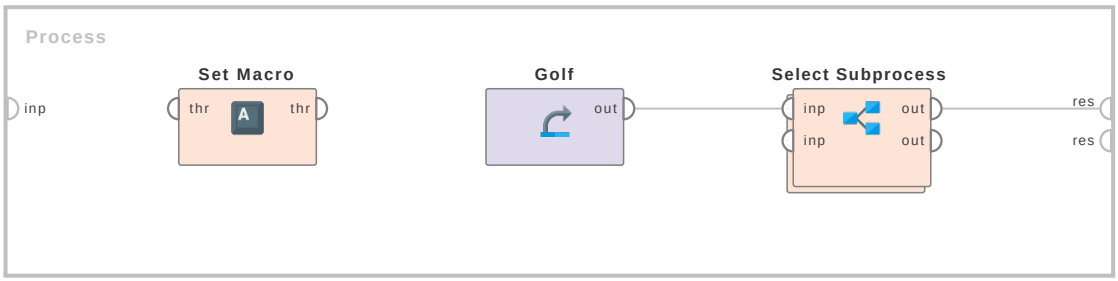
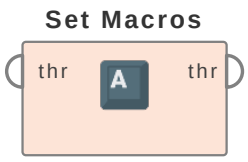



Figure 7.51: Tutorial process 'Introduction to the Set Macro operator'.

and the resulting model is passed to the output. Subprocess 4: The input is directly connected to the output.

Only one of these subprocesses can be executed at a time. The subprocess to be executed can be controlled by the select which parameter of the Select Subprocess operator. The select which parameter is set using the 'number' macro defined by the Set Macro operator. The select which parameter is set to '%{number}'. When the process will be executed, '%{number}' will be replaced with the value of the 'number' macro i.e. '%{number}' will be replaced by 1. Thus the select which parameter is set to 1, thus the first subprocess will be executed. When you run the process you will see the model created by the k-NN operator in the Results workspace. As the value of the select which parameter is provided by the macro created by the Set Macro operator, changing the value of the macro will change the value of the select which parameter. To execute the second subprocess set the value parameter of the Set Macro operator to 2 and run the process again. You will see the model generated by the Naive Bayes operator in the Results Workspace. To execute the third subprocess set the value parameter of the Set Macro operator to 3 and run the process again. You will see the model generated by the Decision Tree operator in the Results Workspace. To execute the fourth subprocess set the value parameter of the Set Macro operator to 4 and run the process again. Now you will see the 'Golf' data set in the Results Workspace because no operator was applied in the fourth subprocess.

# Set Macros




This operator can be used to define multiple macros which can be used by *%{macro_name}* in parameter values of succeeding operators of the current process. The macro values will NOT be derived from any ExampleSet. A macro can be considered as a value that can be used by all operators of the current process that come after the macro has been defined. This operator can also be used to re-define existing macros.

## Description

This operator can be used to define multiple macros which can be used in parameter values of succeeding operators of the current process. Once the macro has been defined, the value of that macro can be used as parameter values in coming operators by writing the macro name in *%{macro_name}* format in a parameter value where 'macro_name' is the name of the macro specified when the macro was defined. In the Set Macros operator, the macro name and value is specified by the *macros* parameter. A macro will be replaced in the value strings of parameters by the macro's value. This operator can also be used to re-define existing macros.

This operator sets the value of multiple macros irrespective of any ExampleSet. That is why this operator can also exist on its own i.e. without being connected to any other operator. If you want to create a single macro from properties of a given input ExampleSet, the Extract Macro operator is the right operator.

### Macros

A macro can be considered as a value that can be used by all operators of the current process that come after the macro has been defined. Whenever using macros, make sure that the operators are in the correct sequence. It is compulsory that the macro should be defined before it can be used in parameter values. The macro is one of the advanced topics of RapidMiner, please study the attached Example Process to develop a better understanding of macros. The Example Processes of the Extract Macro operator are also useful for understanding the concepts related to the macros.

There are also some predefined macros:

- *%{process_name}*: will be replaced by the name of the process (without path and extension)
- *%{process_file}*: will be replaced by the file name of the process (with extension)
- *%{process_path}*: will be replaced by the complete absolute path of the process file
- Several other short macros also exist, e.g. *%{a}* for the number of times the current operator was applied.

Please note that other operators like many of the loop operators (e.g. Loop Values , Loop Attributes) also add specific macros.

## Input Ports

**through *(thr)*** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the

same. The object supplied at the first *through* input port of the Set Macros operator is available at the first *through* output port.

## Output Ports

**through *(thr)*** Objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the macro value is set even if this port is left without connections. The Set Macros operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Set Macros operator is delivered at the first *through* output port

## Parameters

**macros** This parameter is used to specify the names and values of the macros. Macros can be accessed in succeeding operators of the current process by writing the macro's name in *%{macro_name}* format, where 'macro_name' is the name of the macro specified in this parameter.

## Tutorial Processes

### Introduction to the Set Macros operator

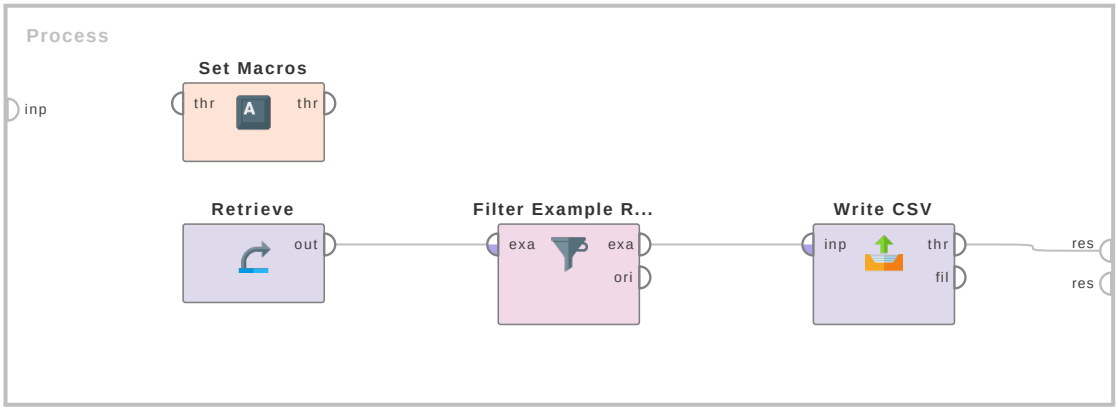



Figure 7.52: Tutorial process 'Introduction to the Set Macros operator'.

This is a very basic process that demonstrates the use of macros and the Set Macros operator. The Set Macros operator is used first of all. Two macros are defined in the macros parameter. They are named 'min' and 'max'. 'min' is given the value 1 and 'max' is given the value 4. Note that this operator is not connected to any other operator; it can exist at its own. Always make sure that the macro is defined before it is used in parameter values.

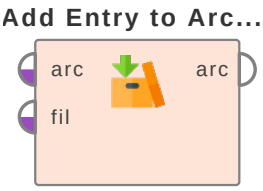
The 'Golf' data set is loaded using the Retrieve operator. The Filter Example Range operator is applied on it. The first example parameter is set to '%{min}' and the last example parameter is set to '%{max}'. When the process will be executed, '%{min}' and '%{max}' will be replaced with the values of the respective macros i.e. '%{min}' and '%{max}' will be replaced by 1 and 4

respectively. Thus the Filter Examples Range operator will deliver only the first four examples of the 'Golf' data set.

At the end, the Write CSV operator is applied to store the output of the Filter Example Range operator in a CSV file. Note that the csv file parameter is set to 'D:\%{process_name}'. Here '%{process_name}' is a predefined macro which will be replaced by the name of the current process (without file extension). Thus the output of the Filter Example Range operator will be written in a csv file in the D drive of your computer. The name of the file will be the same as the name of this process.

# 7.4 Files

## Add Entry to Archive File




This operator adds entries to an archive file object, currently the only available type is a zip file.

### Description

The Add Entry to Archive File operator adds entries, i.e. files, to an archive file object created by the Create Archive File operator. By default, the entries are added to the root directory of the archive file, but you can specify a directory name to create a subdirectory inside the archive file. Please have a look at the tutorial process of this operator to better understand its usage.

### Input Ports

**archive file *(arc)*** This operator can have multiple inputs. When one *input* port is connected, another *input* port becomes available which is ready to accept another input (if any).This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. Output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along with data.

**file input *(fil)*** The Add Entry to Archive File operator can have multiple inputs. When one *input* port is connected, another *input* port becomes available which is ready to accept another input (if any). This input port expects a File Object. File Objects can be created e.g. with the Open File operator.

### Output Ports

**archive file *(arc)*** The same archive file object that has been connected to the input port is output of this port, with the additional entries added by this operator.

### Parameters

**directory *(string)*** This parameter specifies the directory where the entry will be stored inside the archive file. Specify it in the form 'my/sub/directory', or leave it empty to store the entry in the root folder.

**override compression level *(boolean)*** This parameter allows to override the default compression level of the archive file object for the entries created by this operator. The default level is set by the Create Archive File operator that created the archive file object. This is useful, if you are adding pre-compressed files to the archive, such as zip files, jar files etc. These files cannot be further compressed, so you can save some execution time by setting the compression level for new entries of this kind to a low value.

**compression level** ***(integer)*** The compression level of the newly created entries is specified through this parameter. In general, higher compression levels result also in a higher runtime.

## Tutorial Processes

### Creating and storing a zip file

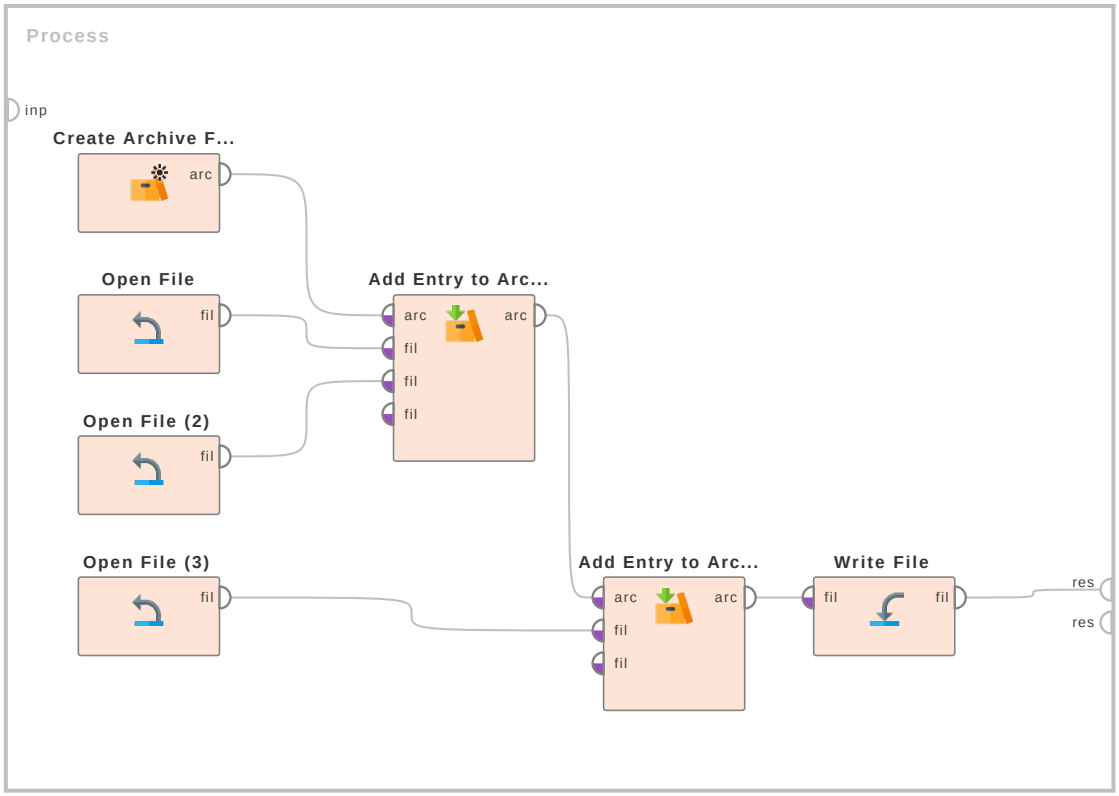



Figure 7.53: Tutorial process 'Creating and storing a zip file'.

This Example Process demonstrates how a zip file can be created in RapidMiner, how entries can be added and how the file can be written to a disk.

First of all, the zip file is created with the Create Archive File operator. Then, some entries are added. At first the Open File operators open some files from your harddisk. These files are then added to the zip file via the Add Entry to Archive File operators. You can see that you can add several files in one single step, and that you can also concatenate several Add Entry to Archive File operators. Finally, the zip file is written to a disk with the Write File operator.

Please be sure to select some valid files from your harddisk in the Open File operators, and to specify a valid location in the Write File operator!

The second Add Entry to Archive File operator creates a directory inside the zip file. After the execution of the process you may open the archive file from your disk and inspect the results.

### Storing freshly created data in a zip file

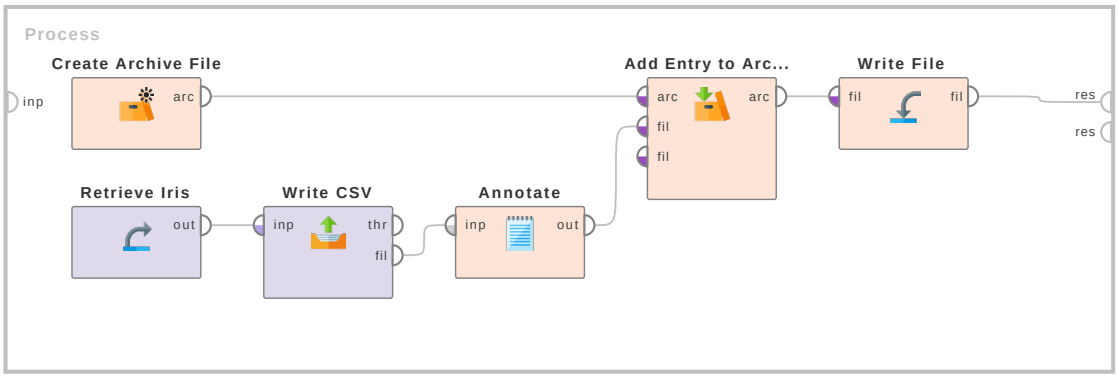


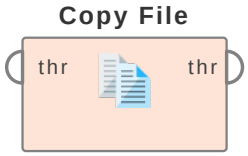

Figure 7.54: Tutorial process 'Storing freshly created data in a zip file'.

This process loads the Iris data set, creates a CSV file from it and stores it in a zip file. The zip file is then saved to disk with the Write File operator.Please note that you have to set the filename parameter of the Write File operator.

When you load a file from disk with Open File, the filename is known to RapidMiner. In the current process, that is not the case, since the CSV file has been created on the fly: we must assign the name manually. This is done by defining the Filename annotation with the help of the Annotate operator.

For more information about annotations and the related operators, please have a look at the documentation of the Annotate operator.

# Copy File




This operator copies the chosen file to the specified destination.

## Description

The Copy File operator copies the file specified in its parameters to the indicated destination. If the inserted path does not already exist the needed folders and the file are created. It is also possible to overwrite an existing document.

## Input Ports

**through *(thr)*** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the Copy File operator is available at the first *through* output port.

## Output Ports

**through *(thr)*** The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the file is copied even if this port is left without connections. The Copy File operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Copy File operator is delivered at the first *through* output port.

## Parameters

**source file *(file)*** The file that should be copied is specified through this parameter.

**new file *(file)*** The copied file is saved as the file and at the target specified through this parameter.

**overwrite *(boolean)*** If this parameter is set to true a file specified in the *new file* parameter that already existed is replaced by the file specified in the *source file* parameter.

## Tutorial Processes

### Writing the Labor-Negotiations data set into an Excel file and copying it

This Example Process shows how the Copy File operator can be used to copy a specified file. For this Example Process you first need a file to copy. An Excel file is created by loading the 'Labor-Negotiations' data set with the Retrieve operator and writing it into an Excel file with the Write

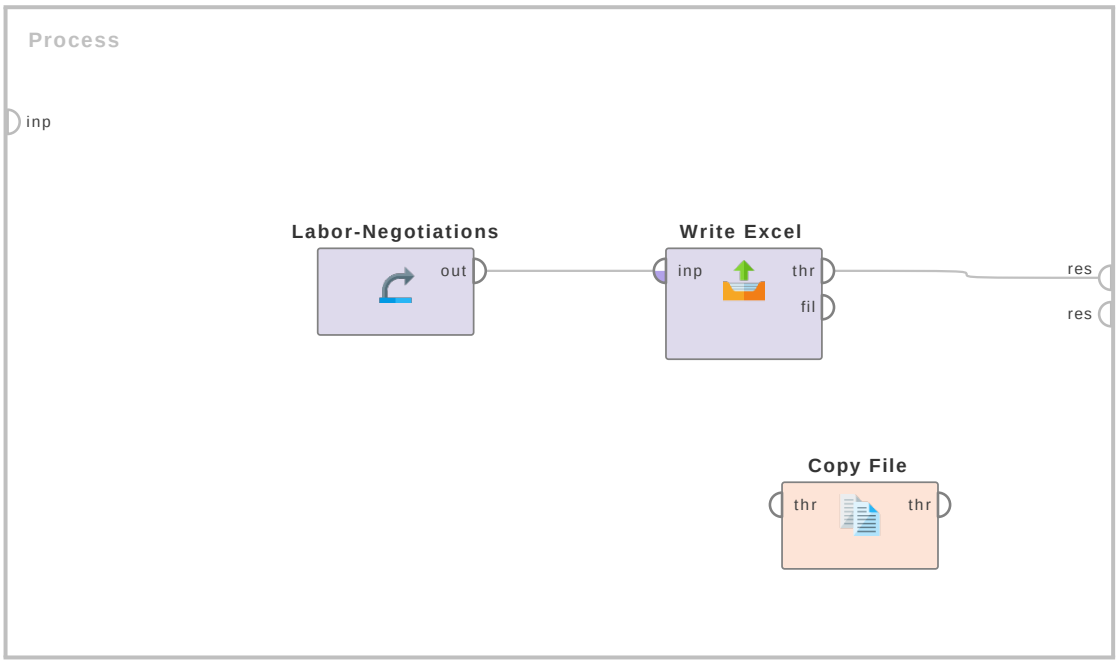



Figure 7.55: Tutorial process ‘Writing the Labor-Negotiations data set into an Excel file and copying it’.

Excel operator. The file is saved as ‘D:\Labor data set.xls’ if this file does not already exist. For further understanding of this operator please read the description of the Write Excel operator.

The Copy File operator is inserted without any connections. The source file parameter is set to ‘D:\Labor data set.xls’ and the new file parameter to ‘D:\test\Labor data set.xls’. Run the process and two files are created in your D drive, the ‘Labor data set.xsl’ of the Write Excel operator and the ‘Labor data set.xsl’ in a separate folder named ‘test’.

# Create Archive File


This operator creates an archive file object, which allows the compression of other file objects. It is only possible to create zip files at the moment. After all entries have been added, the archive file object can be stored in the filesystem.

## Description

The Create Archive File operator creates an archive file object. This object can be passed to the Add Entry to Archive File operator to add data. After all entries have been added, the archive file object can be stored on your harddisk with the Write File operator, or you can store it in the repository.

Currently this operator can only create zip files, but more archive types may be added in a later version.

Please have a look at the tutorial process to better understand the usage of this operator.

## Output Ports

**archive file *(arc)*** The archive file object generated during the execution of this operator is the output of this port.

## Parameters

**buffer type *(selection)*** This operator defines where the buffer for the archive file will be created. There are two possibilities:

- **file** The archive file will be created on a disk. Choose this option if you plan to create a big archive file.
- **memory** The archive file will be cached in the memory. A memory buffered archive file will usually perform faster in terms of execution time, but the complete archive must be kept in memory, which can lead to problems if large files or a large amount of files is added to the archive. Choose this option if you create rather small files or have a lot of memory.

**use default compression level *(boolean)*** This parameter allows you to override the default compression level. The default compression level depends on the host machine, but usually offers a reasonable trade-off between execution time and compression factor.

**compression level *(integer)*** The default compression level of the created zip file is specified by this parameter. This level may be overridden in the subsequent Add Entry to Archive File operators on a per-entry base. In general, higher compression levels result also in a higher runtime.

## Tutorial Processes

### Creating and storing a zip file

This Example Process demonstrates how a zip file can be created in RapidMiner, how entries can be added and how the file can be written to a disk.

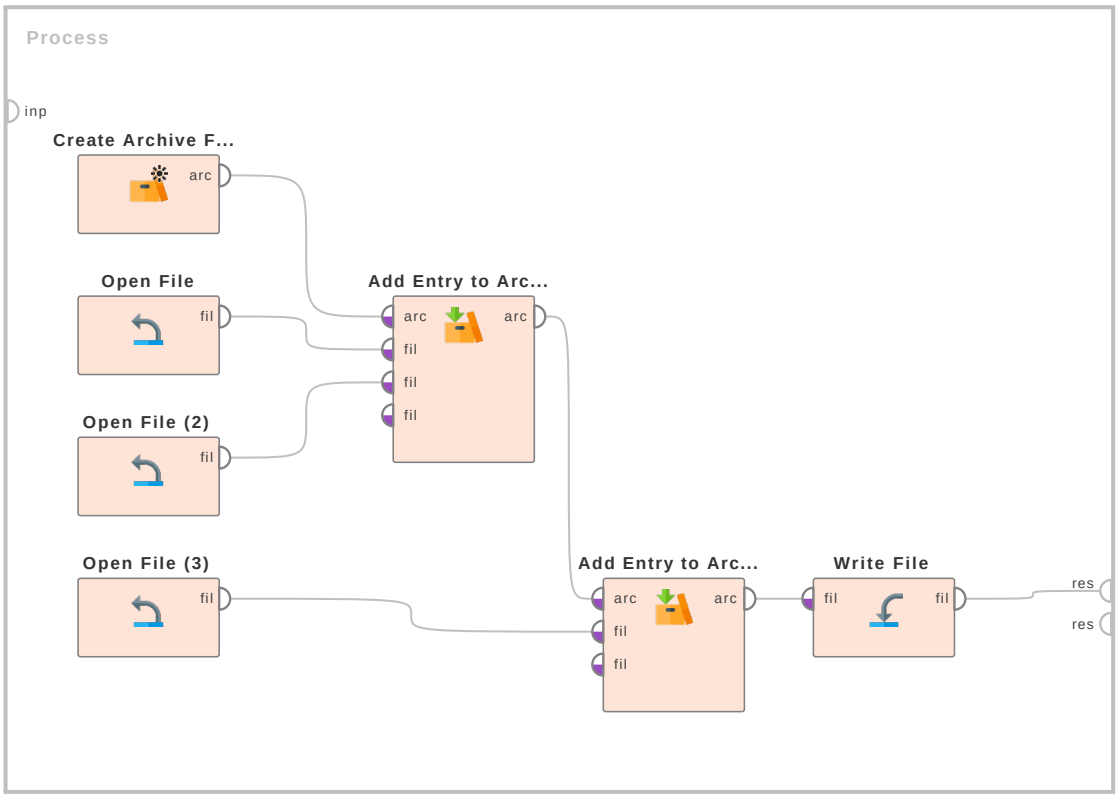


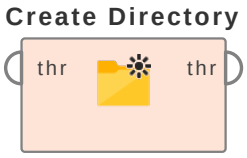

Figure 7.56: Tutorial process ‘Creating and storing a zip file’.

First of all, the zip file is created with the Create Archive File operator. Then, some entries are added. At first the Open File operators open some files from your harddisk. These files are then added to the zip file via the Add Entry to Archive File operators. You can see that you can add several files in one single step, and that you can also concatenate several Add Entry to Archive File operators. Finally, the zip file is written to a disk with the Write File operator.

Please be sure to select some valid files from your harddisk in the Open File operators, and to specify a valid location in the Write File operator!

The second Add Entry to Archive File operator creates a directory inside the zip file. After the execution of the process you may open the archive file from your disk and inspect the results.

# Create Directory




This operator creates a directory at the specified location.

## Description

The Create Directory operator creates a directory at the chosen location in the file system, if the folder does not already exist. If the inserted path does not exist the needed folders are created.

## Input Ports

**through *(thr)*** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the Create Directory operator is available at the first *through* output port.

## Output Ports

**through *(thr)*** The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the directory is created even if this port is left without connections. The Create Directory operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Create Directory operator is delivered at the first *through* output port

## Parameters

**location *(file)*** The location where the new directory is build is specified through this parameter.

**name *(string)*** The name of the new directory is specified through this parameter.

## Tutorial Processes

### Creating a new directory in the D drive

The Create directory operator is inserted in the process. The location parameter is set to 'D:\new' and the name parameter to 'folder'. Run the process and a folder named 'folder' is created in your D drive inside the newly created folder 'new'.


Figure 7.57: Tutorial process 'Creating a new directory in the D drive'.

## Delete File


This operator deletes a file at the specified location.

### Description

The Delete File operator deletes the selected file if possible otherwise an error message is shown. You can also specify that an error message should be shown if the file that should be deleted does not exist.

### Input Ports

**through *(thr)*** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the Delete File operator is available at the first *through* output port.

## Output Ports

**through *(thr)*** The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the file is deleted even if this port is left without connections. The Delete File operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Delete File operator is delivered at the first *through* output port

## Parameters

**file *(file)*** The file that should be deleted is specified through this parameter.

**fail if missing *(boolean)*** Determines whether an exception should be generated if the file is missing, e. g. because it already got deleted in the last run. If set to false nothing happens if this error occurs.

## Tutorial Processes

### Writing the Labor-Negotiations data set into an Excel file and copying it

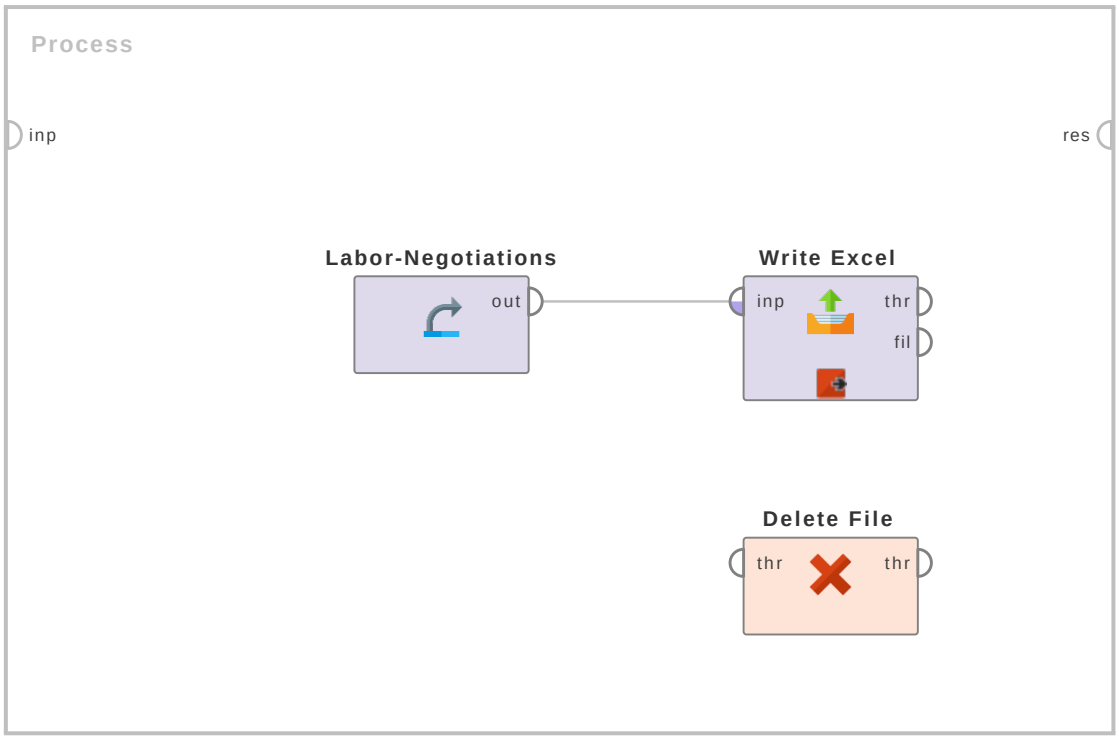
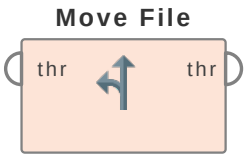



Figure 7.58: Tutorial process ‘Writing the Labor-Negotiations data set into an Excel file and copying it’.

This Example Process shows how the Delete File operator can be used to delete a specified file. For this Example Process you first need a file to delete. An Excel file is created by loading the 'Labor-Negotiations' data set with the Retrieve operator and writing it into an Excel file with the Write Excel operator. The file is saved as 'D:\Labor data set.xls' if this file does not already exist. A breakpoint is inserted so you can have a look at the Excel file in your D drive. For further understanding of this operator please read the description of the Write Excel operator.

The Delete File operator is inserted without any connections. The file parameter is set to 'D:\Labor data set.xls' and the fail if missing parameter to true. Run the process and the file in your D drive will be deleted. If you delete the Retrieve and the Write Excel operator and run the process again an error message will be shown telling you that the file could not be deleted as it does not exist.

# Move File




This operator moves the chosen file to the specified destination.

## Description

The Move File operator moves the selected file from its original directory to the chosen location. If the inserted path does not already exist the needed folders and the file are created. It is also possible to overwrite an existing document.

## Input Ports

**through *(thr)*** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the Move File operator is available at the first *through* output port.

## Output Ports

**through *(thr)*** The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the file is moved even if this port is left without connections. The Move File operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Move File operator is delivered at the first *through* output port

## Parameters

**file *(file)*** The file that should be moved.

**destination *(file)*** The new location of the file.

**overwrite *(boolean)*** Determines whether an already existing file should be overwritten.

## Tutorial Processes

### Writing the Labor-Negotiations data set into an Excel file and moving it

This Example Process shows how the Move File operator can be used to move a specified file. For this Example Process you first need a file to move. An Excel file is created by loading the 'Labor-Negotiations' data set with the Retrieve operator and writing it into an Excel file with the Write Excel operator. The file is saved as 'D:\Labor data set.xls' if this file does not already exist. A breakpoint is inserted so you can have a look at the Excel file in your D drive. For further understanding of this operator please read the description of the Write Excel operator.

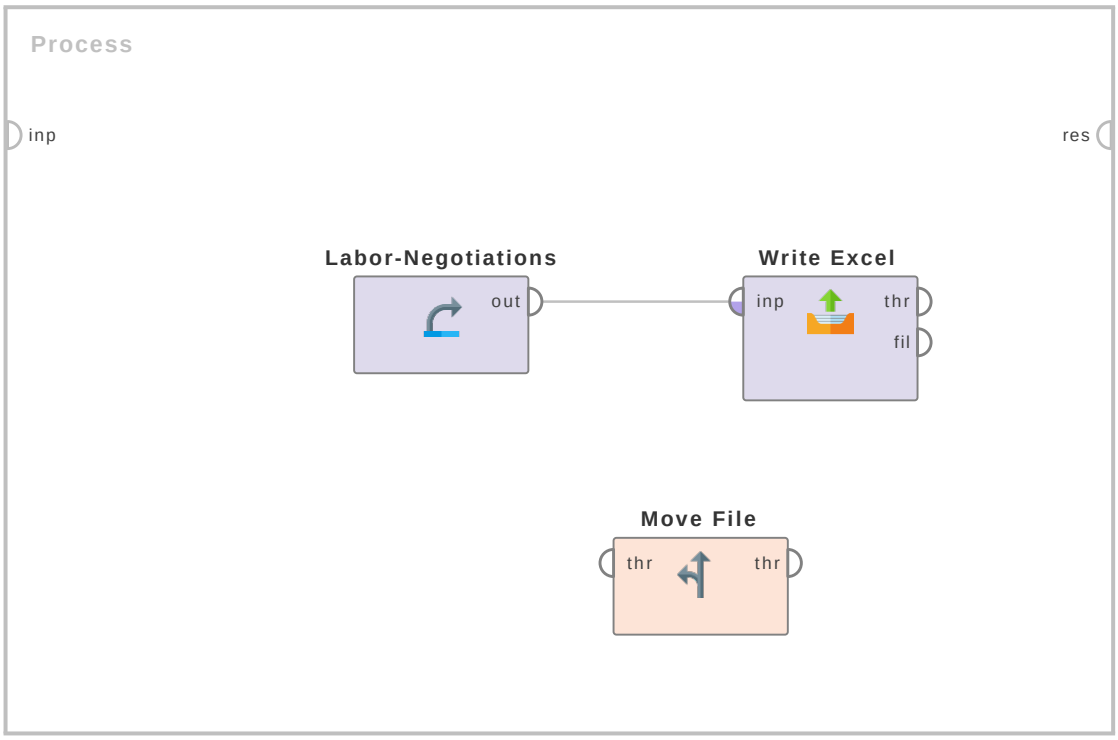
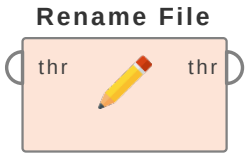



Figure 7.59: Tutorial process 'Writing the Labor-Negotiations data set into an Excel file and moving it'.

The Move File operator is inserted without any connections. The file parameter is set to 'D:\Labor data set.xls' and the destination parameter to 'C:\Data\Labor data set.xls'. Run the process and the file in your D drive will be moved to a newly created folder named 'Data' in your C drive.

# Rename File




This operator renames a file or a folder.

## Description

The Rename File operator allocates a new name to a selected file or folder. Please ensure that the new name of your file has the right ending, e.g. '.xls' as in our Example Process.

## Input Ports

**through *(thr)*** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the Rename File operator is available at the first *through* output port.

## Output Ports

**through *(thr)*** The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the file is renamed even if this port is left without connections. The Rename File operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Rename File operator is delivered at the first *through* output port

## Parameters

**file *(file)*** The file or folder that should be renamed is specified through this parameter.

**new name *(string)*** The new name of the file or folder is specified through this parameter.

## Tutorial Processes

### Writing the Labor-Negotiations data set into an Excel file and renaming it

This Example Process shows how the Rename File operator can be used to rename a specified file. For this Example Process you first need a file to rename. An Excel file is created by loading the 'Labor-Negotiations' data set with the Retrieve operator and writing it into an Excel file with the Write Excel operator. The file is saved as 'D:\Labor data set.xls' if this file does not already exist. A breakpoint is inserted so you can have a look at the Excel file in your D drive. For further understanding of this operator please read the description of the Write Excel operator.

The Rename File operator is inserted without any connections. The file parameter is set to 'D:\Labor data set.xls' and the new name parameter to 'labor_data_set.xls'. Run the process and

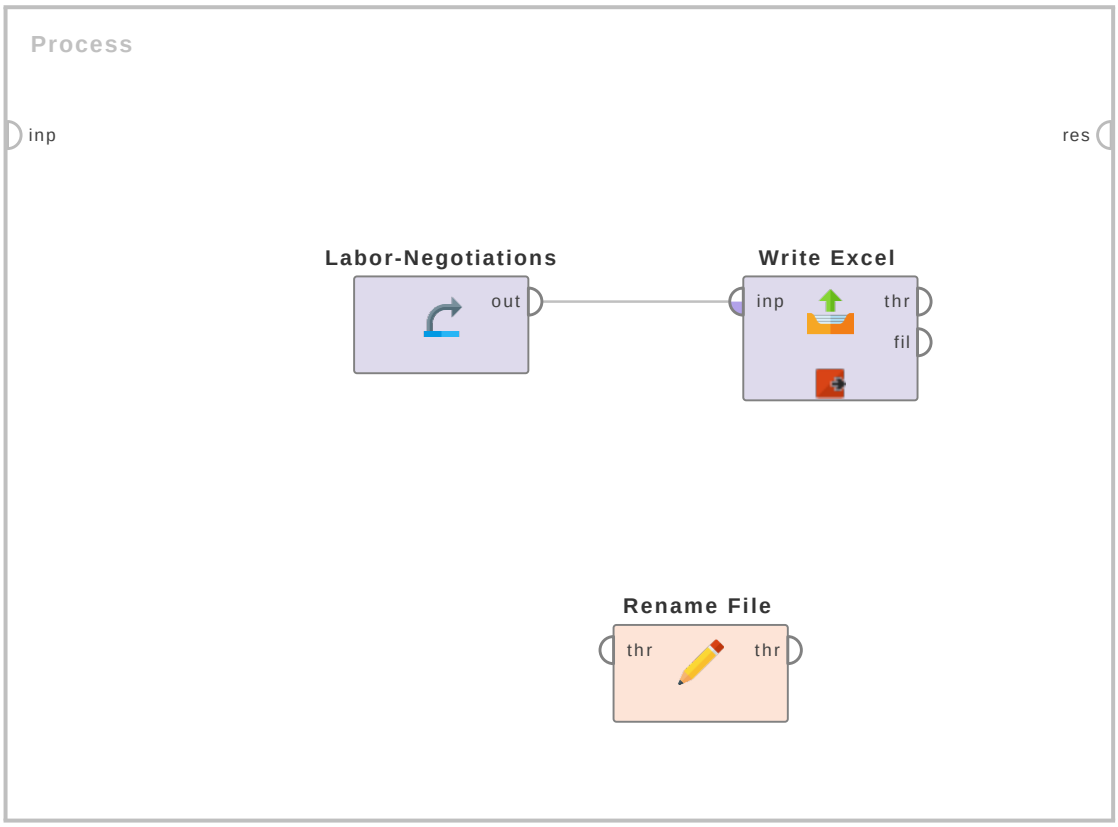
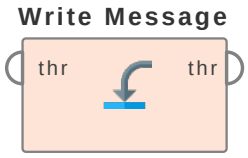



Figure 7.60: Tutorial process 'Writing the Labor-Negotiations data set into an Excel file and renaming it'.

the file in your D drive which was formerly named as 'Labor data set.xls' will now be named as labor_data_set.xls'.

# Write Message




This operator simply writes the given text into the specified file (can be useful in combination with a process branch).

## Description

This operator simply writed the specified text into the specified file. This can be useful in combination with the *ProcessBranch* operator. For example, one could write the success or non-success of a process into the same file depending on the condition specified by a process branch.

## Input Ports

**input *(inp)*** Any results connected at this port are written to the specified file and then delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The result supplied at the first input port of the Write as Text operator is available at its first output port.

## Output Ports

**input *(inp)*** The results that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the results are written into the file even if this port is left without connections. The Write as Text operator can have multiple outputs. When one output is connected, another output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The result connected at the first input port of the Write as Text operator is delivered through the first output port

## Parameters

**file *(filename)*** The path of the text file is specified here. Will be created if it does not exist.

**text** The text which should be written into the file.

**mode *(selection)*** This parameters allows you to control what should happen to existing files. It has the following options:

- **replace** Replace any existing file content with the given text.
- **append** Append the text to the end of the file.

**encoding** The encoding used for reading or writing files.

## Tutorial Processes

### Use Write Message as a custom logger.

In this example we use the append mode of the Write Message operator to create a simple logging mechanism. The Set Macros operator is used to store the logfile path and the process name

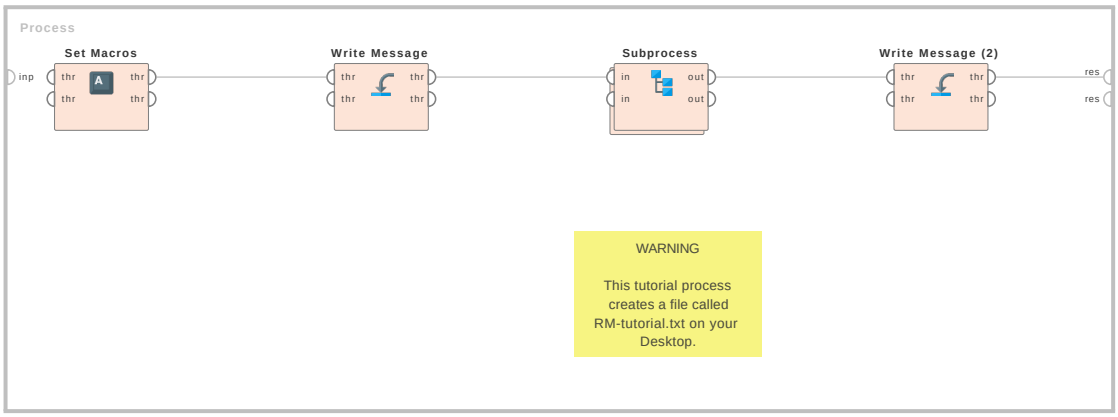
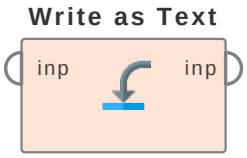



Figure 7.61: Tutorial process 'Use Write Message as a custom logger.'.

information. The Write Message operators are logging the process start and finish. In between we are delaying to get different time stamps.

# Write as Text




This operator writes the given results to the specified file. This operator can be used at each point in an operator chain to write results at every step of the process into a file.

## Description

The Write as Text operator writes the given results to the specified file. The file is specified through the *result file* parameter. This operator does not modify the results; it just writes them to the file and then delivers the unchanged results through its output ports. Every input object which implements the *ResultObject* interface (which is the case for almost all objects generated by the core RapidMiner operators) will write its results to the file specified by the *result file* parameter. If the *result file* parameter is not set then the global result file parameter with the same name of the *ProcessRootOperator* (the root of the process) will be used. If this file is also not specified then the results are simply written to the console (standard out).

## Input Ports

**input *(inp)*** Any results connected at this port are written to the specified file and then delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The result supplied at the first input port of the Write as Text operator is available at its first output port.

## Output Ports

**input *(inp)*** The results that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the results are written into the file even if this port is left without connections. The Write as Text operator can have multiple outputs. When one output is connected, another output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The result connected at the first input port of the Write as Text operator is delivered through the first output port

## Parameters

**result file *(filename)*** The results are written into the file specified through this parameter.

**encoding *(selection)*** This is an expert parameter. There are different options, users can choose any of them

## Tutorial Processes

### Writing multiple results into a file

The 'Golf' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it with default values of all parameters. The Default Model operator trains a model on the 'Golf' data set in the training subprocess. The trained model is applied on the testing data set

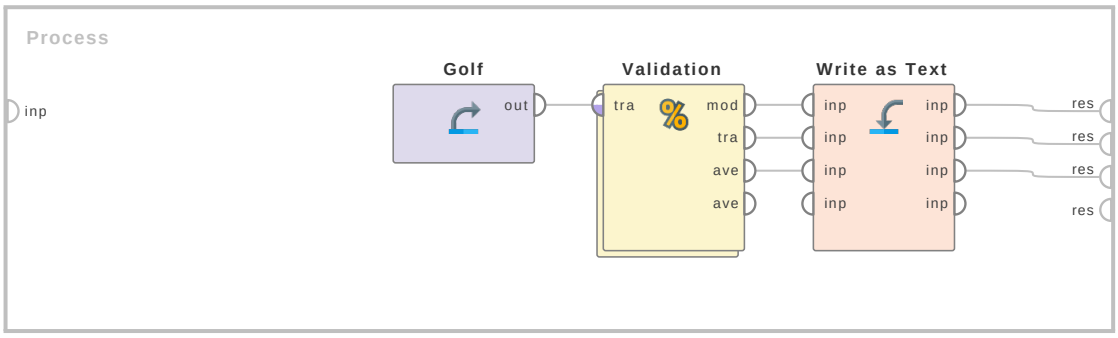
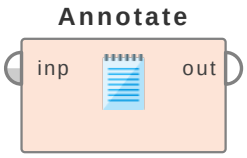



Figure 7.62: Tutorial process 'Writing multiple results into a file'.

in the testing subprocess by the Apply Model operator. The Performance operator measures the performance of the model. The Split Validation operator delivers multiple results i.e. a trained model, the input ExampleSet and the performance vector. All these results are connected to the Write as Text operator to write them into a file. The result file parameter is set to 'D:\results.txt', thus a text file named 'results' is created in the D drive of the computer. If the file already exists then the results are appended to the file. The results are written into the specified file but they are not as detailed as the results in the Results Workspace.

# 7.5 Annotations

## Annotate




Adds annotations to an IOObject or changes existing ones.

### Description

Sometimes it is necessary to attach information to a data object which is not part of the data itself. That could be e.g. the source of a file or example set, information on when or how the data has been acquired etc. In RapidMiner, information of this kind can be attached as so-called annotations to any kind of IOObject.

Annotations are key/value pairs: values can be referenced by or assigned to unique keys.

### Input Ports

**input *(inp)*** Any type of object can be connected to this port. The annotations will be added to this object.

### Output Ports

**output *(out)*** The same object as passed to the input port, with updated annotations.

### Parameters

**annotations *(menu)*** Defines the pairs of annotation names and annotation values. Click the button, select or type an annotation name into the left input field and enter its value into the right field. You can specify an arbitrary amount of annotations here. Please note that it is not possible to create empty annotations.

**duplicate annotations *(selection)*** This parameter indicates what should happen if duplicate annotation names are specified.

- **overwrite** If this option is selected, the values existing annotations will be simply overwritten.
- **ignore** If this option is selected, duplicate annotations are ignored and the value of the original annotation is kept.
- **error** If this option is selected, an error is displayed and the process stops if duplicate annotation names are found.

### Related Documents

- **Annotations to Data** (page **??**)
- **Data to Annotations** (page **??**)
- **Extract Macro from Annotation** (page **??**)

## Tutorial Processes

### Annotating a data set

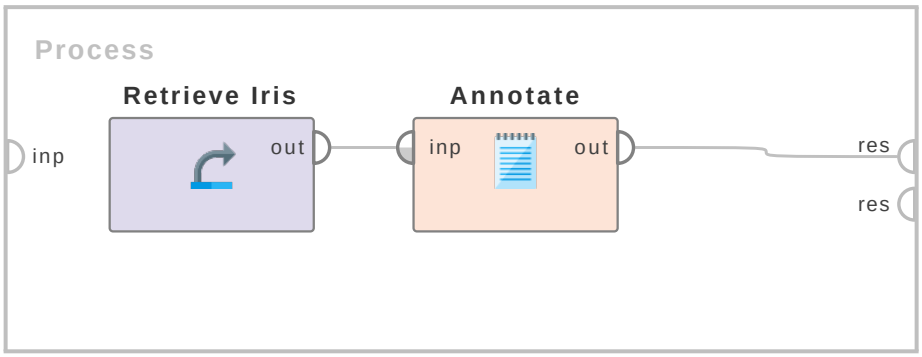
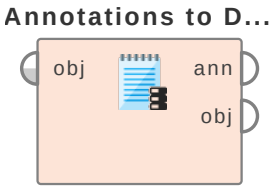



Figure 7.63: Tutorial process 'Annotating a data set'.

The Iris data set is loaded and annotated: the Source annotation is overwritten, and a new annotation is created.

To see the annotations of an object go to the results view and select the Annotations display option.

# Annotations to Data




Extracts all annotations from an IOObject and creates an example set from them.

## Description

The resulting data set will contain two columns: *annotation* contains the annotation names and is unique. *value* contains the respective annotation values.

For a general introduction of annotations please study the Annotate operator.

## Input Ports

**object *(obj)*** Any type of object can be connected to this port. The annotations will be extracted from this object.

## Output Ports

**object through *(obj)*** The same object as passed to the input port, unchanged.

**annotations *(ann)*** The annotations extracted from the input object. Will be empty if the object does not contain any annotations.

## Related Documents

- **Annotate** (page ??)
- **Data to Annotations** (page ??)

## Tutorial Processes

### Extracting annotations into a data set

The process loads the Iris data set and extracts its annotations into a new example set. The only annotation will be the Source annotation which is created by the Retrieve operator.

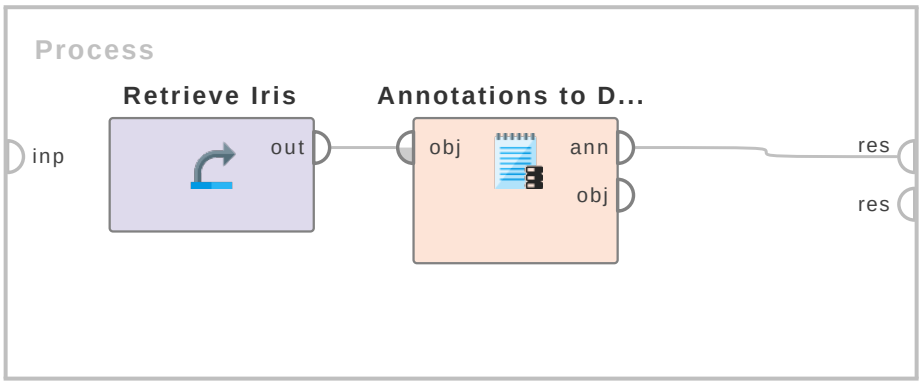
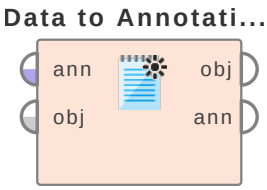



Figure 7.64: Tutorial process 'Extracting annotations into a data set'.

## Data to Annotations




Adds annotations to an object that are extracted from an example set.

### Description

The resulting data set will contain two columns: *annotation* contains the annotation names and is unique. *value* contains the respective annotation values.

For a general introduction to annotations please have a look at the help of the Annotate operator.

### Input Ports

**object *(obj)*** Any type of object can be connected to this port. The annotations will be extracted from this object.

### Output Ports

**object through *(obj)*** The same object as passed to the input port, unchanged.

**annotations *(ann)*** The annotations extracted from the input object. Will be empty if the object does not contain any annotations.

### Parameters

**key attribute *(string)*** The attribute which contains the names of the annotations to be created. Should be unique.

**value attribute *(string)*** The attribute which contains the values of the annotations to be created. If a value is missing, the respective annotation will be removed.

**duplicate annotations** ***(selection)*** Indicates what should happen if duplicate annotation names are specified.

- **overwrite** If this option is selected, the values existing annotations will be simply overwritten.
- **ignore** If this option is selected, duplicate annotations are ignored and the value of the original annotation is kept.
- **error** If this option is selected, an error is displayed and the process stops if duplicate annotation names are found.

## Related Documents

- **Annotate** (page ??)
- **Annotations to Data** (page ??)

## Tutorial Processes

### Annotating an object from a data set

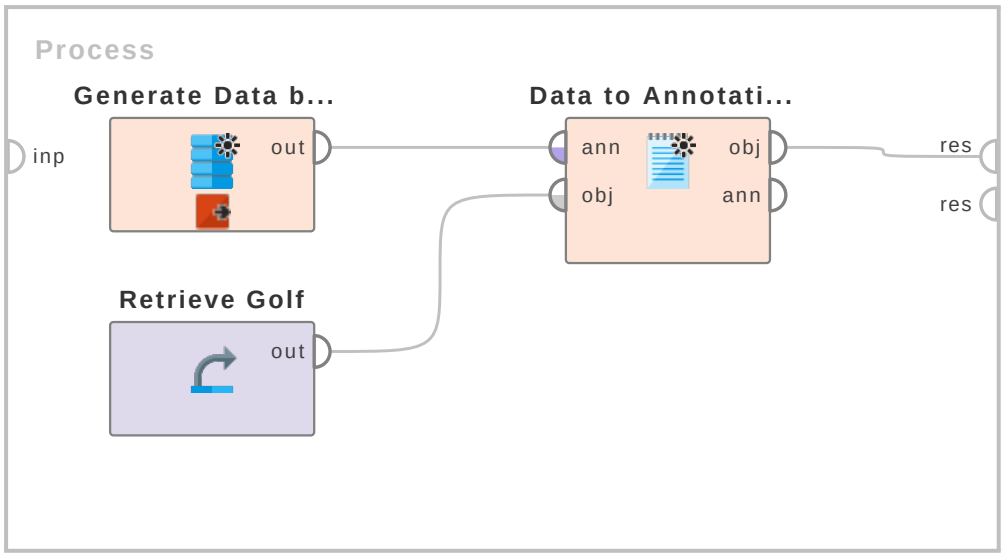
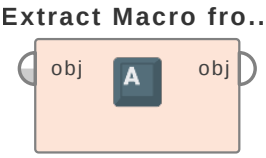



Figure 7.65: Tutorial process 'Annotating an object from a data set '.

A new data set is created and applied as annotations to the Golf data set.

# Extract Macro from Annotation




Extracts one or more annotations from an object and assigns its value to a macro.

## Description

For a general introduction to annotations please have a look at the help of the Annotate operator.

## Input Ports

**object *(obj)*** Any type of object can be connected to this port. The annotations will be extracted from this object.

## Output Ports

**object *(obj)*** The same object as passed to the input port, unchanged.

## Parameters

**extract all *(boolean)*** If checked, all annotations are extracted to macros named the same as the annotations. Optionally, you can define a name prefix which is prepended to the macro names

**macro *(string)*** Defines the name of the created macro.

**annotation *(string)*** The name of the annotation to be extracted.

**name prefix *(string)*** A prefix which is prepended to all macro names.

**fail on missing *(boolean)*** If checked, the operator breaks if the specified annotation can't be found; if it is unchecked an empty macro will be created.

## Related Documents

- **Annotate** (page **??**)
- **Extract Macro** (page **??**)

## Tutorial Processes

### Extracting the Source annotation from an example set

The process loads the Iris data set. The Retrieve operator automatically creates the Source annotation which specifies from where the data has been loaded.

The Extract Macro from Annotation operator then extracts the value of that annotation into a macro, which is printed to the console in the next step. Have a look at the process log after executing the process!

To see the annotations of an object go to the results view and select the Annotations display option.

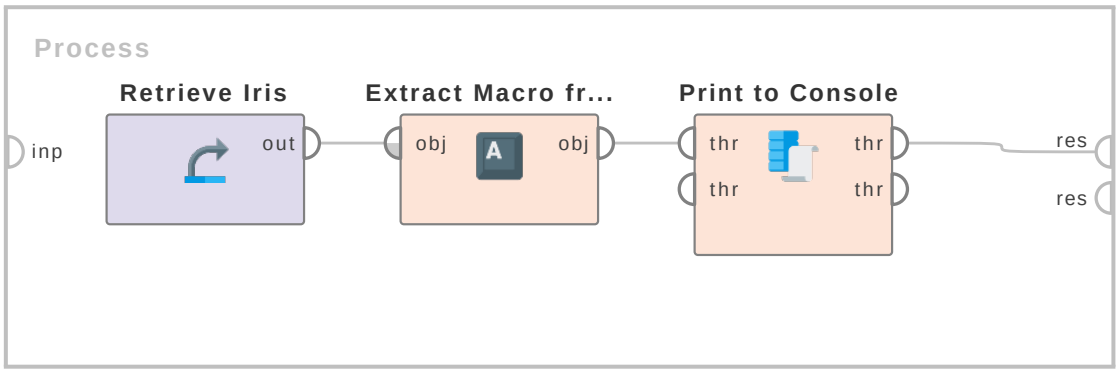
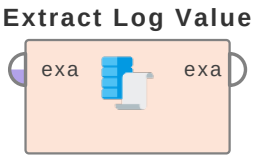



Figure 7.66: Tutorial process 'Extracting the Source annotation from an example set'.

# 7.6 Logging

## Extract Log Value




This operator reads the specified value from the input ExampleSet and provides the value for logging.

### Description

The Extract Log Value operator can be used for logging the value of the specified attribute at the specified index. The attribute name and index are specified through *attribute name* and *example index* parameters respectively. The values within an ExampleSet cannot be logged directly by the Log operator. The Extract Log Value operator makes the selected value loggable. This value can be logged by the Log operator. This value can be accessed in the Log operator at [Name of Extract Log Value operator][value][data_value]. Please study the attached Example Process for more information.

Logging and log-related operators store information into the log table. This information can be almost anything including parameter values of operators, apply-count of operators, execution time etc. The Log is mostly used when you want to see the values calculated during the execution of the process that are otherwise not visible. For example you want to see values of different parameters in all iterations of any Loop operator. For more information regarding logging please study the Log operator.

### Input Ports

**example set *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set *(exa)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**attribute name *(string)*** This parameter specifies the name of the attribute whose value should be provided for logging.

**example index *(integer)*** This parameter specifies the index of the example whose value should be provided for logging. Please note that negative indices are counted from the end of the data set. Positive counting starts with 1, negative counting starts with -1.

## Tutorial Processes

### Logging the value of an attribute by Extract Log Value operator

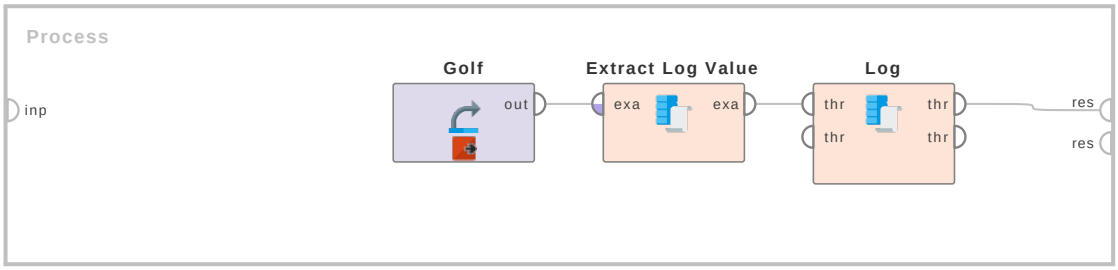
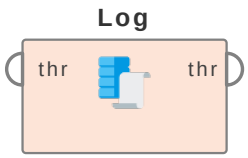



Figure 7.67: Tutorial process 'Logging the value of an attribute by Extract Log Value operator'.

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the 'Outlook' attribute has value 'sunny' in the first example. This value will be logged by the coming operators. The Extract Log Value operator is applied to provide this value as a loggable value. Finally the Log operator is applied to store this value in the log table. Have a look at the log parameter settings in the Log operator. Only one column named 'Value' has been defined. Please note carefully how the attribute value has been accessed. Run the process and you will see one column with just a single entry in the Table View of the results of the Log operator. You can see that it stores the value of the 'Outlook' attribute from the first example.

# Log




This operator stores information into the log table. This information can be almost anything including parameter values of operators, apply-count of operators, execution time etc. The stored information can be plotted by the GUI when the process execution is complete. Moreover, the information can also be written into a file.

## Description

The Log operator is mostly used when you want to see the values calculated during the execution of the process that are otherwise not visible. For example you want to see values of different parameters in all iterations of any Loop operator. In such scenarios the ideal operator is the Log operator. A large variety of information can be stored using this operator. Values of all parameters of all operators in the current process can be stored. Other information like apply-count, cpu-time, execution-time, loop-time etc can also be stored. The information stored in the log table can be viewed in the Results View. The information can also be analyzed in form of various graphs using the Plot View in the Results Workspace. The information can also be written directly into a file using the *filename* parameter.

The *log* parameter is used for specifying the information to be stored. The *column name* option specifies the name of the column in the log table (and/or file). Then you can select any operator from the drop down menu. Once you have selected an operator, you have two choices. You can either store a parameter value or store other values. If you opt for the parameter value, you can choose any parameter of the selected operator through the drop down menu. If you opt for other values, you can choose any value like apply-count, cpu-time etc from the last drop down menu.

Each time the Log operator is applied, all the values and parameters specified by the *log* parameter are collected and stored in a data row. When the process finishes, the operator writes the collected data rows into a file (if the *filename* parameter has a valid path). In GUI mode, 2D or 3D plots are automatically generated and displayed in the Results Workspace. Please study the attached Example Processes to understand working of this operator.

## Input Ports

**through *(thr)*** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the Log operator is available at the first *through* output port.

## Output Ports

**through *(thr)*** The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port. The Log operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Log operator is delivered at the first *through* output port

## Parameters

**filename *(filename)*** This parameter is used if you want to write the stored values into a file. The path of the file is specified here.

**log *(list)*** This is the most important parameter of this operator. It is used for specifying the values that should be stored by the Log operator. The *log* parameter is used for specifying the information to be stored. The *column name* option specifies the name of the column in the log table (and/or file). Then you can select any operator from the drop down menu. Once you have selected an operator, you have two choices. You can either store a parameter value or store other values. If you opt for the parameter value, you can choose any parameter of the selected operator through the drop down menu. If you opt for other values, you can choose any value like apply-count, cpu-time etc from the last drop down menu.

**sorting type *(selection)*** This parameter indicates if the logged values should be sorted according to the specified dimension.

**sorting dimension *(string)*** This parameter is only available when the *sorting type* parameter is set to 'top-k' or 'bottom-k'. This parameter is used for specifying the dimension that is to be used for sorting.

**sorting k *(integer)*** This parameter is only available when the *sorting type* parameter is set to 'top-k' or 'bottom-k'. Only *k* results will be kept.

**persistent *(boolean)*** This is an expert parameter. This parameter indicates if the results should be written to the specified file immediately.

## Tutorial Processes

### Introduction to the Log operator

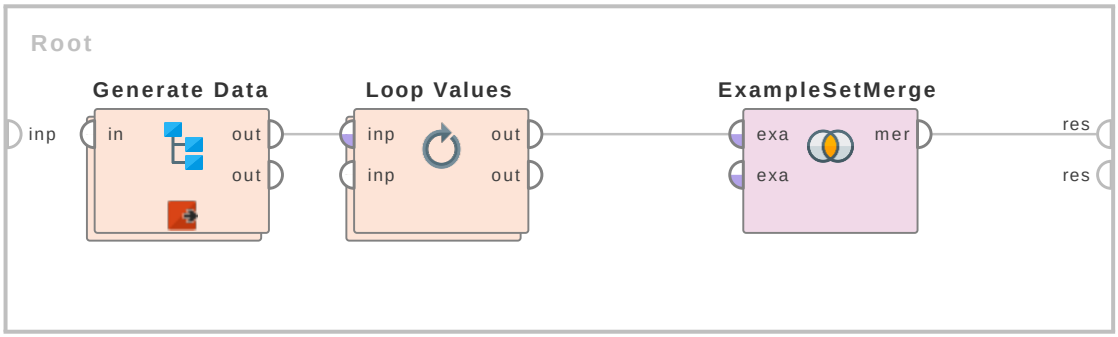



Figure 7.68: Tutorial process 'Introduction to the Log operator'.

This Example Process shows usage of the Log and Extract Macro operator in complex preprocessing. Other than concepts related to the Log operator, this process will cover a number of concepts of macros including redefining macros, macro of Loop Values operator and use of the Extract Macro operator. This process starts with a subprocess which is used to generate data. What is happening inside this subprocess is not relevant to the use of the Log operator, so it is

not discussed here. A breakpoint is inserted after this subprocess so that you can view the ExampleSet. You can see that the ExampleSet has 12 examples and 2 attributes: 'att1' and 'att2'. 'att1' is nominal and has 3 possible values: 'range1', 'range2' and 'range3'. 'att2' has real values.

The Loop Values operator is applied on the ExampleSet. It iterates over the values of the specified attribute (i.e. att1) and applies the inner operators on the given ExampleSet while the current value can be accessed via the macro defined by the iteration macro parameter. The iteration macro parameter is set to 'loop_value', thus the current value can be accessed by specifying %{loop_value} in the parameter values. As att1 has 3 possible values, the Loop Values operator will iterate 3 times, once for each possible value of att1.

Here is an explanation of what happens inside the Loop Values operator. The Loop Values operator is provided with an ExampleSet as input. The Filter Examples operator is applied on it. The condition class parameter is set to 'attribute value filter' and the parameter string is set to 'att1 = %{loop_value}'. Note the use of the loop_value macro here. Only those examples are selected where the value of att1 is equal to the value of the loop_value macro. A breakpoint is inserted here so that you can view the selected examples. Then Aggregation operator is applied on the selected examples. It is configured to take the average of the att2 values of the selected examples. This average value is stored in a new ExampleSet in an attribute named 'average(att2)'. A breakpoint is inserted here so that you can see the average of the att2 values of the selected examples. The Extract Macro operator is applied on this new ExampleSet to store this average value in a macro named 'current_average'. The originally selected examples are passed to the Generate Attributes operator that generates a new attribute named 'att2_abs_avg'. This attribute is defined by the expression 'abs(att2 - %{current_average})'. Note the use of the current_average macro here. Value of the current_average macro is subtracted from all values of att2 and stored in a new attribute named 'att2_abs_avg'. The Resultant ExampleSet is delivered at the output of the Loop Values operator. A breakpoint is inserted here so that you can see the ExampleSet with the 'att2_abs_avg' attribute. This output is fed to the Append operator in the main process. The Append operator merges the results of all the iterations into a single ExampleSet which is visible at the end of this process in the Results Workspace.

Note the Log operator in the subprocess of the Loop Values operator. Three columns are created using the log parameter. The 'Average att2' column stores the value of the macro of the Extract Macro operator. The 'Iteration' column stores the apply-count of the Aggregate operator which is the same as the number of iterations of the Loop Values operator. The 'att1 value' column stores the value of att1 in the current iteration. At the end of the process, you will see that the Log operator stores a lot of information that was not directly accessible. Moreover, it displays all the required information at the end of the process, thus breakpoints are not required.

Also note that the filename parameter of the Log operator is set to: 'D:\log.txt'. Thus a text file named 'log' is created in your 'D' drive. This file has the information stored during this process by the Log operator.

Here is what you see when you run the process:

The ExampleSet generated by the Generate Data subprocess. Then the process enters the Loop Value operator and iterates 3 times.

Iteration 1: The ExampleSet where the 'att1' value is equal to the current value of the loop_value macro i.e. 'range1'The average of the 'att2' values for the selected examples. The average is -1.161.The ExampleSet with the 'att2_abs_avg' attribute for iteration 1. Iteration 2: The ExampleSet where the 'att1' value is equal to the current value of loop_value macro i.e. 'range2'The Average of the 'att2' values for the selected examples. The average is -1.656.The ExampleSet with the 'att2_abs_avg' attribute for iteration 2.

Iteration 3: The ExampleSet where the 'att1' value is equal to the current value of loop_value macro i.e. 'range3'The Average of the 'att2' values for the selected examples. The average is 1.340.The ExampleSet with the 'att2_abs_avg attribute' for iteration 3. Now the process comes out of the Loop Values operator and the Append operator merges the final ExampleSets of all

three iterations into a single ExampleSet that you can see in the Results Workspace.

Now have a look at the results of the Log operator. You can see all the required values in tabular form using the Table View. You can see that all the values that were viewed using breakpoints are available in a single table. You can see the results in the Plot View as well. Also have a look at the file stored in the 'D' drive. This file has exactly the same information.

### Viewing Training vs Testing error using the Log operator

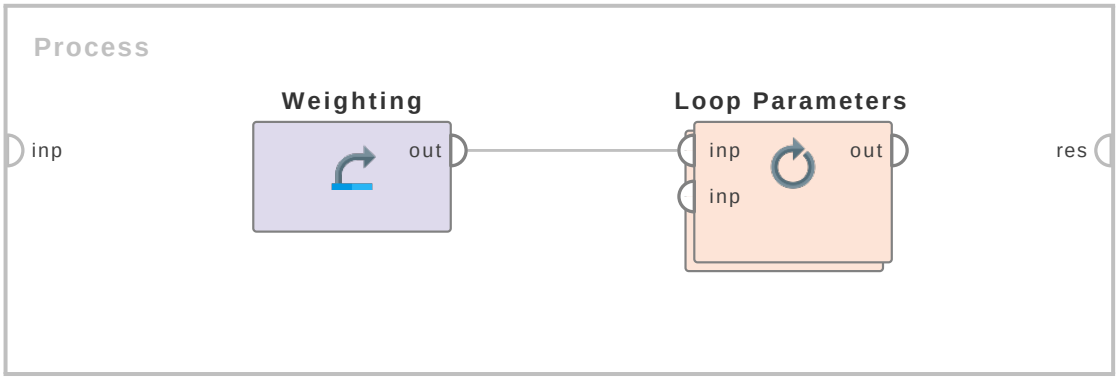



Figure 7.69: Tutorial process 'Viewing Training vs Testing error using the Log operator'.

The 'Weighting' is loaded using the Retrieve operator. The Loop Parameters operator is applied on it. The parameters of the Loop Parameters operator are set such that this operator loops 25 times. Thus its subprocess is executed 25 times. In every iteration, the value of the C parameter of the SVM(LibSVM) operator is changed. The value of the C parameter is 0.001 in the first iteration. The value is increased logarithmically until it reaches 100000 in the last iteration.

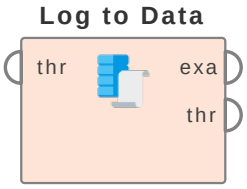
Have a look at the subprocess of the Loop Parameters operator. First the data is split into two equal partitions using the Split Data operator. The SVM (LibSVM) operator is applied on one partition. The resultant classification model is applied using two Apply Model operators on both the partitions. The statistical performance of the SVM model on both testing and training partitions is measured using the Performance (Classification) operators. At the end the Log operator is used to store the required results.

The log parameter of the Log operator stores four things. The iterations of the Loop Parameter operator are counted by apply-count of the SVM operator. This is stored in a column named 'Count'.The value of the classification error parameter of the Performance (Classification) operator that was applied on the Training partition is stored in a column named 'Training Error'. The value of the classification error parameter of the Performance (Classification) operator that was applied on the Testing partition is stored in a column named 'Testing Error'. The value of the C parameter of the SVM (LibSVM) operator is stored in a column named 'SVM C'. Also note that the stored information will be written into a file as specified in the filename parameter.

Run the process and turn to the Results View. You can see all the values in the Table View. This table can be used to study how classification errors in training and testing partitions behave with the increase in the value of the C parameter of the SVM(LibSVM) operator. To view these results in graphical form, switch to the Plot View. Select an appropriate plotter. You can use 'Series Multiple' plotter with 'SVM-C' as the 'Index Dimension'. Select 'Training Error' and 'Testing Error' in the 'Plot Series'. The 'scatter multiple' plotter can also be used. Now you can analyze how the training and testing error behaved with the increase in the parameter C.

Please note that since RapidMiner version 8.0, the Loop Parameters Operator has been updated to a) be parallel and b) log the parameter set and performance automatically. Please see the help of that operator for more information.

# Log to Data




This operator transforms the data generated by the Log operator into an ExampleSet which can then be used by other operators of the process.

## Description

The Log operator stores information into the log table. This information can be almost anything including parameter values of operators, apply-count of operators, execution time etc. The Log operator is mostly used when you want to see the values calculated during the execution of the process that are otherwise not visible. For example you want to see values of different parameters in all iterations of any Loop operator. In such scenarios the ideal operator is the Log operator. A large variety of information can be stored using this operator. The information stored in the log table can be viewed in the Results View. But this information is not directly accessible in the process. To solve this problem, the Log to Data operator provides the information in the Log table in form of an ExampleSet. This ExampleSet can be used in the process like any other ExampleSet. RapidMiner automatically guesses the type of attributes of this ExampleSet and all attributes have regular role. The type and role can be changed by using the corresponding operators.

## Input Ports

**through *(thr)*** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the Log to Data operator is available at the first *through* output port.

## Output Ports

**example set *(exa)*** The data generated by the Log operator is delivered as an ExampleSet through this port.

**through *(thr)*** The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port. The Log to Data operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Log to Data operator is delivered at the first *through* output port

## Parameters

**log name *(string)*** This parameter specifies the name of the Log operator that generated the log data which should be returned as an ExampleSet. If this parameter is left blank then the first found data table is returned as an ExampleSet.

## Tutorial Processes

### Accessing Training vs Testing error using the Log and Log to Data operators

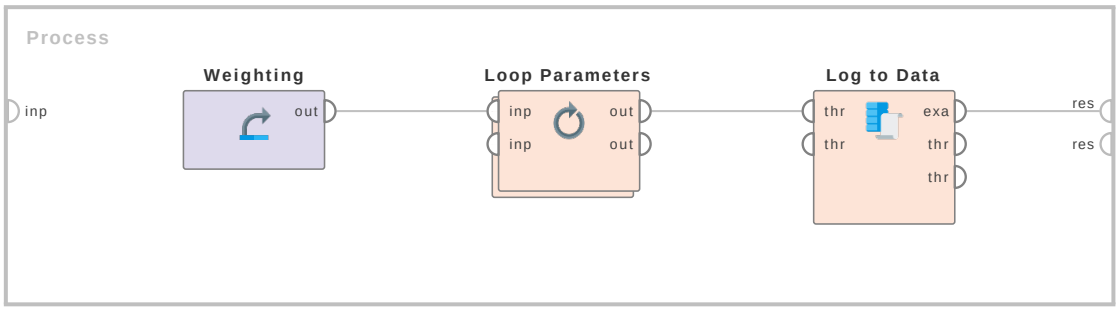



Figure 7.70: Tutorial process 'Accessing Training vs Testing error using the Log and Log to Data operators'.

The 'Weighting' data set is loaded using the Retrieve operator. The Loop Parameters operator is applied on it. The parameters of the Loop Parameters operator are set such that this operator loops 25 times. Thus its subprocess is executed 25 times. In every iteration, the value of the C parameter of the SVM(LibSVM) operator is changed. The value of the C parameter is 0.001 in the first iteration. The value is increased logarithmically until it reaches 100000 in the last iteration.

Have a look at the subprocess of the Loop Parameters operator. First the data is split into two equal partitions using the Split Data operator. The SVM (LibSVM) operator is applied on one partition. The resultant classification model is applied using two Apply Model operators on both the partitions. The statistical performance of the SVM model on both testing and training partitions is measured using the Performance (Classification) operators. At the end the Log operator is used to store the required results.

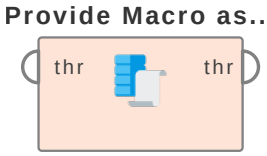
The log parameter of the Log operator stores four things. The iterations of the Loop Parameter operator are counted by apply-count of the SVM operator. This is stored in a column named 'Count'.The value of the classification error parameter of the Performance (Classification) operator that was applied on the Training partition is stored in a column named 'Training Error'. The value of the classification error parameter of the Performance (Classification) operator that was applied on the Testing partition is stored in a column named 'Testing Error'. The value of the C parameter of the SVM (LibSVM) operator is stored in a column named 'SVM C'.

In the main process, the Log to Data operator is used for providing the log values in form of an ExampleSet. The resultant ExampleSet is connected to the result port of the process and it can be seen in the Results Workspace. You can see the meta data of the ExampleSet in the Meta Data View and the values of the ExampleSet can be seen in the Data View. This ExampleSet can be used to study how classification errors in training and testing partitions behave with the increase in the value of the C parameter of the SVM(LibSVM) operator. To view these results in graphical form, switch to the Plot View. Select an appropriate plotter. You can use 'Series Multiple' plotter with 'SVM-C' as the 'Index Dimension'. Select 'Training Error' and 'Testing Error' in the 'Plot Series'. The 'scatter multiple' plotter can also be used. Now you can analyze how the training and testing error behaved with the increase in the parameter C. More importantly this ExampleSet is available in the process, so information stored in it can be used by other operators of the process.

Please note that since RapidMiner version 8.0, the Loop Parameters Operator has been updated to a) be parallel and b) log the parameter set and performance automatically. Please see

the help of that operator for more information.

# Provide Macro as Log Value




This operator reads the value of the specified macro and provides the macro value for logging. This operator is only useful if the required macro value cannot be logged directly.

## Description

The Provide Macro as Log Value operator can be used to log the current value of the specified macro. The name of the required macro is specified in the *macro name* parameter. Most operators provide the macro they define as a loggable value and in these cases this value can be logged directly. But in all other cases where the operator does not provide a loggable value for the defined macro, this operator may be used to do so. Please note that the value will be logged as a nominal value even if it is actually a numerical value.

You must be familiar with the basic concepts of macros and logging in order to understand this operator completely. Some basics of macros and logging are discussed in the coming paragraphs.

A macro can be considered as a value that can be used by all operators of the current process that come after the macro has been defined. Once the macro has been defined, the value of that macro can be used as a parameter value in coming operators by writing the macro name in *%{macro_name}* format in the parameter value where 'macro_name' is the name of the macro. Please note that many operators like many of the loop operators (e.g. Loop Values , Loop Attributes) also add specific macros. For more information regarding macros please study the Set Macro operator.

Logging and log-related operators store information into the log table. This information can be almost anything including parameter values of operators, apply-count of operators, execution time etc. The Log is mostly used when you want to see the values calculated during the execution of the process that are otherwise not visible. For example you want to see values of different parameters in all iterations of any Loop operator. For more information regarding logging please study the Log operator.

## Input Ports

**through *(thr)*** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the Provide Macro as Log Value operator is available at the first *through* output port.

## Output Ports

**through *(thr)*** The object that was given as input is passed without changing to the output through this port. It is not compulsory to attach this port to any other port. The Provide Macro as Log Value operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Provide Macro as Log Value operator is delivered at the first *through* output port

## Parameters

**macro name *(string)*** This parameter specifies the name of the macro whose value should be provided for logging.

## Tutorial Processes

### Logging the value of a macro with or without the Provide Macro as Log Value operator

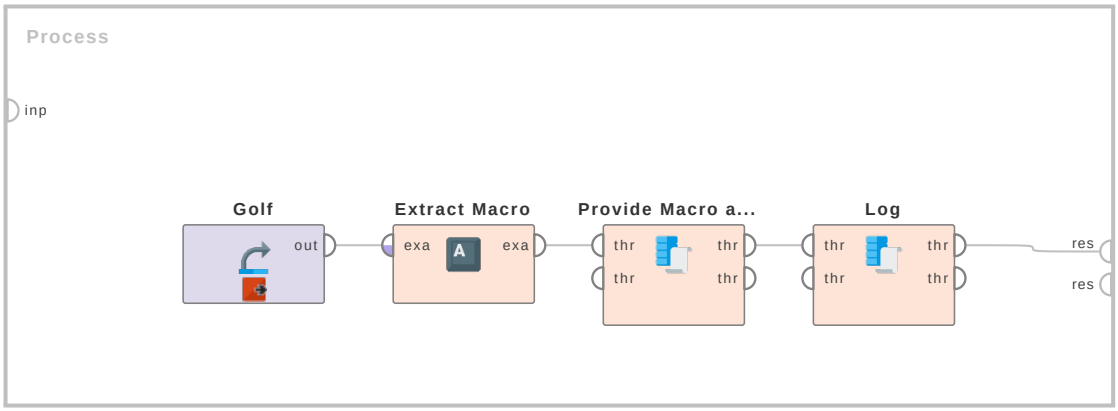
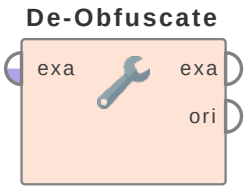



Figure 7.71: Tutorial process 'Logging the value of a macro with or without the Provide Macro as Log Value operator'.

This Example Process shows how macro values can be logged with or without the Provide Macro as Log Value operator. The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. As you can see, there are 14 examples in the ExampleSet. The Extract Macro operator is applied on the ExampleSet. The macro of the extract macro operator is named 'eCount' and it stores the number of examples in the ExampleSet i.e. 14. The Provide Macro as Log Value operator is applied to provide the value of the 'eCount' macro as a loggable value. Finally the Log operator is applied to store the value of the 'eCount' macro in the log table. Have a look at the log parameter settings in the Log operator. Two columns have been defined: 'Direct' and 'Indirect'. The 'Direct' column gets the value of the 'eCount' macro directly from the Extract Macro operator. This only works if the macro is provided in loggable form by the operator. The 'Indirect' column gets the value of the 'eCount' macro from the Provide Macro as Log Value operator. This is how a macro value should be logged if it cannot be logged directly. Run the process and you will see two columns in the Table View of the results of the Log operator. Both columns have the value of the 'eCount' macro (14).

# 7.7 Data Anonymization

## De-Obfuscate




Replaces all obfuscated values and attribute names by the ones given in a file.

### Description

This operator remaps the old values and names from an obfuscated ExampleSet according to an obfuscating map file. It can be used to deanonymize your data.

### Input Ports

**example set *(exa)*** This input port expects an ExampleSet.

### Output Ports

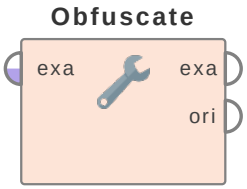
**example set *(exa)***

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

### Parameters

**obfuscation map file *(filename)*** File where the obfuscator map was written to.

# Obfuscate




Replaces all nominal values and attribute names by random strings.

## Description

This operator takes an ExampleSet as input and maps all nominal values to randomly created strings. The names and the construction descriptions of all attributes will also replaced by random strings. This operator can be used to anonymize your data. It is possible to save the obfuscating map into a file which can be used to remap the old values and names. Please use the operator De-Obfuscator for this purpose. The new example set can be written with an ExampleSetWriter.

## Input Ports

**example set *(exa)*** This input port expects an ExampleSet.

## Output Ports

**example set *(exa)***

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

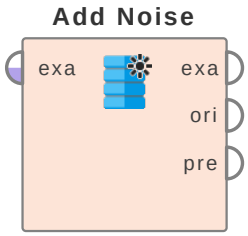
**obfuscation map file *(filename)*** File where the obfuscator map should be written to.

**use local random seed *(boolean)*** Indicates if a local random seed should be used.

**local random seed *(integer)*** Specifies the local random seed

# 7.8 Random Data Generation

## Add Noise




This operator adds noise in the given ExampleSet by adding random attributes to the ExampleSet and by adding noise in the existing attributes.

### Description

The Add Noise operator provides a number of parameters for selecting the attributes for adding noise in them. This operator can add noise to the label attribute or to the regular attributes separately. In case of a numerical label the given label noise (specified by the *label noise* parameter) is the percentage of the label range which defines the standard deviation of normal distributed noise which is added to the label attribute. For nominal labels the *label noise* parameter defines the probability to randomly change the nominal label value. In case of adding noise to regular attributes the *default attribute noise* parameter simply defines the standard deviation of normal distributed noise without using the attribute value range. Using the parameter list is also possible for setting different noise levels for different attributes (by using the *noise* parameter). However, it is not possible to add noise to nominal attributes.

The Add Noise operator can add random attributes to the ExampleSet. The number of random attributes is specified by the *random attributes* parameter. New random attributes are simply filled with random data which is not correlated to the label at all. The *offset* and *linear factor* parameters are available for adjusting the values of new random attributes.

### Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

### Output Ports

**example set output *(exa)*** Noise is added to the given ExampleSet and the resultant ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model *(pre)*** This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

## Parameters

**attribute filter type *(selection)*** This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameters panel.
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameters panel.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameters panel.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to *numeric* type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameters panel.
- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameters panel.
- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.
- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameters panel. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute *(string)*** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes *(string)*** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression *(string)*** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except regular expression** ***(string)*** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** ***(selection)*** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception** ***(boolean)*** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameters panel.

**except value type** ***(selection)*** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type** ***(selection)*** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception** ***(boolean)*** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameters panel.

**except block type** ***(selection)*** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** ***(string)*** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '$> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '$> 6$ && $< 11$' or '$<= 5$ || $< 0$'. But && and || cannot be used together in one numeric condition. Conditions like '($> 0$ && $< 2$) || ($>10$ && $< 12$)' are not allowed because they use both && and ||. Use a blank space after '$>$', '=' and '$<$' e.g. '$<5$' will not work, so use '$< 5$' instead.

**include special attributes** ***(boolean)*** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection** ***(boolean)*** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**random attributes** ***(integer)*** This parameter specifies the required number of new random attributes to add to the input ExampleSet.

**label noise** ***(real)*** This parameter specifies the noise to be added in the label attribute. In case of a numerical label the given label noise is the percentage of the label range which defines the standard deviation of normal distributed noise which is added to the label attribute.

For nominal labels the *label noise* parameter defines the probability to randomly change the nominal label value.

**default attribute noise *(real)*** This parameter specifies the default noise for all the selected regular attributes. The *default attribute noise* parameter simply defines the standard deviation of normal distributed noise without using the attribute value range

**noise *(list)*** This parameter gives the flexibility of adding different noises to different attributes by providing a list of noises for all attributes.

**offset *(real)*** The *offset* value is added to the values of all the random attributes created by this operator

**linear factor *(real)*** The *linear factor* value is multiplied with the values of all the random attributes created by this operator

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Adding noise to the Polynomial data set

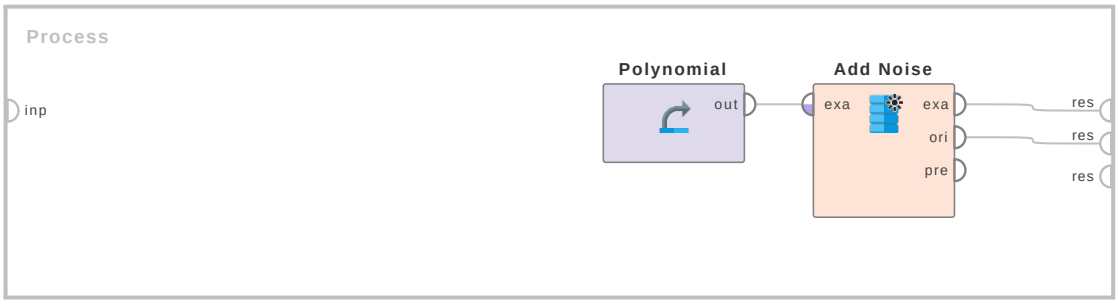
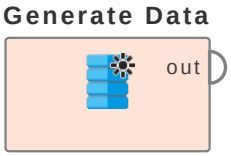



Figure 7.72: Tutorial process 'Adding noise to the Polynomial data set'.

The 'Polynomial' data set is loaded using the Retrieve operator. The Add Noise operator is applied on it. The attribute filter type parameter is set to 'all', thus noise will be added to all the attributes of the ExampleSet. The label noise and default attribute noise parameters are set to 0.05 and 0.06 respectively. The random attributes parameter is set to 1, thus a new random attribute will be added to the ExampleSet. The ExampleSet with noise and the original ExampleSet are connected to the result ports of the process. Both ExampleSets can be seen in the Results Workspace. You can see that there is a new random attribute in the ExampleSet generated by the Add Noise operator. By comparing the values of both ExampleSets you can see how noise was added by the Add Noise operator.

# Generate Data




This operator generates an ExampleSet based on numerical attributes. The number of attributes, number of examples, lower and upper bounds of attributes, and target function can be specified by the user.

## Description

The Generate Data operator generates an ExampleSet with a specified number of numerical attributes which is controlled by the *number of attributes* parameter. Please note that in addition to the specified number of regular attributes, the label attribute is automatically generated by applying the function selected by the *target function* parameter. The selected target function is applied on the attributes to generate the label attribute. For example if the *number of attributes* parameter is set to 3 and the *target function* is set to 'sum'. Then three regular numerical attributes will be created. In addition to these regular attributes a label attribute will be generated automatically. As the target function is set to 'sum', the label attribute value will be the sum of all three regular attribute values.

The label target functions are calculated as follows (assuming *n* generated attributes):

- random: The label is randomly generated.
- sum (needs at least 3 attributes): The label is the sum of the arguments: att1 + att2 + ... + att[*n*]
- polynomial (needs at least 3 attributes): att1^3 + att2^2 + att3
- non linear (needs at least 3 attributes): att1 * att2 * att3 + att1 * att2 + att2 * att2
- one variable non linear (needs 1 attribute): 3 * att1^3 - att1^2 + 1000 / |att1| + 2000 * |att1|
- complicated function (needs at least 3 attributes): att1 * att1 * att2 + att2 * att3 + *max(*att1,att2*)* - *e*^att3
- complicated function2 (needs at least 3 attributes): att1 * att1 * att1 + att2 * att2 + att1 * att2 + att1 / |att3| - 1 / (att3 * att3)
- simple sinus (needs 1 attribute): *sin(*att1*)*
- sinus (needs 2 attributes): *sin(*att1 * att2*)* + *sin(*att1 + att2*)*
- simple superposition (needs 1 attribute): 5 * *sin(*att1*)* + *sin(*30 * att1*)*
- sinus frequency (needs at least 2 attributes): 10 * *sin(*3 * att1*)* + 12 * *sin(*7 * att1*)* + 11 * *sin(* 5 * att2*)* + 9 * *sin(*10 * att2*)* + 10 * *sin(*8 * (att1 + att2*))*
- sinus with trend (needs 1 attribute): *sin(*att1*)* + 0.1 * att1
- sinc: *sin(*x*)* / ||x||, if ||x|| is not 0, else 0.
- triangular function (needs 1 attribute): The label is the fractional part of the argument.
- square pulse function (needs 1 attribute): The label is a square pulse in the attribute.
- random classification: The label is randomly "negative" or "positive".

- one third classification: The label is "positive" if att1 $<$ 0.3333 and "negative" else.
- sum classification: The label is "positive" if the sum of all arguments is positive, else "negative".
- quadratic classification (needs at least 2 attributes): The label is "positive" if att2 $>$ att1^2, else "negative".
- simple non linear classification (needs at least 2 attributes): The label is "positive" if 50 $<$ att1*att2 $<$ 80, else "negative".
- interaction classification (needs at least 3 attributes): The label is "positive" if att1 $<$ 0 or att2 $>$ 0 and att3 $<$ 0, else "negative".
- simple polynomial classification (needs at least 1 attribute): The label is "positive" if att1^4 $>$ 100, else "negative".
- polynomial classification (needs at least 4 attributes): The label is "positive" if att1^3 + att2^2 - att3^2 + att4 $>$ 0, else "negative".
- checkerboard classification (needs 2 attributes): The label is "positive" or "negative", according to a checkerboard pattern, where the size of each tile is 5.
- random dots classification (needs 2 attributes): Some randomly sized and placed positive and negative dots are generated on the 2D field. The label is "positive" if the example is only contained by positive dots, else "negative".
- global and local models classification (needs 2 attributes): The label is "positive" if the sum of both arguments is positive, else "negative". In addition, several local patterns in different sizes are placed in the data space.
- sinus classification (needs at least 2 attributes): The label is "positive" if *sin(*att1*att2*)* + *sin(*att1+att2*)* $>$ 0, else "negative".
- multi classification: The label is "one" if the sum of all arguments modulo 2 is 0, "two" if the sum modulo 3 is 0, "three" if the sum modulo 5 is 0, else "four".
- two gaussians classification: Generates two Gaussian clusters. The label is either "cluster0" or "cluster1".
- transactions dataset (needs at least 5 attributes): Generates an association function transaction dataset, all attribute values are 0 or 1. The first four attributes are correlated. No label is generated.
- grid function: Generates a uniformly distributed grid in the given dimensions. A label with zero value is generated.
- three ring clusters (needs 2 attributes): Generates three concentric ring clusters. The label values are "core", "first_ring" and "second_ring", accordingly.
- spiral cluster (needs 2 attributes): Generates two interlocking spiral clusters. The label values are "spiral1" and "spiral2", accordingly.
- single gaussian cluster: Generates a Gaussian cluster. A label with zero value is generated.
- gaussian mixture clusters: Generates a mixture of Gaussian clusters. Each attribute doubles the cluster amount, so 2^*n* clusters are generated. A label with the cluster id is generated.

- driller oscillation timeseries (needs at least 2 attributes): Generates an artificial audio data set (based on real-world data from drilling processes). No label is generated.

## Output Ports

**output *(out)*** The Generate Data operator generates an ExampleSet based on numerical attributes which is delivered through this port. The meta data is also delivered along with the data.This output is same as the output of the Retrieve operator.

## Parameters

**target function *(selection)*** This parameter specifies the target function for generating the label attribute. There are different options; users can choose any of them.

**number examples *(integer)*** This parameter specifies the number of examples to be generated.

**number of attributes *(integer)*** This parameter specifies the number of regular attributes to be generated. Please note that the *label* attribute is generated automatically besides these regular attributes.

**attributes lower bound *(real)*** This parameter specifies the minimum possible value for the attributes to be generated. In other words this parameter specifies the lower bound of the range of possible values of regular attributes. In case of target functions using Gaussian distribution, the attribute values may exceed this value.

**attributes upper bound *(real)*** This parameter specifies the maximum possible value for the attributes to be generated. In other words this parameter specifies the upper bound of the range of possible values of regular attributes. In case of target functions using Gaussian distribution, the attribute values may exceed this value.

**gaussian standard deviation *(real)*** This parameter specifies the standard deviation of the Gaussian distribution used for generating attributes.

**largest radius *(real)*** This parameter specifies the radius of the outermost ring cluster.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same ExampleSet. Changing the value of this parameter changes the way examples are randomized, thus the ExampleSet will have a different set of values.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

**data management *(selection)*** This is an expert parameter. A long list is provided; users can select any option from this list.

## Tutorial Processes

### Introduction to the Generate Data operator

The Generate Data operator is applied for generating an ExampleSet. The target function parameter is set to 'sum', thus the label attribute will be the sum of all attributes' values. The number examples parameter is set to 100, thus the ExampleSet will have 100 examples. The

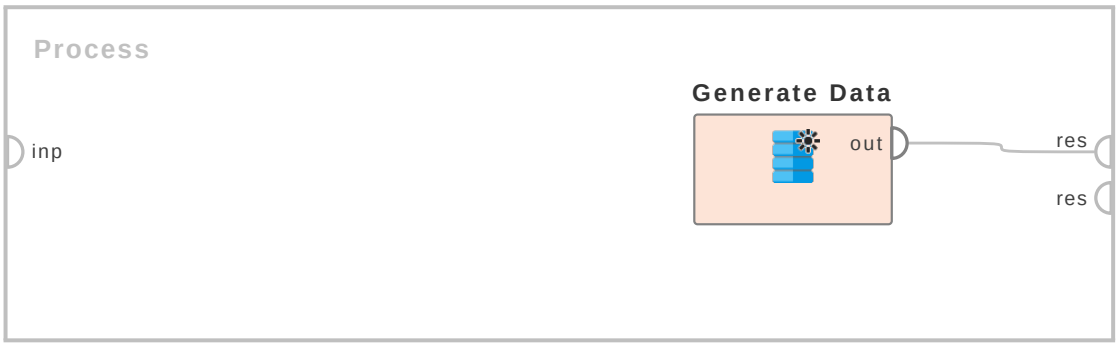
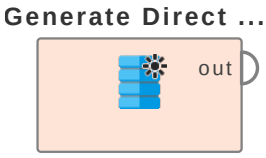



Figure 7.73: Tutorial process 'Introduction to the Generate Data operator'.

number of attributes parameter is set to 3, thus three numerical attributes will be generated beside the label attribute. The attributes lower bound and attributes upper bound parameters are set to -10 and 10 respectively, thus values of the regular attributes will be within this range. You can verify this by viewing the results in the Results Workspace. The use local random seed parameter is set to false in this Example process. Set the use local random seed parameter to true and run the process with different values of local random seed. You will see that changing the values of local random seed changes the randomization.

# Generate Direct Mailing Data




This operator generates an ExampleSet that represents direct mailing data. The number of examples can be specified by the user.

## Description

The Generate Direct Mailing Data operator generates an ExampleSet that represents direct mailing data. This ExampleSet can be used when you do not have a data set that represents a real direct mailing data. This ExampleSet can be used as a placeholder for such a requirement. This data set has 8 regular attributes and 1 special attribute. The regular attributes are name (nominal), age (integer), lifestyle (nominal), zip code (integer), family status (nominal), car (nominal), sports (nominal) and earnings (integer). The special attribute is label (nominal). The number of examples in the data set can be set by the *number examples* parameter. To have a look at this ExampleSet, just run the attached Example Process.

## Output Ports

**output *(out)*** The Generate Direct Mailing Data operator generates an ExampleSet which is delivered through this port. The meta data is also delivered along with the data.This output is same as the output of the Retrieve operator.

## Parameters

**number examples *(integer)*** This parameter specifies the number of examples to be generated.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same ExampleSet. Changing the value of this parameter changes the way examples are randomized, thus the ExampleSet will have a different set of values.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

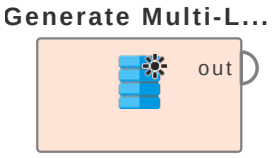
## Tutorial Processes

### Introduction to the Generate Direct Mailing Data operator

The Generate Direct Mailing Data operator is applied for generating an ExampleSet that represents direct mailing data. The number examples parameter is set to 10000, thus the ExampleSet will have 10000 examples. You can see the ExampleSet in the Results Workspace. The use local random seed parameter is set to false in this Example Process. Set the use local random seed parameter to true and run the process with different values of local random seed. You will see that changing the values of local random seed changes the randomization.


Figure 7.74: Tutorial process 'Introduction to the Generate Direct Mailing Data operator'.

# Generate Multi-Label Data




This operator generates a multi-label ExampleSet based on numerical attributes. The number of examples, lower and upper bounds of attributes can be specified by the user.

## Description

The Generate Multi-Label Data operator generates an ExampleSet with 5 numerical attributes and 3 label attributes. If the *regression* parameter is set to false, then the labels have two possible values i.e. positive or negative. Otherwise, the labels have real values. The number of examples to be generated can be specified by the *number examples* parameter. The upper and lower bounds of the numerical values can be specified by the *attributes upper bound* and *attributes lower bound* parameters. This operator is used for generating a random ExampleSet for testing purposes.

## Output Ports

**output *(out)*** The Generate Multi-Label Data operator generates a multi-label ExampleSet based on numerical attributes which is delivered through this port. The meta data is also delivered along with the data.This output is the same as the output of the Retrieve operator.

## Parameters

**number examples *(integer)*** This parameter specifies the number of examples to be generated.

**regression *(boolean)*** This parameter specifies if multiple labels for regression tasks should be generated. If this parameter is set to false, then the labels have two possible values i.e. positive or negative. Otherwise, the labels have real values.

**attributes lower bound *(real)*** This parameter specifies the minimum possible value for the attributes to be generated. In other words this parameter specifies the lower bound of the range of possible values of regular attributes.

**attributes upper bound *(real)*** This parameter specifies the maximum possible value for the attributes to be generated. In other words this parameter specifies the upper bound of the range of possible values of regular attributes.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same ExampleSet. Changing the value of this parameter changes the way examples are randomized, thus the ExampleSet will have a different set of values.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

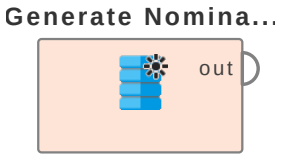
## Tutorial Processes

### Introduction to the Generate Multi-Label Data operator


Figure 7.75: Tutorial process 'Introduction to the Generate Multi-Label Data operator'.

The Generate Multi-Label Data operator is applied for generating an ExampleSet. The number examples parameter is set to 100, thus the ExampleSet will have 100 examples. The attributes lower bound and attributes upper bound parameters are set to -10 and 10 respectively, thus values of the regular attributes will be within this range. The regression parameter is set to false, thus the ExampleSet will have nominal labels. You can verify this by viewing the results in the Results Workspace. The use local random seed parameter is set to false in this Example process. Set the use local random seed parameter to true and run the process with different values of local random seed. You will see that changing the values of local random seed changes the randomization.

# Generate Nominal Data




This operator generates an ExampleSet based on nominal attributes. The number of examples, number of attributes, and number of values can be specified by the user.

## Description

The Generate Nominal Data operator generates an ExampleSet with the specified number of nominal attributes which is controlled by the *number of attributes* parameter. Please note that in addition to the specified number of regular attributes, the label attribute is automatically generated. The label attribute generated by this operator has only two possible values i.e. positive and negative. This operator is used for generating a random ExampleSet for testing purposes.

## Output Ports

**output *(out)*** The Generate Nominal Data operator generates an ExampleSet based on nominal attributes which is delivered through this port. The meta data is also delivered along with the data.This output is the same as the output of the Retrieve operator.

## Parameters

**number examples *(integer)*** This parameter specifies the number of examples to be generated.

**number of attributes *(integer)*** This parameter specifies the number of regular attributes to be generated. Please note that the *label* attribute is generated automatically besides these regular attributes.

**number of values *(integer)*** This parameter specifies the number of unique values of the attributes.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same ExampleSet. Changing the value of this parameter changes the way examples are randomized, thus the ExampleSet will have a different set of values.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

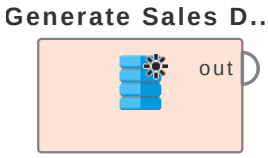
### Introduction to the Generate Nominal Data operator

The Generate Nominal Data operator is applied for generating an ExampleSet. The number examples parameter is set to 100, thus the ExampleSet will have 100 examples. The number of attributes parameter is set to 3, thus three nominal attributes will be generated beside the label attribute. The number of values parameter is set to 5, thus each attribute will have 5 possible values. You can verify this by viewing the results in the Results Workspace. The use local random seed parameter is set to false in this Example process. Set the use local random seed parameter


Figure 7.76: Tutorial process 'Introduction to the Generate Nominal Data operator'.

to true and run the process with different values of local random seed. You will see that changing the values of local random seed changes the randomization.

# Generate Sales Data




This operator generates an ExampleSet that represents sales data. The number of examples can be specified by the user.

## Description

The Generate Sales Data operator generates an ExampleSet that represents sales data. This ExampleSet can be used when you do not have a data set that represents actual sales data. This ExampleSet can be used as a placeholder for such a requirement. This data set has 7 regular attributes and 1 special attribute. The regular attributes are store_id (nominal), customer_id (nominal), product_id (integer), product_category (nominal), date (date), amount (integer) and single_price (real). The special attribute is transaction_id (integer) which is an id attribute. The number of examples in the data set can be set by the *number examples* parameter. To have a look at this ExampleSet, just run the attached Example Process.

## Output Ports

**output *(out)*** The Generate Sales Data operator generates an ExampleSet which is delivered through this port. The meta data is also delivered along with the data.This output is the same as the output of the Retrieve operator.

## Parameters

**number examples *(integer)*** This parameter specifies the number of examples to be generated.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same ExampleSet. Changing the value of this parameter changes the way examples are randomized, thus the ExampleSet will have a different set of values.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is available only if the *use local random seed* parameter is set to true.

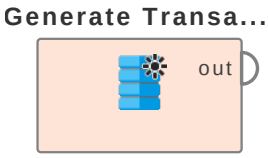
## Tutorial Processes

### Introduction to the Generate Sales Data operator

The Generate Sales Data operator is applied for generating an ExampleSet that represents sales data. The number examples parameter is set to 10000, thus the ExampleSet will have 10000 examples. You can see the ExampleSet in the Results Workspace. The use local random seed parameter is set to false in this Example Process. Set the use local random seed parameter to true and run the process with different values of local random seed. You will see that changing the values of local random seed changes the randomization.


Figure 7.77: Tutorial process 'Introduction to the Generate Sales Data operator'.

# Generate Transaction Data




This operator generates an ExampleSet that represents transaction data. The number of transactions, number of customers, number of items and number of clusters can be specified by the user.

## Description

The Generate Transaction Data operator generates an ExampleSet representing transaction data. This ExampleSet can be used when you do not have a data set that represents a real transaction data. It can also be used as a placeholder for such a requirement. This data set has 2 regular attributes and 1 special attribute. The regular attributes are Item (nominal) and Amount (integer). The special attribute is Id (nominal). This Id attribute represents the customer Id. All items purchased by a single customer are listed in form of multiple examples with the same customer Id. The Item attribute tells which item was purchased and the Amount attribute tells the quantity of the item that was purchased. The number of transactions can be set by the *number transactions* parameter. To have a look at this ExampleSet, just run the attached Example Process.

## Output Ports

**output *(out)*** The Generate Transaction Data operator generates an ExampleSet which is delivered through this port. The meta data is also delivered along with the data.This output is the same as the output of the Retrieve operator.

## Parameters

**number transactions *(integer)*** This parameter specifies the number of generated transactions.

**number customers *(integer)*** This parameter specifies the number of generated customers.

**number items *(integer)*** This parameter specifies the number of generated items.

**number clusters *(integer)*** This parameter specifies the number of generated clusters.

**use local random seed *(boolean)*** This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same ExampleSet. Changing the value of this parameter changes the way examples are randomized, thus the ExampleSet will have a different set of values.

**local random seed *(integer)*** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Introduction to the Generate Transaction Data operator


Figure 7.78: Tutorial process 'Introduction to the Generate Transaction Data operator'.

The Generate Transaction Data operator is applied for generating an ExampleSet that represents transaction data. The number transactions parameter is set to 1000, thus the ExampleSet will have 1000 examples. The number customers parameter is set to 50, thus there will be 50 unique values in the Id attribute. The number items parameter is set to 80, thus there will be 80 unique values in the Item attribute. You can see the ExampleSet in the Results Workspace. The use local random seed parameter is set to false in this Example Process. Set the use local random seed parameter to true and run the process with different values of local random seed. You will see that changing the values of local random seed changes the randomization.

# 7.9 Misc

## Free Memory


This operator frees unused memory. It might be useful after large preprocessing chains with a lot of currently unused views or even data copies.

### Description

The Free Memory operator cleans up unused memory resources. It might be very useful in combination with the Materialize Data operator after large preprocessing trees using a lot of views or data copies. It can be very useful after the data set was materialized in memory. Internally, this operator simply invokes a garbage collection from the underlying Java programming language.

Please note that RapidMiner keeps data sets in memory as long as possible. So if there is any memory left, RapidMiner will not discard previous results of the process or data at the port. The Free Memory operator can be useful if you get the *OutOfMemoryException*. Also note that operators like the Remember operator put the objects in the Store. The Free Memory operator does not clean up the store. This operator will only free memory which is no longer needed which is not the case if the object is in the Store.

After process execution has been completed, everything including the Stores is freed, but only if needed! So you can take a look at your previous results in the Result History as long as they fit into the memory. RapidMiner will automatically discard all these Results if a currently running process or new result needs free memory. So the memory usage will constantly grow with the time until it has reached a peak value and RapidMiner starts to discard previous results.

### Input Ports

**through *(thr)*** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the Free Memory operator is available at the first *through* output port.

### Output Ports

**through *(thr)*** The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the memory is freed even if this port is left without connections. The Free Memory operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Free Memory operator is delivered at the first *through* output port

### Tutorial Processes

### Introduction to the Free Memory operator

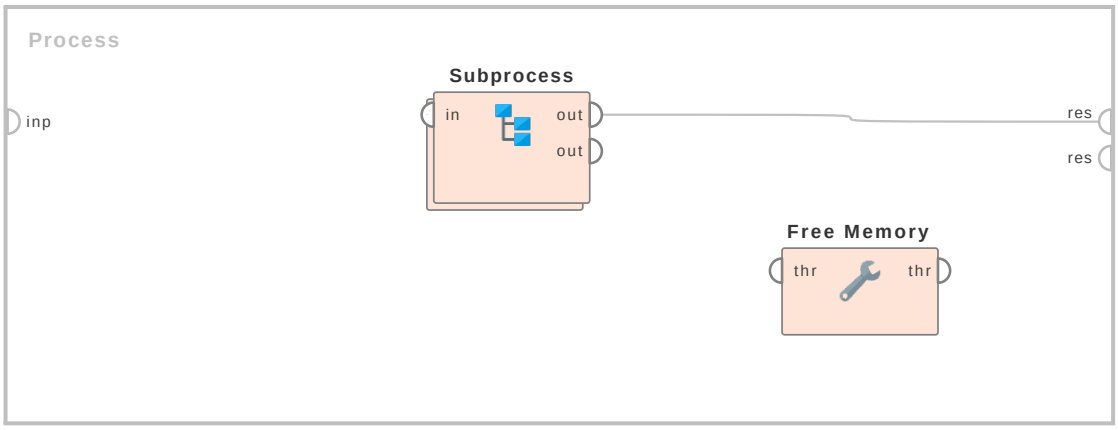



Figure 7.79: Tutorial process 'Introduction to the Free Memory operator'.

This is a very simple Example Process which just shows how to use the Free Memory operator. First the Subprocess operator is applied. Suppose we have a memory intensive task in the subprocess and we want to free unused memory after the subprocess is complete. The Free Memory operator is applied after the Subprocess operator to free the unused memory. No memory intensive task is performed in this Example Process. This Process was intended to discuss only the way this operator can be applied. Please make sure that the operators are applied in correct sequence. Also note that the Free Memory operator is not connected to any other operator but still it can perform its task.

# Join Paths


This operator delivers the first non-null input to its output.

## Description

The Join Paths operator can have multiple inputs but it has only one output. This operator returns the first non-null input that it receives. This operator can be useful when some parts of the process are susceptible of producing null results which can halt the entire process. In such a scenario the Join Paths operator can be used to filter out this possibility.

## Input Ports

**input *(inp)*** This operator can have multiple inputs. When one input is connected, another *input* port becomes available which is ready to accept another input (if any). Multiple inputs can be provided but only the first non-null object will be returned by this operator.

## Output Ports

**output *(out)*** The first non-null object that this operator receives is returned through this port.

## Tutorial Processes

### Returning the first non-null object

This Example Process starts with the Subprocess operator. Two outputs of the Subprocess operator are attached to the first two input ports of the Join Paths operator. But both these inputs are null because the Subprocess operator has no inner operators. The 'Golf' and 'Polynomial' data sets are loaded using the Retrieve operator. The Join Paths operator has four inputs but it returns only the 'Golf' data set because it is the first non-null input that it received.

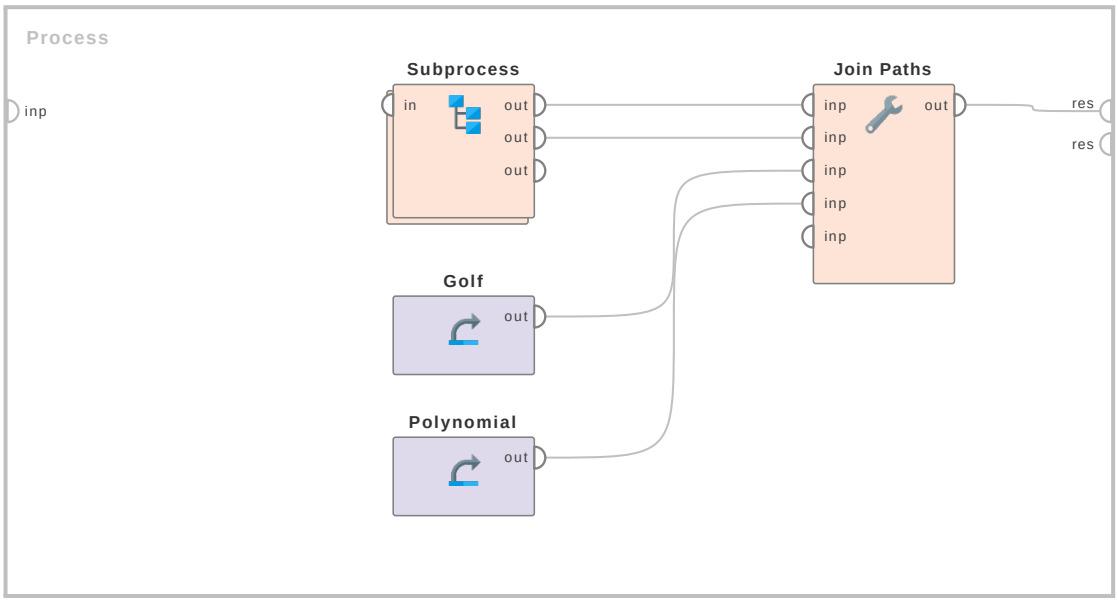
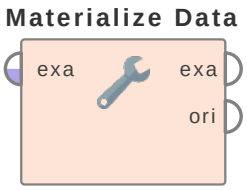



Figure 7.80: Tutorial process 'Returning the first non-null object'.

# Materialize Data




This operator creates a fresh and clean copy of the data in the memory.

## Description

The Materialize Data operator creates a fresh and clean copy of the data in the memory. It might be useful after large preprocessing chains with a lot of views or even data copies. In such cases, it can be especially useful in combination with a memory cleanup operator e.g. the Free Memory operator.

## Input Ports

**example set input *(exa)*** This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input.

## Output Ports

**example set output *(exa)*** The fresh and clean copy of the ExampleSet is delivered through this port.

**original *(ori)*** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

## Parameters

**datamanagement *(selection)*** This is an expert parameter. There are different options, users can choose any of them

## Tutorial Processes

### Creating fresh copy of an ExampleSet

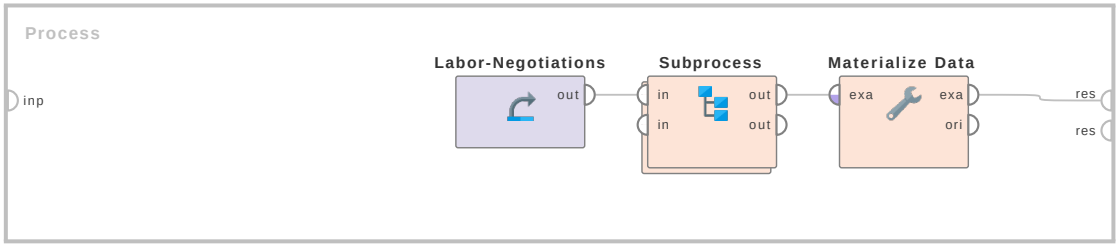
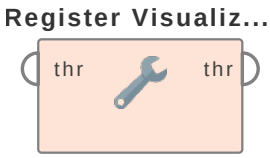



Figure 7.81: Tutorial process 'Creating fresh copy of an ExampleSet'.

This is a very simple Example Process which just shows how to use the Materialize Data operator. The 'Labor-Negotiations' data set is loaded using the Retrieve operator. The Subprocess operator is applied on it. No operator is applied in the Subprocess operator because it is used as a dummy operator here. Suppose we have large preprocessing chains with a lot of views or even data copies in the subprocess and we want a fresh copy of data after the subprocess is complete. The Materialize Data operator is applied after the Subprocess operator to create a fresh and clean copy of the data. No large preprocessing tasks were performed in this Example Process because this Process was intended to discuss only the way this operator can be applied.

# Register Visualization from Database




Allows the visualization of examples (attribute values) in the plot view of an example set (double click on data point). The data is directly derived from the specified database table.

## Description

Queries the database table for the row with the requested ID and creates a generic example visualizer. This visualizer simply displays the attribute values of the example. Adding this operator is might be necessary to enable the visualization of single examples in the provided plotter or graph components. In contrast to the usual example visualizer, this version does not load the complete data set into memory but simply queries the information from the database and just shows the single row.

## Input Ports

**through *(thr)*** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the operator is available at the first *through* output port.

## Output Ports

**through *(thr)*** The objects that were given as input are passed without changing to the output through this port. It is not compulsory to connect this port to any other port. The operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the operator is delivered at the first *through* output port.

## Parameters

**define connection *(selection)*** Indicates how the database connection should be specified.

**connection *(selection)*** A predefined database connection.

**database system *(selection)*** The used database system.

**database url *(string)*** The URL connection string for the database, e.g. 'jdbc:mysql://foo.bar:portnr/database'

**username *(string)*** The database username.

**password *(string)*** The password for the database.

**jndi name *(string)*** JNDI name for a data source.

**use default schema *(boolean)*** If checked, the user's default schema will be used.

**schema name** ***(string)*** The schema name to use, unless use_default_schema is true.

**table name** ***(string)*** A database table.

**id column** ***(string)*** The column of the table holding the object ids for detail data querying.

## Related Documents

- **Read Database** (page 52)

rapidminer

Global leader in predictive analytics software.
Boston | London | Dortmund | Budapest
www.rapidminer.com