

---

# **Orange Visual Programming Documentation**

***Release 3***

**Orange Data Mining**

**Sep 05, 2017**



---

## Contents

---

<b>1</b>	<b>Getting Started</b>	<b>1</b>
<b>2</b>	<b>Widgets</b>	<b>13</b>





Here we need to copy the getting started guide.

### Loading your Data

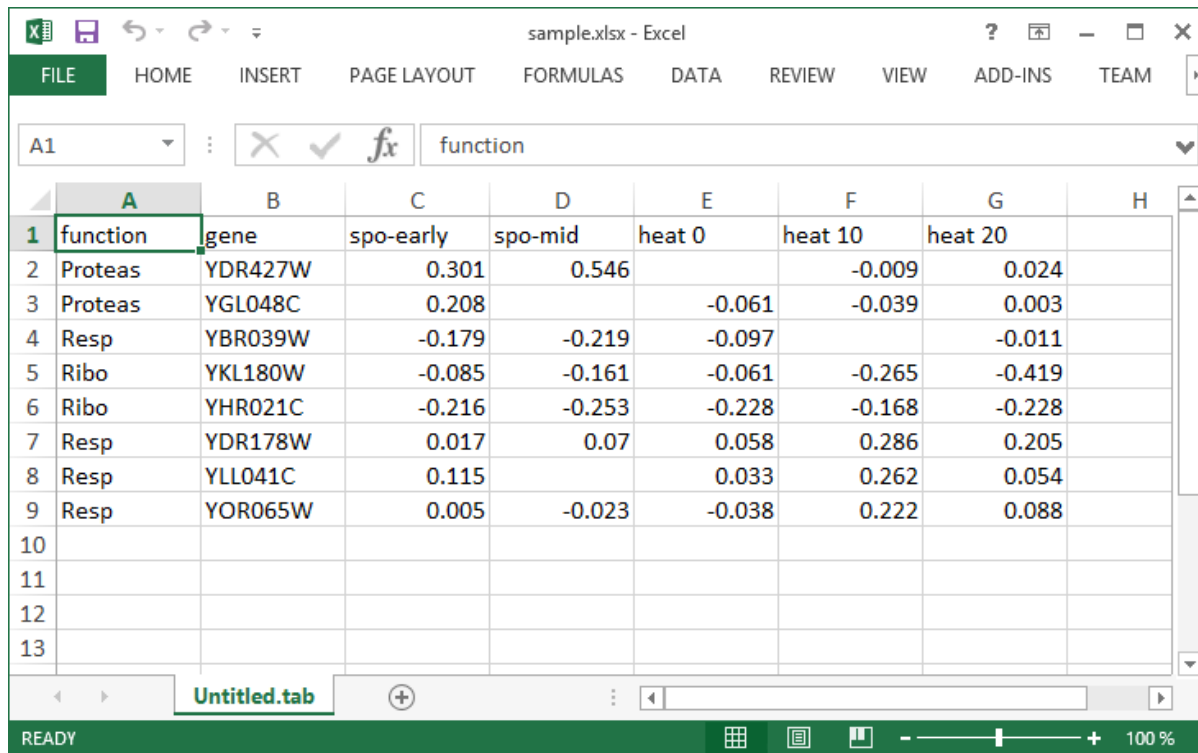
Orange comes with its [own data format](#), but can also handle native Excel (.xlsx or .xls), comma- or tab-delimited data files. The input data set is usually a table, with data instances (samples) in rows and data attributes in columns. Attributes can be of different *types* (continuous, discrete, time, and strings) and have assigned *roles* (input features, meta attributes, and class). Data attribute type and role can be provided in the data table header. They can also be subsequently changed in the [File](#) widget, while data role can also be modified with [Select Columns](#) widget.

### In a Nutshell

- Orange can import any comma- or tab-delimited data file, or Excel's native files or Google Sheets document. Use [File](#) widget to load the data and, if needed, define the class and meta attributes.
- Attribute names in the column header can be preceded with a label followed by a hash. Use c for class and m for meta attribute, i to ignore a column, w for weights column, and C, D, T, S for continuous, discrete, time, and string attribute types. Examples: C#mph, mS#name, i#dummy.
- An alternative to the hash notation is Orange's native format with three header rows: the first with attribute names, the second specifying the type (**continuous**, **discrete**, **time**, or **string**), and the third providing information on the attribute role (**class**, **meta**, **weight** or **ignore**).

### Example: Data from Excel

Here is an example data set (download it from `sample.xlsx`) as entered in Excel:



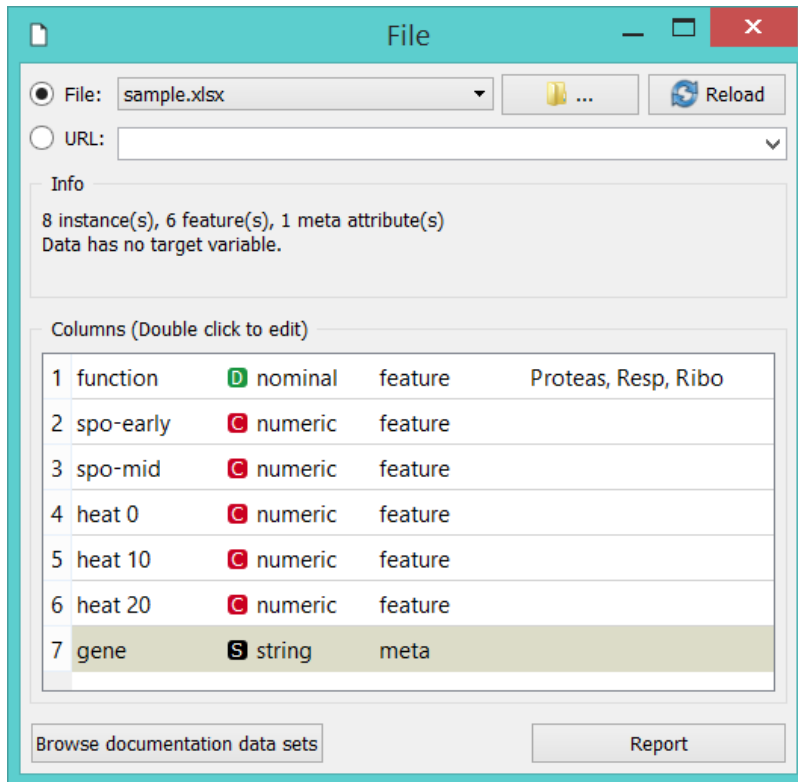
	A	B	C	D	E	F	G	H
1	function	gene	spo-early	spo-mid	heat 0	heat 10	heat 20	
2	Proteas	YDR427W	0.301	0.546		-0.009	0.024	
3	Proteas	YGL048C	0.208		-0.061	-0.039	0.003	
4	Resp	YBR039W	-0.179	-0.219	-0.097		-0.011	
5	Ribo	YKL180W	-0.085	-0.161	-0.061	-0.265	-0.419	
6	Ribo	YHR021C	-0.216	-0.253	-0.228	-0.168	-0.228	
7	Resp	YDR178W	0.017	0.07	0.058	0.286	0.205	
8	Resp	YLL041C	0.115		0.033	0.262	0.054	
9	Resp	YOR065W	0.005	-0.023	-0.038	0.222	0.088	
10								
11								
12								
13								

The file contains a header row, eight data instances (rows) and seven data attributes (columns). Empty cells in the table denote missing data entries. Rows represent genes; their function (class) is provided in the first column and their name in the second. The remaining columns store measurements that characterize each gene. With this data, we could, say, develop a classifier that would predict gene function from its characteristic measurements.

Let us start with a simple workflow that reads the data and displays it in a table:



To load the data, open the File widget (double click on the icon of the widget), click on the file browser icon ("...") and locate the downloaded file (from `sample.xlsx`) on your disk:



### File Widget: Setting the Attribute Type and Role

The **File** widget sends the data to the **Data Table**. Double click the **Data Table** to see its contents:

**Data Table**

**Info**

8 instances  
 5 features (10.0% missing values)  
 Continuous target variable (no missing values)  
 1 meta attribute (no missing values)

Restore Original Order

**Variables**

☒ Show variable labels (if present)  
☐ Visualize continuous values  
☒ Color by instance classes

**Selection**

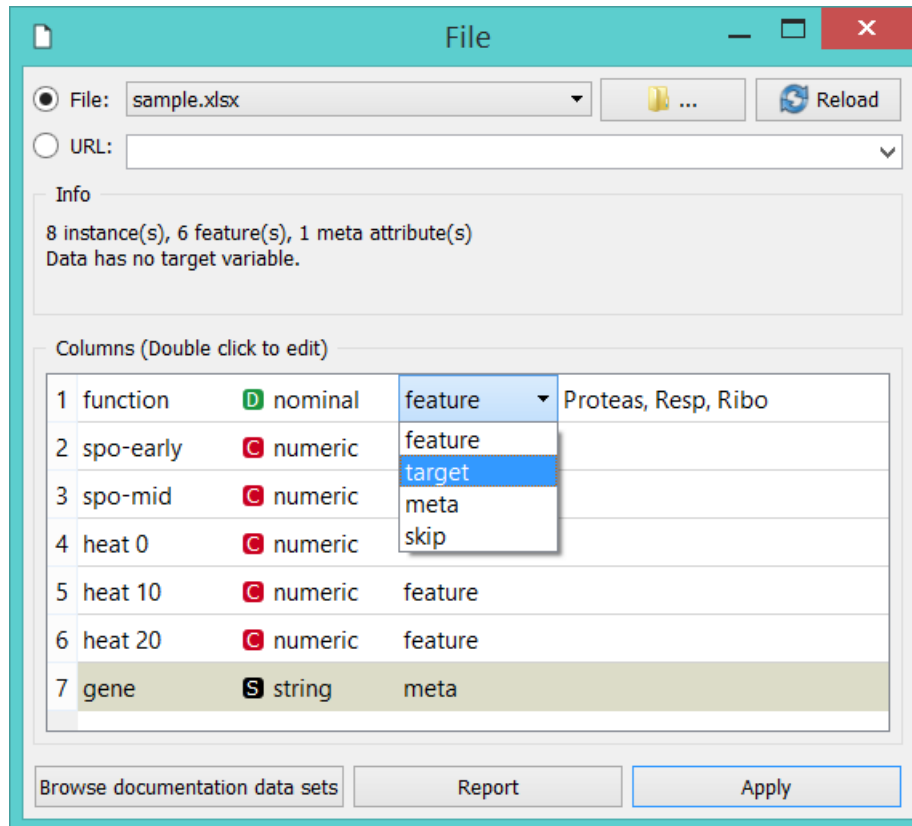
☒ Select full rows

☒ Auto send is on

Report

	function	spo-early	spo-mid	heat 0	heat 10
1	Proteas	0.301	0.546	?	-0.009
2	Proteas	0.208	?	-0.061	-0.039
3	Resp	-0.179	-0.219	-0.097	?
4	Ribo	-0.085	-0.161	-0.061	-0.265
5	Ribo	-0.216	-0.253	-0.228	-0.168
6	Resp	0.017	0.070	0.058	0.286
7	Resp	0.115	?	0.033	0.262
8	Resp	0.005	-0.023	-0.038	0.222

Orange correctly assumed that a column with gene names is meta information, which is displayed in the **Data Table** in columns shaded with light-brown. It has not guessed that *function*, the first non-meta column in our data file, is a class column. To correct this in Orange, we can adjust attribute role in the column display of File widget (below). Double-click the *feature* label in the *function* row and select *target* instead. This will set *function* attribute as our target (class) variable.



You can also change attribute type from nominal to numeric, from string to datetime, and so on. Naturally, data values have to suit the specified attribute type. Datetime accepts only values in [ISO 8601](#) format, e.g. 2016-01-01 16:16:01. Orange would also assume the attribute is numeric if it has several different values, else it would be considered nominal. All other types are considered strings and are as such automatically categorized as meta attributes.

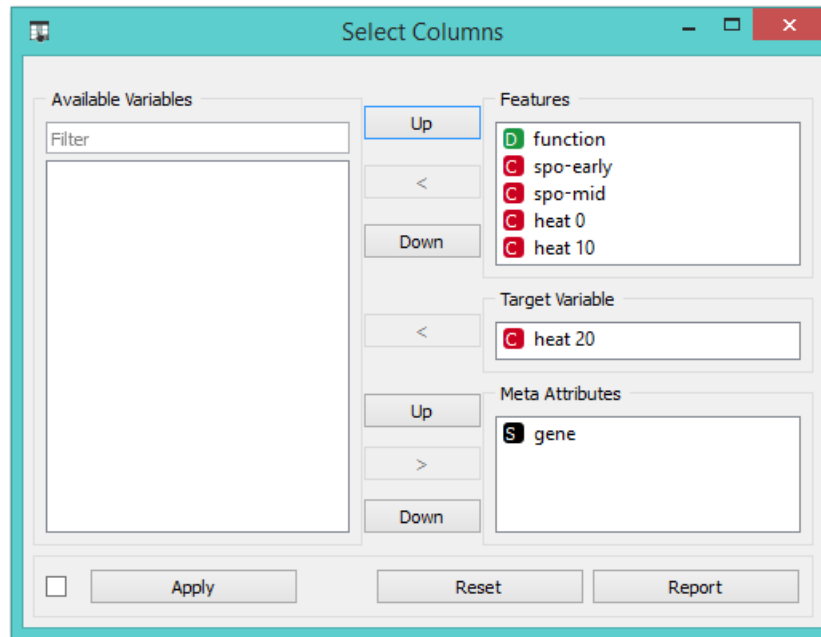
Change of attribute roles and types should be confirmed by clicking the **Apply** button.

### Select Columns: Setting the Attribute Role

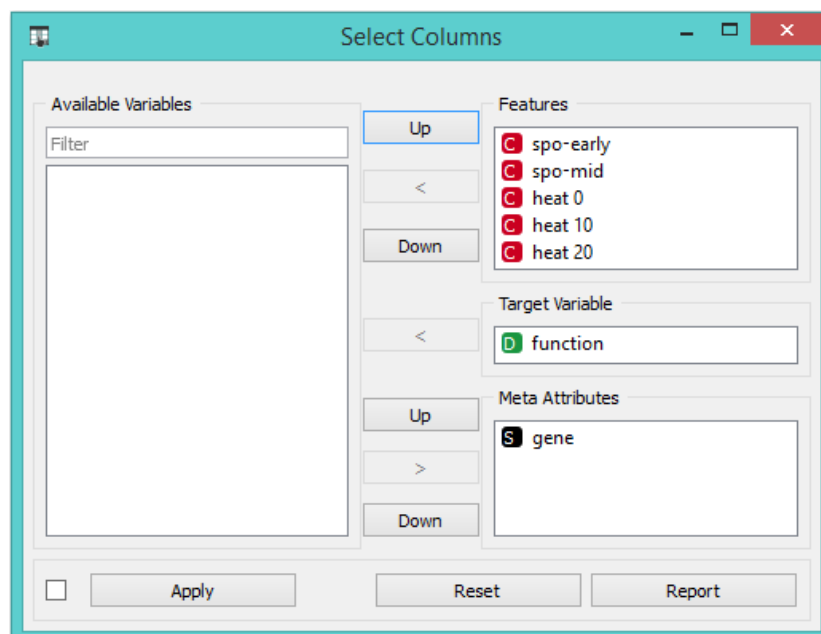
Another way to set the data role is to feed the data to the *Select Columns* widget:



Opening *Select Columns* reveals Orange's classification of attributes. We would like all of our continuous attributes to be data features, gene function to be our target variable and gene names considered as meta attributes. We can obtain this by dragging the attribute names around the boxes in **Select Columns**:



To correctly reassign attribute types, drag attribute named *function* to a **Class** box, and attribute named *gene* to a **Meta Attribute** box. The *Select Columns* widget should now look like this:



Change of attribute types in *Select Columns* widget should be confirmed by clicking the **Apply** button. The data from this widget is fed into *Data Table* that now renders the data just the way we intended:

	spo-early	spo-mid	heat 0	heat 10	heat 20	function	gene
1	0.301	0.546	?	-0.009	0.024	Proteas	YDR427W
2	0.208	?	-0.061	-0.039	0.003	Proteas	YGL048C
3	-0.179	-0.219	-0.097	?	-0.011	Resp	YBR039W
4	-0.085	-0.161	-0.061	-0.265	-0.419	Ribo	YKL180W
5	-0.216	-0.253	-0.228	-0.168	-0.228	Ribo	YHR021C
6	0.017	0.070	0.058	0.286	0.205	Resp	YDR178W
7	0.115	?	0.033	0.262	0.054	Resp	YLL041C
8	0.005	-0.023	-0.038	0.222	0.088	Resp	YOR065W

We could also define the domain for this data set in a different way. Say, we could make the data set ready for regression, and use *heat 0* as a continuous class variable, keep gene function and name as meta variables, and remove *heat 10* and *heat 20* from the data set:

Available Variables

Filter

heat 20

heat 10

Up

Down

Features

spo-early

spo-mid

Target Variable

heat 0

Meta Attributes

gene

function

Apply

Reset

Report

By setting the attributes as above, the rendering of the data in the Data Table widget gives the following output:

	spo-early	spo-mid	heat 0	gene	function	
1	0.301	0.546	?	YDR427W	Proteas	
2	0.208	?	-0.061	YGL048C	Proteas	
3	-0.179	-0.219	-0.097	YBR039W	Resp	
4	-0.085	-0.161	-0.061	YKL180W	Ribo	
5	-0.216	-0.253	-0.228	YHR021C	Ribo	
6	0.017	0.070	0.058	YDR178W	Resp	
7	0.115	?	0.033	YLL041C	Resp	
8	0.005	-0.023	-0.038	YOR065W	Resp	

## Header with Attribute Type Information

Consider again the `sample.xlsx` data set. This time we will augment the names of the attributes with prefixes that define attribute type (continuous, discrete, time, string) and role (class or meta attribute). Prefixes are separated from the attribute name with a hash sign (“#”). Prefixes for attribute roles are:

- c: class attribute
- m: meta attribute
- i: ignore the attribute
- w: instance weights

and for the type:

- C: Continuous
- D: Discrete
- T: Time
- S: String

This is how the header with augmented attribute names looks like in Excel (`sample-head.xlsx`):



	A	B	C	D	E	F	G	H
	mD#function	mS#gene	spo-early	spo-mid	c#heat 0	i#heat 10	i#heat 20	
2	Proteas	YDR427W	0.301	0.546		-0.009	0.024	
3	Proteas	YGL048C	0.208		-0.061	-0.039	0.003	
4	Resp	YBR039W	-0.179	-0.219	-0.097		-0.011	
5	Ribo	YKL180W	-0.085	-0.161	-0.061	-0.265	-0.419	
6	Ribo	YHR021C	-0.216	-0.253	-0.228	-0.168	-0.228	
7	Resp	YDR178W	0.017	0.07	0.058	0.286	0.205	
8	Resp	YLL041C	0.115		0.033	0.262	0.054	
9	Resp	YOR065W	0.005	-0.023	-0.038	0.222	0.088	

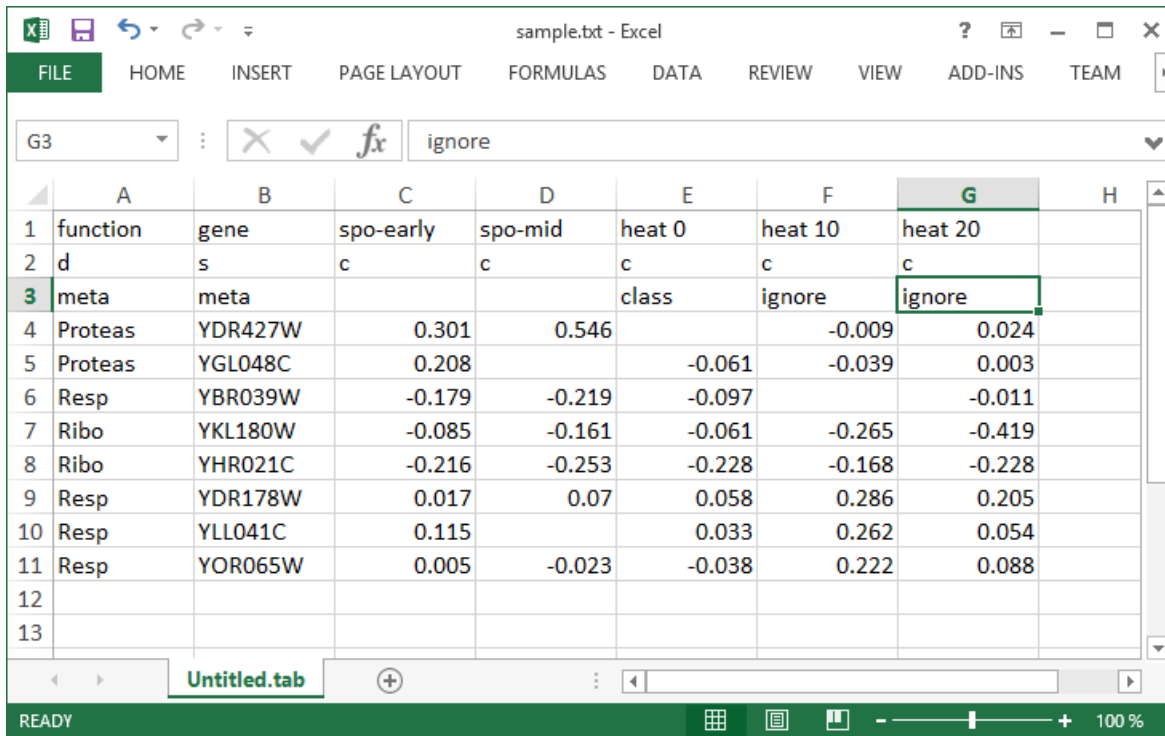
We can again use a **File** widget to load this data set and then render it in the **Data Table**:

	spo-early	spo-mid	heat 0	function	gene
1	0.301	0.546	?	Proteas	YDR427W
2	0.208	?	-0.061	Proteas	YGL048C
3	-0.179	-0.219	-0.097	Resp	YBR039W
4	-0.085	-0.161	-0.061	Ribo	YKL180W
5	-0.216	-0.253	-0.228	Ribo	YHR021C
6	0.017	0.070	0.058	Resp	YDR178W
7	0.115	?	0.033	Resp	YLL041C
8	0.005	-0.023	-0.038	Resp	YOR065W

Notice that the attributes we have ignored (label “i” in the attribute name) are not present in the data set.

## Three-Row Header Format

Orange’s legacy native data format is a tab-delimited text file with three header rows. The first row lists the attribute names, the second row defines their type (continuous, discrete, time and string, or abbreviated c, d, t, and s), and the third row an optional role (class, meta, weight, or ignore). Here is an example:



sample.txt - Excel

FILE HOME INSERT PAGE LAYOUT FORMULAS DATA REVIEW VIEW ADD-INS TEAM

G3 :

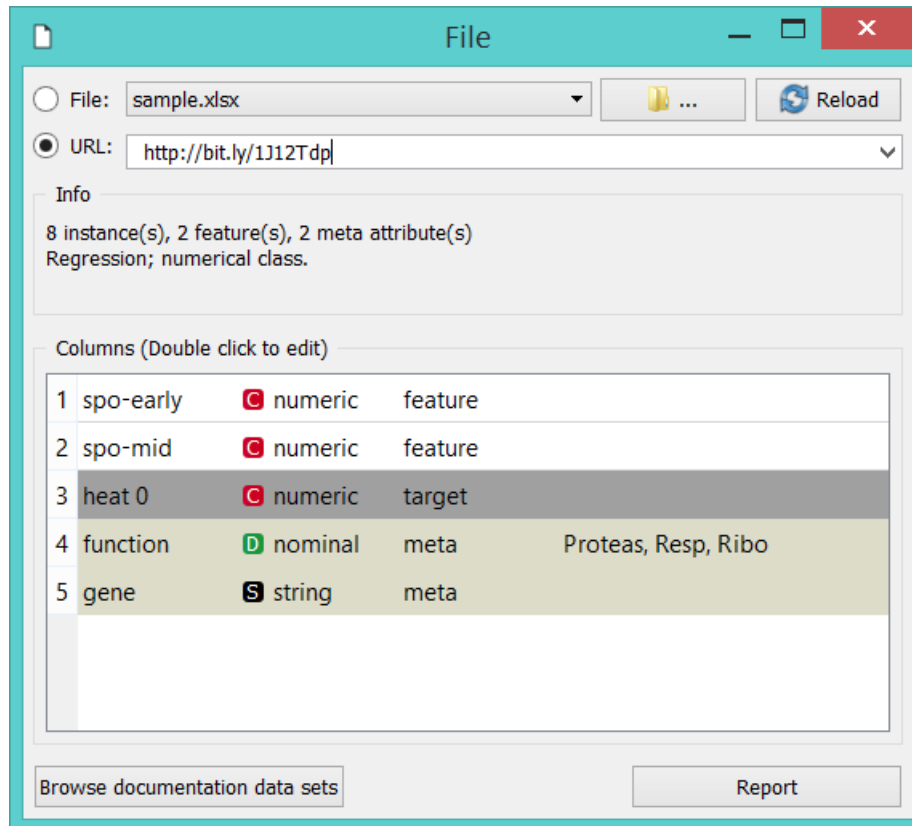
	A	B	C	D	E	F	G	H
1	function	gene	spo-early	spo-mid	heat 0	heat 10	heat 20	
2	d	s	c	c	c	c	c	
3	meta	meta			class	ignore	ignore	
4	Proteas	YDR427W	0.301	0.546		-0.009	0.024	
5	Proteas	YGL048C	0.208		-0.061	-0.039	0.003	
6	Resp	YBR039W	-0.179	-0.219	-0.097		-0.011	
7	Ribo	YKL180W	-0.085	-0.161	-0.061	-0.265	-0.419	
8	Ribo	YHR021C	-0.216	-0.253	-0.228	-0.168	-0.228	
9	Resp	YDR178W	0.017	0.07	0.058	0.286	0.205	
10	Resp	YLL041C	0.115		0.033	0.262	0.054	
11	Resp	YOR065W	0.005	-0.023	-0.038	0.222	0.088	
12								
13								

Untitled.tab

READY

## Data from Google Sheets

Orange can read data from Google Sheets, as long as it conforms to the data presentation rules we have presented above. In Google Sheets, copy the shareable link (Share button, then Get shareable link) and paste it in the *Data File / URL* box of the File widget. For a taste, here's one such link you can use: <http://bit.ly/1J12Tdp>, and the way we have entered it in the **File** widget:



## Data from LibreOffice

If you are using LibreOffice, simply save your files in Excel (.xlsx or .xls) format (available from the drop-down menu under *Save As Type*).

File name:	sample1.xlsx
Save as type:	Excel Workbook (*.xlsx)

## Datetime Format

To avoid ambiguity, Orange supports date and/or time formatted in one of [ISO 8601](#) formats. E.g., the following values are all valid:

```
2016
2016-12-27
2016-12-27 14:20:51+02:00
16:20
```



## Data

### File



Reads attribute-value data from an input file.

### Signals

#### Inputs:

- (None)

#### Outputs:

- **Data**

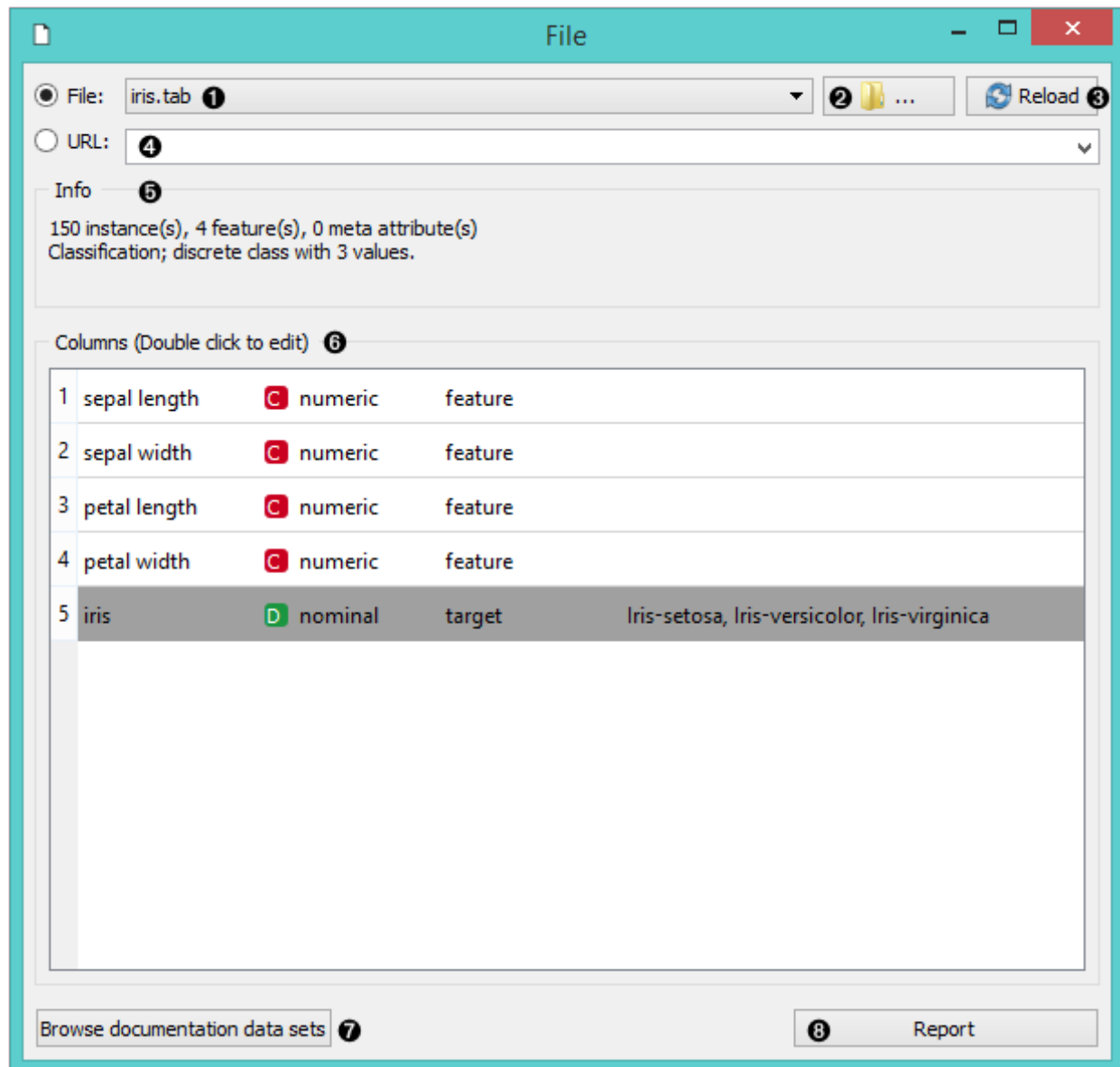
Attribute-valued data from the input file

### Description

The **File** widget *reads the input data file* (data table with data instances) and sends the data set to its output channel. The history of most recently opened files is maintained in the widget. The widget also includes a directory with sample data sets that come pre-installed with Orange.

The widget reads data from Excel (**.xlsx**), simple tab-delimited (**.txt**), comma-separated files (**.csv**) or URLs.

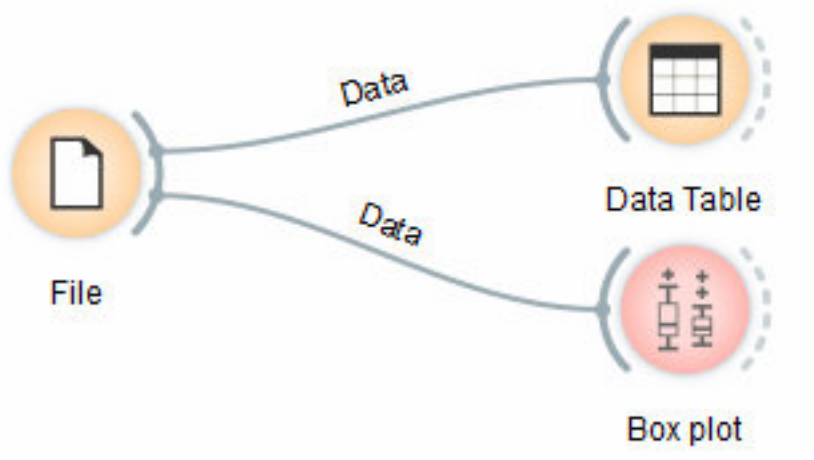
1. Browse through previously opened data files, or load any of the sample ones.
2. Browse for a data file.



3. Reloads currently selected data file.
4. Insert data from URL addresses, including data from Google Sheets.
5. Information on the loaded data set: data set size, number and types of data features.
6. Additional information on the features in the data set. Features can be edited by double-clicking on them. The user can change the attribute names, select the type of variable per each attribute (*Continuous*, *Nominal*, *String*, *Datetime*), and choose how to further define the attributes (as *Features*, *Targets* or *Meta*). The user can also decide to ignore an attribute.
7. Browse documentation data sets.
8. Produce a report.

### Example

Most Orange workflows would probably start with the **File** widget. In the schema below, the widget is used to read the data that is sent to both the *Data Table* and the *Box Plot* widget.



### Loading your data

- Orange can import any comma, .xlsx or tab-delimited data file or URL. Use the *File widget* and then, if needed, select class and meta attributes.
- To specify the domain and the type of the attribute, attribute names can be preceded with a label followed by a hash. Use c for class and m for meta attribute, i to ignore a column, and C, D, S for continuous, discrete and string attribute types. Examples: C#mpg, mS#name, i#dummy. Make sure to set **Import Options** in *File* widget and set the header to **Orange simplified header**.
- Orange's native format is a tab-delimited text file with three header rows. The first row contains attribute names, the second the type (**continuous**, **discrete** or **string**), and the third the optional element (**class**, **meta** or **string**).

Read more on loading your data [here](#).

### SQL Table

Reads data from an SQL database.

sample-head.xlsx - Excel

FILE HOME INSERT PAGE LAYOUT FORMULAS DATA REVIEW VIEW ADD-INS TEAM

A1 : mD#function

	A	B	C	D	E	F	G	H
1	mD#function	mS#gene	spo-early	spo-mid	c#heat 0	i#heat 10	i#heat 20	
2	Proteas	YDR427W	0.301	0.546		-0.009	0.024	
3	Proteas	YGL048C	0.208		-0.061	-0.039	0.003	
4	Resp	YBR039W	-0.179	-0.219	-0.097		-0.011	
5	Ribo	YKL180W	-0.085	-0.161	-0.061	-0.265	-0.419	
6	Ribo	YHR021C	-0.216	-0.253	-0.228	-0.168	-0.228	
7	Resp	YDR178W	0.017	0.07	0.058	0.286	0.205	
8	Resp	YLL041C	0.115		0.033	0.262	0.054	
9	Resp	YOR065W	0.005	-0.023	-0.038	0.222	0.088	
10								
11								
12								
13								

Untitled.tab

READY

100 %





## Signals

### Inputs:

- (None)

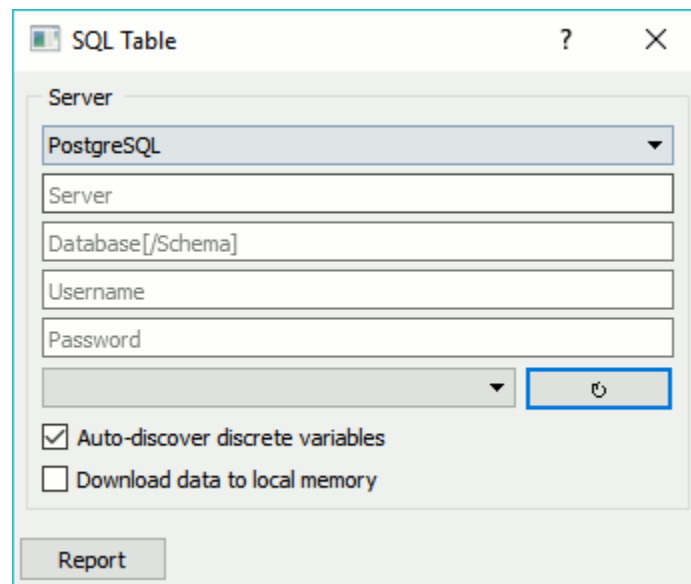
### Outputs:

- **Data**

Attribute-valued data from the database

## Description

The **SQL** widget accesses data stored in an SQL database. It can connect to PostgreSQL (requires `psycopg2` module) or **SQL Server** (requires `pymssql` module).



## Save Data



Saves data to a file.

## Signals

### Inputs:

- **Data**

A data set.

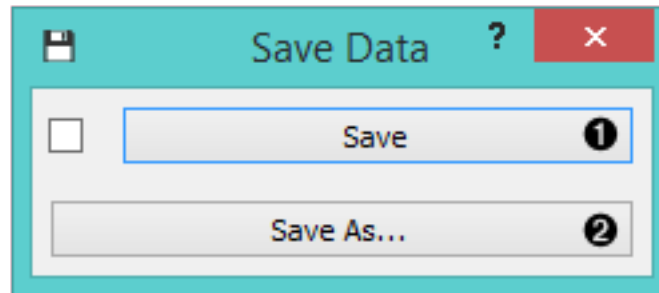
### Outputs:

- (None)

## Description

The **Save Data** widget considers a data set provided in the input channel and saves it to a data file with a specified name. It can save the data as a tab-delimited or a comma-separated file.

The widget does not save the data every time it receives a new signal in the input as this would constantly (and, mostly, inadvertently) overwrite the file. Instead, the data is saved only after a new file name is set or the user pushes the *Save* button.



1. Save by overwriting the existing file.
2. *Save as* to create a new file.

## Example

In the workflow below, we used the *Zoo* data set. We loaded the data into the *Scatter Plot* widget, with which we selected a subset of data instances and pushed them to the **Save Data** widget to store them in a file.

## Data Info

Displays information on a selected data set.

## Signals

### Inputs:

- **Data**  
A data set.
- **Selected Data**  
A data subset.

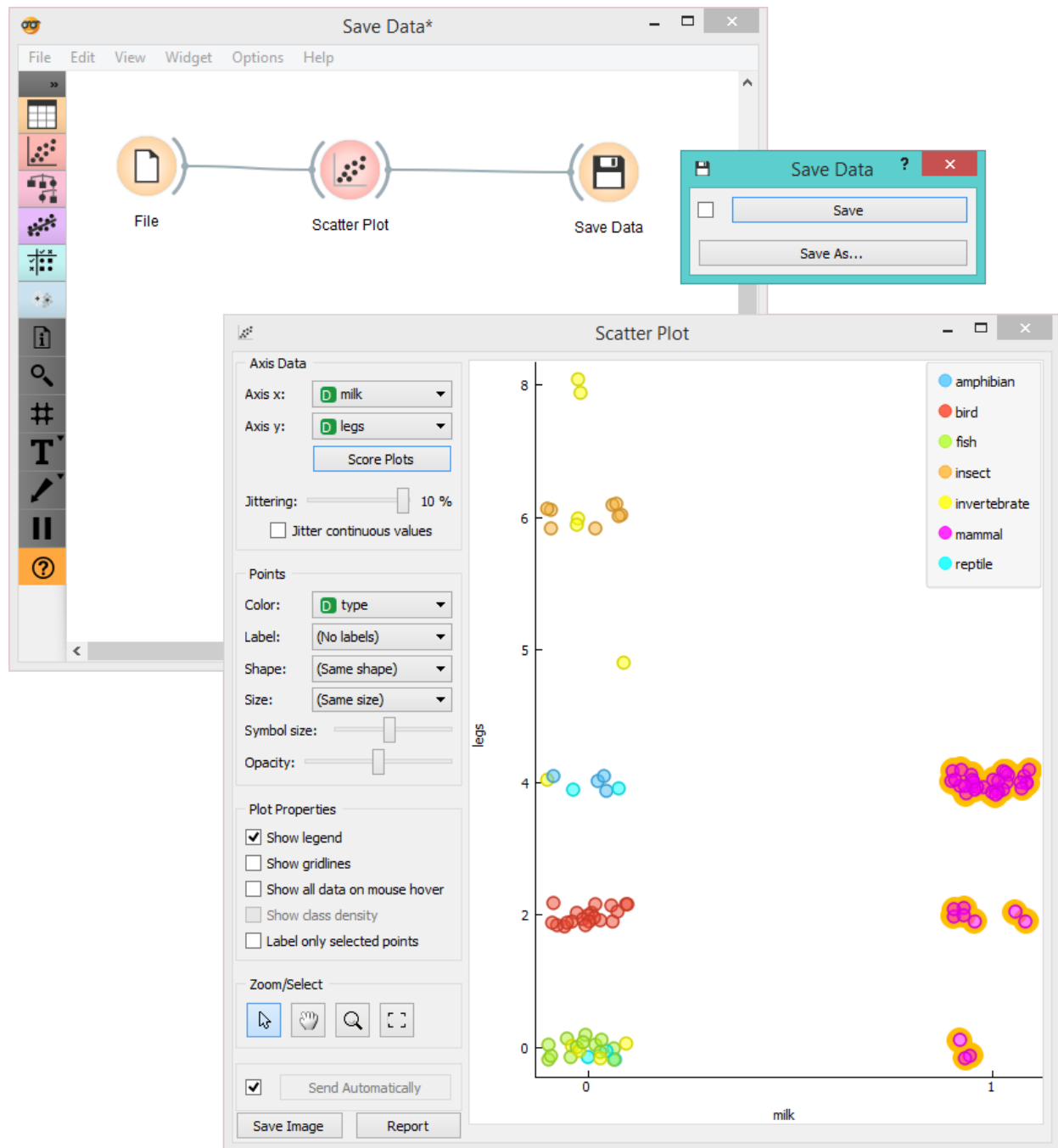
### Outputs:

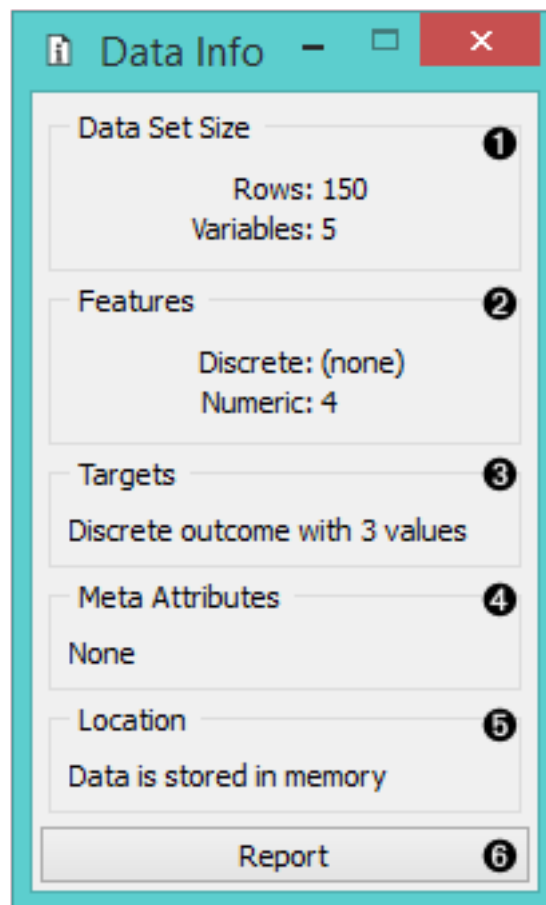
- (None)

## Description

A simple widget that presents information on data set size, features, targets, meta attributes, and location.

1. Information on data set size
2. Information on discrete and continuous features

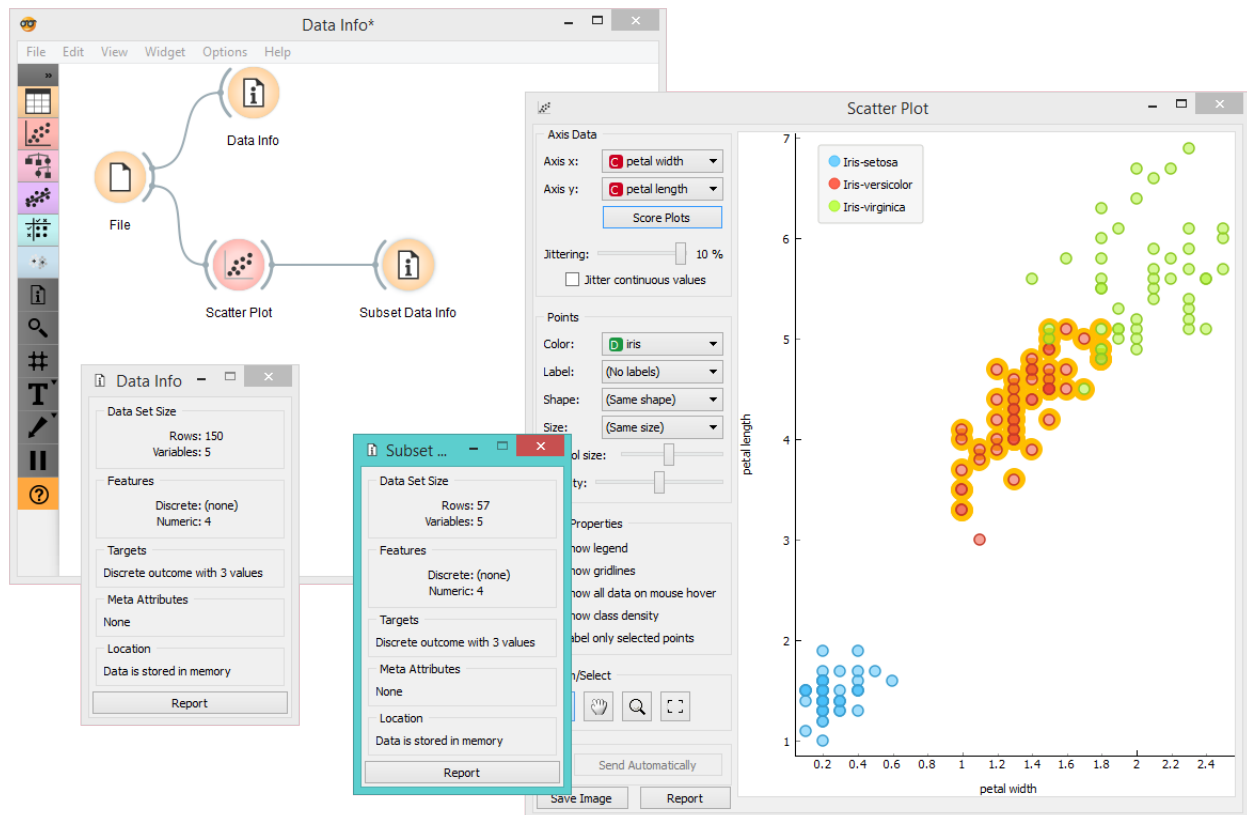




3. Information on targets
4. Information on meta attributes
5. Information on where the data is stored
6. Produce a report.

## Example

Below, we compare the basic statistics of two **Data Info** widgets - one with information on the entire data set and the other with information on the (manually) selected subset from the *Scatterplot* widget. We used the *Iris* data set.



## Data Table



Displays attribute-value data in a spreadsheet.

## Signals

**Inputs:**

- **Data**

Attribute-valued data set.

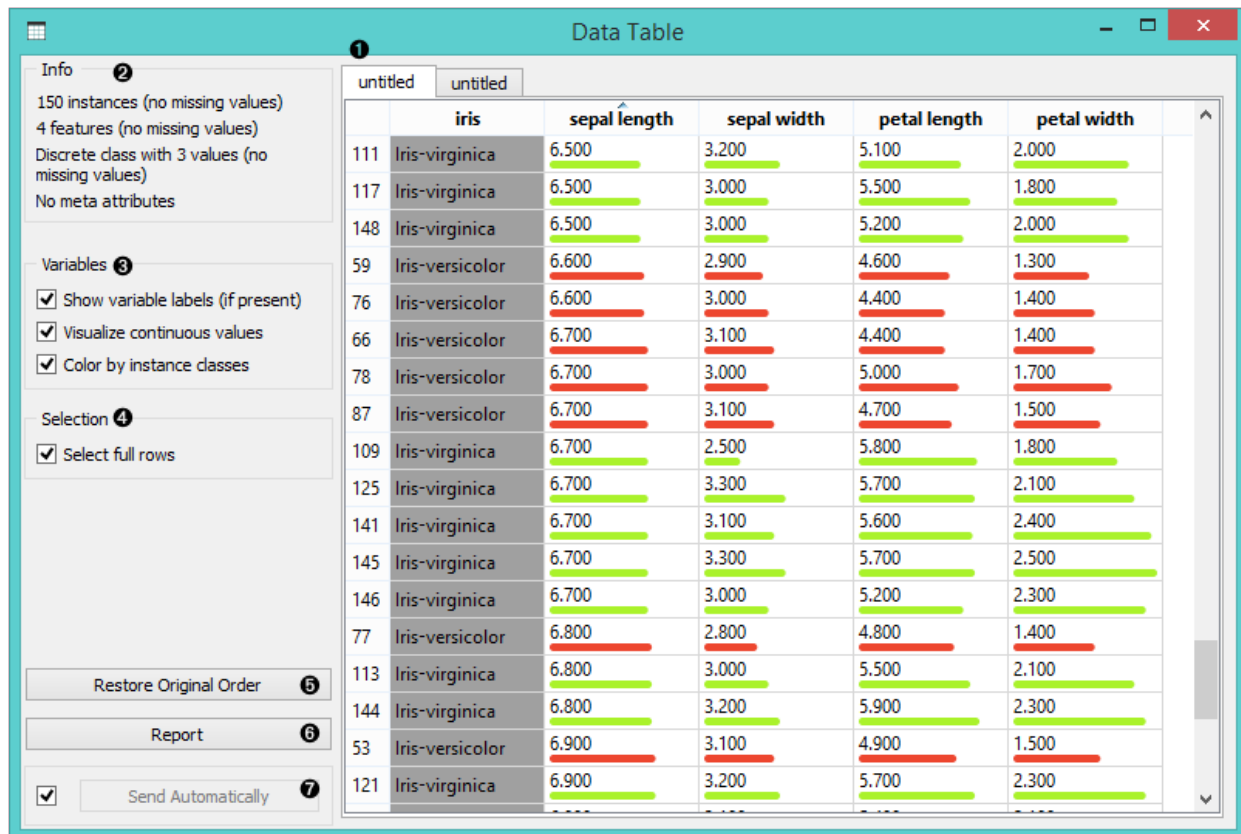
**Outputs:**

- **Selected Data**

Selected data instances.

**Description**

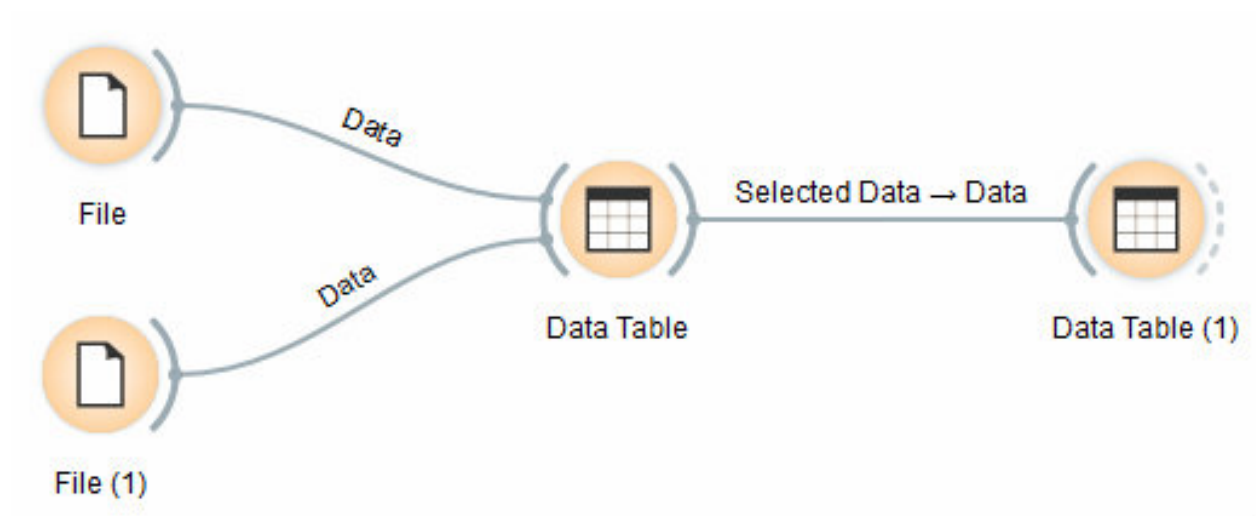
The **Data Table** widget receives one or more data sets in its input and presents them as a spreadsheet. Data instances may be sorted by attribute values. The widget also supports manual selection of data instances.



1. The name of the data set (usually the input data file). Data instances are in rows and their attribute values in columns. In this example, the data set is sorted by the attribute “sepal length”.
2. Info on current data set size and number and types of attributes
3. Values of continuous attributes can be visualized with bars; colors can be attributed to different classes.
4. Data instances (rows) can be selected and sent to the widget’s output channel.
5. Use the *Restore Original Order* button to reorder data instances after attribute-based sorting.
6. Produce a report.
7. While auto-send is on, all changes will be automatically communicated to other widgets. Otherwise, press *Send Selected Rows*.

## Example

We used two *File* widgets to read the *Iris* and *Glass* data set (provided in Orange distribution), and send them to the **Data Table** widget.



Selected data instances in the first **Data Table** are passed to the second **Data Table**. Notice that we can select which data set to view (iris or glass). Changing from one data set to another alters the communicated selection of data instances if *Commit on any change* is selected.

## Select Columns

Manual selection of data attributes and composition of data domain.

## Signals

### Inputs:

- **Data**  
Attribute-valued data set.

### Outputs:

- **Data**  
Attribute-valued data set composed using the domain specification from the widget.

## Description

The **Select Columns** widget is used to manually compose your **data domain**. The user can decide which attributes will be used and how. Orange distinguishes between ordinary attributes, (optional) class attributes and meta attributes. For instance, for building a classification model, the domain would be composed of a set of attributes and a discrete class attribute. Meta attributes are not used in modelling, but several widgets can use them as instance labels.

Orange attributes have a type and are either discrete, continuous or a character string. The attribute type is marked with a symbol appearing before the name of the attribute (D, C, S, respectively).

1. Left-out data attributes that will not be in the output data file

The screenshot displays the Orange Visual Programming interface with the Data Table widget and its associated windows.

**Data Table (1) Widget:** The widget is connected to two File widgets (Iris and Glass) and another Data Table widget. The Data Table (1) window shows the following data:

	y	Ri	Na	Mg	Al	
1	5	1.520	13.380	0.000		
2	5	1.521	12.850	1.610		
3	5	1.521	12.970	0.330		
4	6	1.519	14.000	2.390		
5	6	1.519	13.790	2.410		
6	6	1.518	14.460	2.240		
7	6	1.519	14.090	2.190		
8	6	1.513	14.400	1.740	1.540	74.550
9	6	1.519	14.990	0.780	1.740	72.500
10	6	1.519	14.150	0.000	2.090	72.740
11	6	1.520	14.560	0.000	0.560	73.480
12	6	1.511	17.380	0.000	0.340	75.410
13	7	1.511	13.690	3.200	1.810	72.810
14	7	1.518	14.320	3.260	2.220	71.250
15	7	1.523	13.440	3.340	1.230	72.380

**Data Table Info Window:** The window shows the following information:

- 214 instances (no missing values)
- 9 features (no missing values)
- Discrete class with 6 values (no missing values)
- No meta attributes

**Data Table Variables Window:** The window shows the following variables:

- ☒ Show variable labels (if present)
- ☒ Visualize continuous values
- ☒ Color by instance classes

**Data Table Selection Window:** The window shows the following selection options:

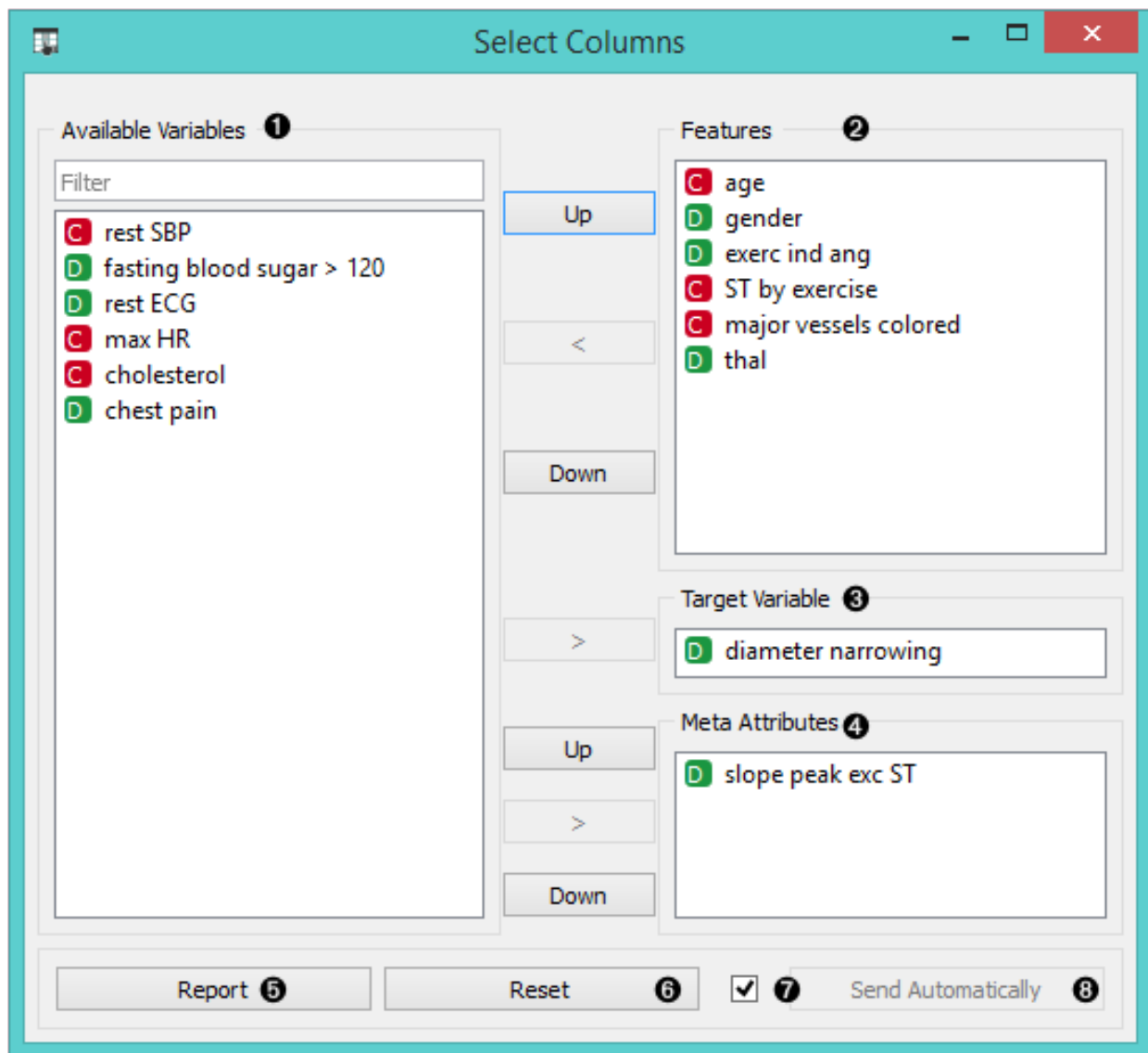
- ☒ Select full rows

**Data Table Table Window:** The window shows the following data:

	y	Ri	Na	Mg	Al	
172	5	1.513	13.020	0.000	3.040	70.480
173	5	1.513	13.000	0.000	3.020	70.700
174	5	1.520	13.380	0.000	1.400	72.250
175	5	1.521	12.850	1.610	2.170	72.180
176	5	1.521	12.870	0.330	1.510	73.390
177	6	1.519	14.000	2.390	1.560	72.370
178	6	1.519	13.790	2.410	1.190	72.760
179	6	1.518	14.460	2.240	1.620	72.380
180	6	1.519	14.090	2.190	1.660	72.670
181	6	1.513	14.400	1.740	1.540	74.550
182	6	1.519	14.990	0.780	1.740	72.500
183	6	1.519	14.150	0.000	2.090	72.740
184	6	1.520	14.560	0.000	0.560	73.480
185	6	1.511	17.380	0.000	0.340	75.410
186	7	1.511	13.690	3.200	1.810	72.810
187	7	1.518	14.320	3.260	2.220	71.250
188	7	1.523	13.440	3.340	1.230	72.380







2. Data attributes in the new data file
3. Target variable. If none, the new data set will be without a target variable.
4. Meta attributes of the new data file. These attributes are included in the data set but are, for most methods, not considered in the analysis.
5. Produce a report.
6. Reset the domain composition to that of the input data file.
7. Tick if you wish to auto-apply changes of the data domain.
8. Apply changes of the data domain and send the new data file to the output channel of the widget.

## Examples

In the workflow below, the *Iris* data from the [File](#) widget is fed into the **Select Columns** widget, where we select to output only two attributes (namely petal width and petal length). We view both the original data set and the data set with selected columns in the [Data Table](#) widget.

The screenshot displays the Orange Visual Programming interface with a workflow consisting of three widgets: **File**, **Select Columns**, and **Data Table (1)**. The **Select Columns** widget is configured to select 'petal length' and 'petal width' as features, with 'iris' as the target variable. The **Data Table (1)** widget shows the resulting data set with 150 instances and 2 features.

**Data Table (1) Info:**

- 150 instances (no missing values)
- 2 features (no missing values)
- Discrete class with 3 values (no missing values)
- No meta attributes

**Data Table (1) Variables:**

- ☒ Show variable labels (if present)
- ☒ Visualize continuous values
- ☒ Color by instance classes

**Data Table (1) Selection:**

- ☒ Select full rows

**Data Table (1) Data:**

	iris	petal length	petal width
1	Iris-setosa	1.400	0.200
2	Iris-setosa	1.400	0.200
3	Iris-setosa	1.300	0.200
4	Iris-setosa	1.500	0.200
5	Iris-setosa	1.400	0.200
6	Iris-setosa	1.700	0.400
7	Iris-setosa	1.400	0.300
8	Iris-setosa	1.500	0.200
9	Iris-setosa	1.400	0.200
10	Iris-setosa	1.500	0.100
11	Iris-setosa	1.500	0.200
12	Iris-setosa	1.600	0.200
13	Iris-setosa	1.100	0.100
14	Iris-setosa	1.200	0.200
15	Iris-setosa	1.500	0.400
16	Iris-setosa	1.300	0.400
17	Iris-setosa	1.400	0.300
18	Iris-setosa	1.400	0.300
19	Iris-setosa	1.700	0.300

**Select Columns Info:**

- 150 instances (no missing values)
- 4 features (no missing values)
- Discrete class with 3 values (no missing values)
- No meta attributes

**Select Columns Variables:**

- ☒ Show variable labels (if present)
- ☒ Visualize continuous values
- ☒ Color by instance classes

**Select Columns Selection:**

- ☒ Select full rows

**Select Columns Data:**

	iris	sepal length	sepal width	petal length	petal width
1	Iris-setosa	5.100	3.500	1.400	0.200
2	Iris-setosa	4.900	3.000	1.400	0.200
3	Iris-setosa	4.700	3.200	1.300	0.200
4	Iris-setosa	4.600	3.100	1.500	0.200
5	Iris-setosa	5.000	3.600	1.400	0.200
6	Iris-setosa	5.400	3.900	1.700	0.400
7	Iris-setosa	4.600	3.400	1.400	0.300
8	Iris-setosa	5.000	3.400	1.500	0.200
9	Iris-setosa	4.400	2.900	1.400	0.200
10	Iris-setosa	4.900	3.100	1.500	0.100
11	Iris-setosa	5.400	3.700	1.500	0.200
12	Iris-setosa	4.800	3.400	1.600	0.200
13	Iris-setosa	4.800	3.000	1.400	0.100
14	Iris-setosa	4.300	3.000	1.100	0.100
15	Iris-setosa	5.800	4.000	1.200	0.200
16	Iris-setosa	5.700	4.400	1.500	0.400
17	Iris-setosa	5.400	3.900	1.300	0.400

**Select Columns Configuration:**

- Available Variables:** sepal length, sepal width
- Features:** petal length, petal width
- Target Variable:** iris
- Meta Attributes:**

**Select Columns Buttons:** Report, Reset, Send Automatically (checked)

For a more complex use of the widget, we composed a workflow to redefine the classification problem in the *heart-disease* data set. Originally, the task was to predict if the patient has a coronary artery diameter narrowing. We changed the problem to that of gender classification, based on age, chest pain and cholesterol level, and informatively kept the diameter narrowing as a meta attribute.

**Select Columns\***

File Edit View Widget Options Help

Naive Bayes  
Classification Tree  
Random Forest Classification

**Confusion Matrix**

Learners  
Naive Bayes  
Classification Tree  
Random Forest Classification

Show  
Number of instances

Select  
Select Correct  
Select Misclassified  
Clear Selection

Output  
☒ Predictions  
☐ Probabilities  
☒ Send Automatically

**Confusion Matrix**

		Predicted		Σ
		female	male	
Actual	female	10	87	97
	male	8	198	206
Σ		18	285	303

**Data Table**

Info  
303 instances (no missing values)  
3 features (no missing values)  
Discrete class with 2 values (no missing values)  
1 meta attribute (no missing values)

Variables  
☒ Show variable labels (if present)  
☒ Visualize continuous values  
☒ Color by instance classes

Selection  
☒ Select full rows

Restore Original Order  
Report  
☒ Send Automatically

	gender	diameter narrowing	age	chest pain
1	male	0	63.000	typical ang
2	male	1	67.000	asymptomatic
3	male	1	67.000	asymptomatic
4	male	0	37.000	non-anginal
5	female	0	41.000	atypical ang
6	male	0	56.000	atypical ang
7	female	1	62.000	asymptomatic
8	female	0	57.000	asymptomatic
9	male	1	63.000	asymptomatic
10	male	1	53.000	asymptomatic
11	male	0	57.000	asymptomatic
12	female	0	56.000	atypical ang
13	male	1	56.000	non-anginal
14	male	0	44.000	atypical ang
15	male	0	52.000	non-anginal
16	male	0	57.000	non-anginal
17	male	1	48.000	atypical ang
18	male	0	54.000	asymptomatic

**Select Columns**

Available Variables  
Filter  
rest SBP  
fasting blood sugar > 120  
rest ECG  
max HR  
slope peak exc ST  
exerc ind ang  
ST by exercise  
major vessels colored  
thal

Features  
age  
chest pain  
cholesterol

Target Variable  
gender

Meta Attributes  
diameter narrowing

Report Reset ☒ Send Automatically



## Select Rows

Selects data instances based on conditions over data features.

### Signals

#### Inputs:

- **Data**  
Data set.

#### Outputs:

- **Matching Data**  
Instances that match the conditions.
- **Non-Matching Data**  
Instances that do not match the conditions.

### Description

This widget selects a subset from an input data set, based on user-defined conditions. Instances that match the selection rule are placed in the output *Matching Data* channel.

Criteria for data selection are presented as a collection of conjuncted terms (i.e. selected items are those matching all the terms in ‘*Conditions*’).

Condition terms are defined through selecting an attribute, selecting an operator from a list of operators, and, if needed, defining the value to be used in the condition term. Operators are different for discrete, continuous and string attributes.

1. Conditions you want to apply, their operators and related values
2. Add a new condition to the list of conditions.
3. Add all the possible variables at once.
4. Remove all the listed variables at once.
5. Information on the input data set and information on instances that match the condition(s)
6. Purge the output data.
7. When the *Send automatically* box is ticked, all changes will be automatically communicated to other widgets.
8. Produce a report.

Any change in the composition of the condition will update the information pane (*Data Out*).

If *Send automatically* is selected, then the output is updated on any change in the composition of the condition or any of its terms.

### Example

In the workflow below, we used the *Zoo* data from the *File* widget and fed it into the **Select Rows** widget. In the widget, we chose to output only two animal types, namely fish and reptiles. We can inspect both the original data set and the data set with selected rows in the *Data Table* widget.

**Select Rows**

**Conditions** ①

<b>D</b> fuel-type	is not	gas
<b>D</b> make	is	toyota
<b>C</b> price	is below	25000

② Add Condition    ③ Add All Variables    ④ Remove All

**Data** ⑤

In: ~205 rows, 26 variables  
Out: ~3 rows, 13 variables

**Purging** ⑥

☒ Remove unused features  
☒ Remove unused classes

⑦ Report    ☒ Send automatically ⑧    Send

The screenshot displays the Orange Visual Programming interface with three windows open:

- Select Rows<sup>4</sup>**: A workflow diagram showing a 'File' widget connected to a 'Data Table' widget, which is then connected to a 'Select Rows' widget. The 'Data Table (selected)' widget is also shown.
- Select Rows**: A configuration window for the 'Select Rows' widget. The 'Conditions' section shows a condition: 'type is one of fish, reptile'. The 'Data' section shows 'In: ~101 rows, 18 variables' and 'Out: ~18 rows, 12 variables'. The 'Purging' section has checkboxes for 'Remove unused features' and 'Remove unused classes', both of which are checked. The 'Send automatically' checkbox is also checked.
- Data Table**: A window showing the original dataset with 18 rows and 8 columns: type, name, hair, feathers, eggs, milk, and an unnamed column. The rows are numbered 1 to 18.
- Data Table (selected)**: A window showing the filtered dataset with 18 rows and 7 columns: type, name, eggs, aquatic, predator, and toothed. The rows are numbered 1 to 18.

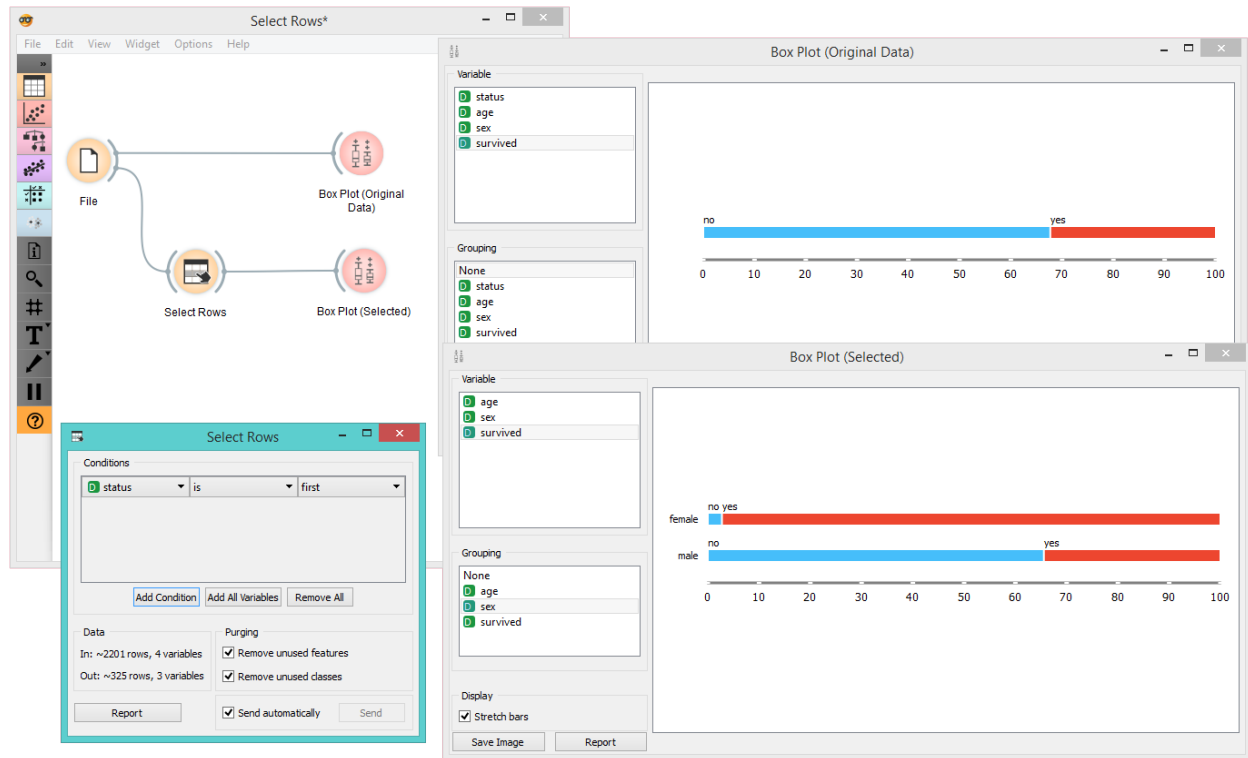
The 'Data Table' window contains the following data:

	type	name	hair	feathers	eggs	milk
1	mammal	aardvark	1	0	0	1
2	mammal	antelope	1	0	0	1
3	fish	bass	0	0	1	0
4	mammal	bear	1	0	0	1
5	mammal	boar	1	0	0	1
6	mammal	buffalo	1	0	0	1
7	mammal	calf	1	0	0	1
8	fish	carp	0	0	1	0
9	fish	catfish	0	0	1	0
10	mammal	cavy	1	0	0	1
11	mammal	cheetah	1	0	0	1
12	bird	chicken	0	1	1	0
13	fish	chub	0	0	1	0
14	invertebrate	clam	0	0	1	0
15	invertebrate	crab	0	0	1	0
16	invertebrate	crayfish	0	0	1	0
17	bird	crow	0	1	1	0
18	mammal	deer	1	0	0	1

The 'Data Table (selected)' window contains the following data:

	type	name	eggs	aquatic	predator	toothed
1	fish	bass	1	1	1	1
2	fish	carp	1	1	0	1
3	fish	catfish	1	1	1	1
4	fish	chub	1	1	1	1
5	fish	dogfish	1	1	1	1
6	fish	haddock	1	1	0	1
7	fish	herring	1	1	1	1
8	fish	pike	1	1	1	1
9	fish	piranha	1	1	1	1
10	reptile	pitviper	1	0	1	1
11	fish	seahorse	1	1	0	1
12	reptile	seasnake	0	1	1	1
13	reptile	slowworm	1	0	1	1
14	fish	sole	1	1	0	1
15	fish	stingray	1	1	1	1
16	reptile	tortoise	1	0	0	0
17	reptile	tuatara	1	0	1	1
18	fish	tuna	1	1	1	1

In the next example, we used the data from the *Titanic* data set and similarly fed it into the *Box Plot* widget. We first observed the entire data set based on survival. Then we selected only first class passengers in the **Select Rows** widget and fed it again into the *Box Plot*. There we could see all the first class passengers listed by their survival rate and grouped by gender.



## Data Sampler



Selects a subset of data instances from an input data set.

### Signals

#### Inputs:

- **Data**  
Input data set to be sampled.

#### Outputs:

- **Data Sample**  
A set of sampled data instances.
- **Remaining Data**  
All other data instances from the input data set, which are not included in the sample.

## Description

The **Data Sampler** widget implements several means of sampling data from an input channel. It outputs a sampled and a complementary data set (with instances from the input set that are not included in the sampled data set). The output is processed after the input data set is provided and *Sample Data* is pressed.

1. Information on the input and output data set
2. The desired sampling method:
  - **Fixed proportion of data** returns a selected percentage of the entire data (e.g. 70% of all the data)
  - **Fixed sample size** returns a selected number of data instances with a chance to set *Sample with replacement*, which always samples from the entire data set (does not subtract instances already in the subset)
  - **Cross Validation** partitions data instances into complementary subsets, where you can select the number of folds (subsets) and which fold you want to use as a sample.
3. *Replicable sampling* maintains sampling patterns that can be carried across users, while *stratification* mimics the composition of the input data set.
4. Produce a report.
5. Press *Sample data* to output the data sample.

## Examples

First, let's see how the **Data Sampler** works. Let's look at the information on the original data set in the *Data Info* widget. We see there are 24 instances in the data (we used *lenses.tab*). We sampled the data with the **Data Sampler** widget and we chose to go with a fixed sample size of 5 instances for simplicity. We can observe the sampled data in the *Data Table* widget. The second *Data Table* shows the remaining 19 instances that weren't in the sample.

In the workflow below, we have sampled 10 data instances from the *Iris* data set and sent the original data and the sample to *Scatter Plot* widget for exploratory data analysis. The sampled data instances are plotted with filled circles, while the original data set is represented with empty circles.

## Transpose

Transposes a data table.

## Signals

### Inputs:

- **Data**  
A data set.

### Outputs:

- **Data**  
Transposed data set

## Description

**Transpose** widget transposes data table.



**Data Sampler** ? ×

**1 Information**

150 instances in input data set.  
Outputting 105 instances.

**2 Sampling Type**

☒ Fixed proportion of data:

70 %

☐ Fixed sample size

Instances: 10

☐ Sample with replacement

☐ Cross validation

Number of folds: 10

Selected fold: 1

☐ Bootstrap

**3 Options**

☐ Replicable (deterministic) sampling

☐ Stratify sample (when possible)

**4 Report** **5 Sample Data**

The screenshot displays the Orange Visual Programming interface with a workflow involving the **Data Sampler** widget. The workflow starts with a **File** widget connected to **Data Info**, which then connects to **Data Sampler**. The **Data Sampler** widget is connected to two **Data Table** widgets: **Data Table (Sample Data)** and **Data Table (Remaining Data)**.

The **Data Sampler** widget's configuration dialog is open, showing the following settings:

- Information:** 24 instances in input data set. Outputting 5 instances.
- Sampling Type:**
  - ☒ Fixed sample size: Instances: 5
  - ☐ Fixed proportion of data: 70 %
  - ☐ Cross validation: Number of folds: 10, Selected fold: 1
  - ☐ Bootstrap
- Options:**
  - ☐ Replicable (deterministic) sampling
  - ☐ Stratify sample (when possible)

The **Data Table (Sample Data)** widget displays the following data:

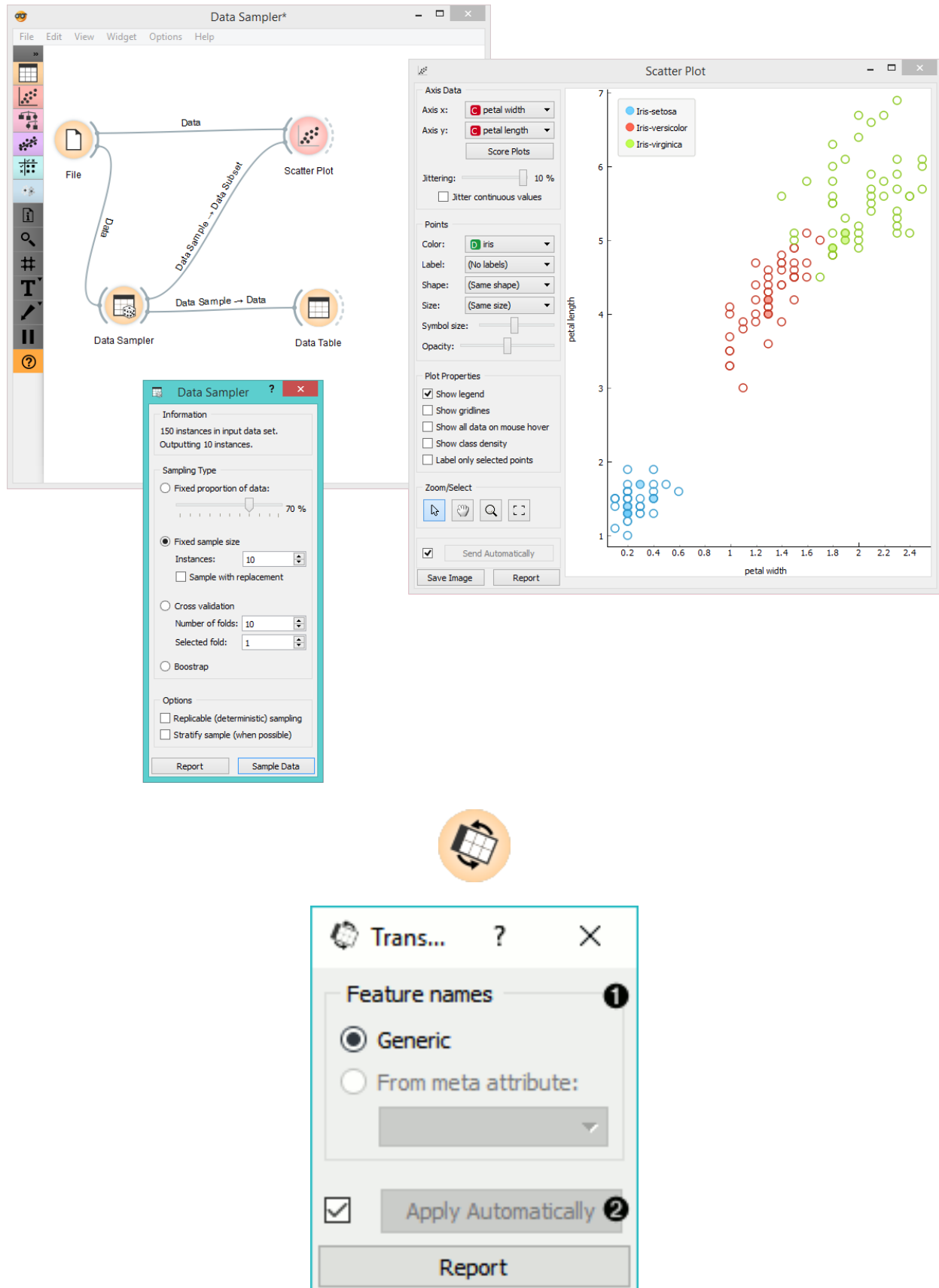
	lenses	age	prescription	astigmatic	tear_rate
1	hard	presbyopic	myope	yes	normal
2	hard	young	myope	yes	normal
3	none	presbyopic	myope	yes	reduced
4	none	presbyopic	hypermetrope	no	reduced
5	none	pre-presbyopic	hypermetrope	no	reduced

The **Data Table (Remaining Data)** widget displays the following data:

	lenses	age	prescription	astigmatic	tear_rate
1	none	presbyopic	myope	no	normal
2	none	presbyopic	myope	no	reduced
3	soft	young	hypermetrope	no	normal
4	hard	young	hypermetrope	yes	normal
5	none	presbyopic	hypermetrope	yes	reduced
6	none	young	myope	no	reduced
7	soft	young	myope	no	normal
8	soft	pre-presbyopic	myope	no	normal
9	hard	pre-presbyopic	myope	yes	normal
10	soft	presbyopic	hypermetrope	no	normal
11	none	presbyopic	hypermetrope	yes	normal
12	none	pre-presbyopic	myope	no	reduced
13	none	young	hypermetrope	yes	reduced
14	soft	pre-presbyopic	hypermetrope	no	normal
15	none	pre-presbyopic	hypermetrope	yes	normal
16	none	young	myope	yes	reduced
17	none	young	hypermetrope	no	reduced
18	none	pre-presbyopic	myope	yes	reduced
19	none	pre-presbyopic	hypermetrope	yes	reduced

The **Data Info** widget shows the following information:

- Data Set Size:** Rows: 24, Variables: 5
- Features:** Discrete: 4, Numeric: (none)
- Targets:** Discrete outcome with 3 values
- Meta Attributes:** None
- Location:** Data is stored in memory



## Example

This is a simple workflow showing how to use **Transpose**. Connect the widget to *File* widget. The output of **Transpose** is a transposed data table with rows as columns and columns as rows. You can observe the result in a *Data Table*.

The screenshot shows the Orange Visual Programming interface with a workflow consisting of three widgets: **File**, **Transpose**, and **Data Table**. The **Data Table** widget is open, showing the following data:

	Feature name	Feature 001 Iris-setosa	Feature 002 Iris-setosa	Feature 003 Iris-setosa	Feature 004 Iris-setosa
1	sepal length	5.100	4.900	4.700	4.600
2	sepal width	3.500	3.000	3.200	3.100
3	petal length	1.400	1.400	1.300	1.500
4	petal width	0.200	0.200	0.200	0.200

The **Data Table** widget also displays the following information:

- Info:** 4 instances (no missing values), 150 features (no missing values), No target variable, 1 meta attribute (no missing values).
- Variables:**
  - ☒ Show variable labels (if present)
  - ☐ Visualize continuous values
  - ☒ Color by instance classes
- Selection:**
  - ☒ Select full rows
- Buttons:** Restore Original Order, Report, Send Automatically (checked).

## Discretize



Discretizes continuous attributes from an input data set.

### Signals

#### Inputs:

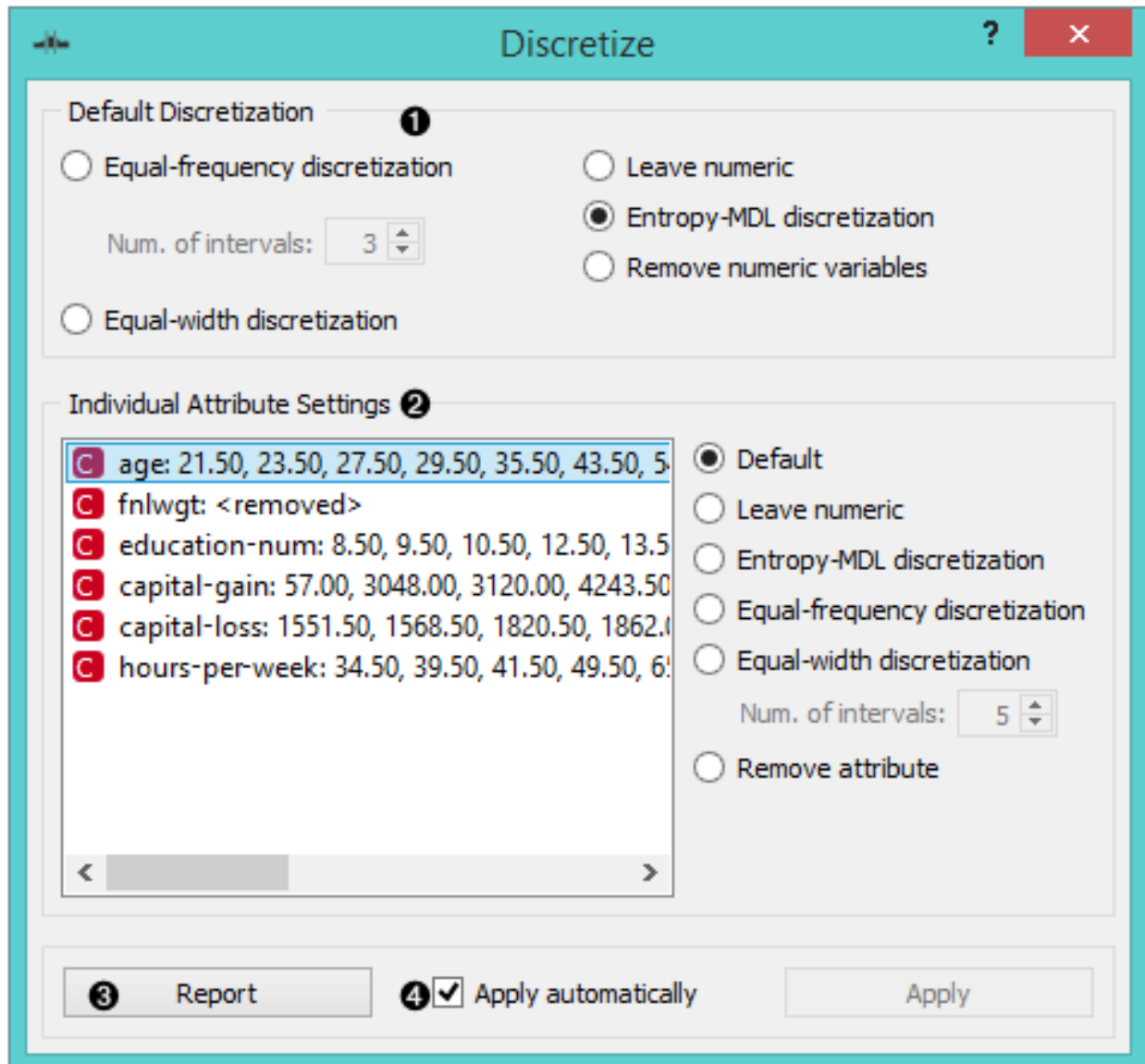
- **Data**  
Attribute-valued data set.

#### Outputs:

- **Data**  
A data set with discretized values.

## Description

The **Discretize** widget discretizes continuous attributes with a selected method.



1. The basic version of the widget is rather simple. It allows choosing between three different discretizations.

- **Entropy-MDL**, invented by Fayyad and Irani is a top-down discretization, which recursively splits the attribute at a cut maximizing information gain, until the gain is lower than the minimal description length of the cut. This discretization can result in an arbitrary number of intervals, including a single interval, in which case the attribute is discarded as useless (removed).
- **Equal-frequency** splits the attribute into a given number of intervals, so that they each contain approximately the same number of instances.
- **Equal-width** evenly splits the range between the smallest and the largest observed value. The *Number of intervals* can be set manually.
- The widget can also be set to leave the attributes continuous or to remove them.

- To treat attributes individually, go to **Individual Attribute Settings**. They show a specific discretization of each attribute and allow changes. First, the top left list shows the cut-off points for each attribute. In the snapshot, we used the entropy-MDL discretization, which determines the optimal number of intervals automatically; we can see it discretized the age into seven intervals with cut-offs at 21.50, 23.50, 27.50, 35.50, 43.50, 54.50 and 61.50, respectively, while the capital-gain got split into many intervals with several cut-offs. The final weight (fnlwgt), for instance, was left with a single interval and thus removed.

On the right, we can select a specific discretization method for each attribute. Attribute “*fnlwgt*” would be removed by the MDL-based discretization, so to prevent its removal, we select the attribute and choose, for instance, **Equal-frequency discretization**. We could also choose to leave the attribute continuous.

- Produce a report.
- Tick *Apply automatically* for the widget to automatically commit changes. Alternatively, press *Apply*.

## Example

In the schema below, we show the *Iris* data set with continuous attributes (as in the original data file) and with discretized attributes.

The screenshot displays the Orange Visual Programming interface. A workflow is shown with a 'File' widget connected to a 'Data' widget, which then connects to a 'Discretize' widget. The 'Discretize' widget is configured with 'Entropy-MDL discretization' and 'Num. of intervals: 3'. The 'Individual Attribute Settings' panel shows the following cut-off points for the Iris dataset attributes:

- sepal length: 5.55, 6.15
- sepal width: 2.95, 3.35
- petal length: 2.45, 4.75
- petal width: 0.80, 1.75

The 'Data Table' widget shows the original data with continuous attributes. The 'Data Table (1)' widget shows the data after discretization, where continuous attributes are converted into discrete intervals.

	iris	sepal length	sepal width	petal length	petal width
1	Iris-setosa	5.100	3.500	1.400	0.200
2	Iris-setosa	4.900	3.000	1.400	0.200
3	Iris-setosa	4.700	3.200	1.300	0.200
4	Iris-setosa	4.600	3.100	1.500	0.200
5	Iris-setosa	5.000	3.600	1.400	0.200
6	Iris-setosa	5.400	3.900	1.700	0.400
7	Iris-setosa	4.600	3.400	1.400	0.300
8	Iris-setosa	5.000	3.400	1.500	0.200
9	Iris-setosa	4.400	2.900	1.400	0.200
10	Iris-setosa	4.900	3.100	1.500	0.100
11	Iris-setosa	5.400	3.700	1.500	0.200
12	Iris-setosa	4.800	3.400	1.600	0.200
13	Iris-setosa	4.800	3.000	1.400	0.100
14	Iris-setosa	4.300	3.000	1.100	0.100

	iris	sepal length	sepal width	petal length	petal width
1	Iris-setosa	< 5.55	≥ 3.35	< 2.45	< 0.8
2	Iris-setosa	< 5.55	2.95 - 3.35	< 2.45	< 0.8
3	Iris-setosa	< 5.55	2.95 - 3.35	< 2.45	< 0.8
4	Iris-setosa	< 5.55	2.95 - 3.35	< 2.45	< 0.8
5	Iris-setosa	< 5.55	≥ 3.35	< 2.45	< 0.8
6	Iris-setosa	< 5.55	≥ 3.35	< 2.45	< 0.8
7	Iris-setosa	< 5.55	≥ 3.35	< 2.45	< 0.8
8	Iris-setosa	< 5.55	≥ 3.35	< 2.45	< 0.8
9	Iris-setosa	< 5.55	< 2.95	< 2.45	< 0.8
10	Iris-setosa	< 5.55	2.95 - 3.35	< 2.45	< 0.8
11	Iris-setosa	< 5.55	≥ 3.35	< 2.45	< 0.8
12	Iris-setosa	< 5.55	≥ 3.35	< 2.45	< 0.8
13	Iris-setosa	< 5.55	2.95 - 3.35	< 2.45	< 0.8
14	Iris-setosa	< 5.55	2.95 - 3.35	< 2.45	< 0.8

## Continuize

Turns discrete attributes into continuous dummy variables.



## Signals

### Inputs:

- **Data**  
Input data set

### Outputs:

- **Data**  
Output data set

## Description

The **Continuize** widget receives a data set in the input and outputs the same data set in which the discrete attributes (including binary attributes) are replaced with continuous ones.

1. **Continuization methods**, which define the treatment of multivalued discrete attributes. Say that we have a discrete attribute status with the values low, middle and high, listed in that order. Options for their transformation are:
  - **Target or First value as base**: the attribute will be transformed into two continuous attributes, status=middle with values 0 or 1 signifying whether the original attribute had value middle on a particular example, and similarly, status=high. Hence, a three-valued attribute is transformed into two continuous attributes, corresponding to all except the first value of the attribute.
  - **Most frequent value as base**: similar to the above, except that the data is analyzed and the most frequent value is used as a base. So, if most examples have the value middle, the two newly constructed continuous attributes will be status=low and status=high.
  - **One attribute per value**: this would construct three continuous attributes out of a three-valued discrete one.
  - **Ignore multinominal attributes**: removes the multinominal attributes from the data.
  - **Treat as ordinal**: converts the attribute into a continuous attribute with values 0, 1, and 2.
  - **Divide by number of values**: same as above, except that the values are normalized into range 0-1. So, our case would give values 0, 0.5 and 1.
2. Define the treatment of continuous attributes. You will usually prefer the *Leave them as they are* option. The alternative is *Normalize by span*, which will subtract the lowest value found in the data and divide by the span, so all values will fit into [0, 1]. Finally, *Normalize by standard deviation* subtracts the average and divides by the deviation.
3. Define the treatment of class attributes. Besides leaving it as it is, there are also a couple of options available for multinominal attributes, except for those options which split the attribute into more than one attribute - this obviously cannot be supported since you cannot have more than one class attribute.
4. With *value range*, you can define the values of new attributes. In the above text, we supposed the range *from 0 to 1*. You can change it to *from -1 to 1*.
5. Produce a report.

**Continuize** ? ×

**Multinomial Attributes** ①

- ☒ Target or first value as base
- ☐ Most frequent value as base
- ☐ One attribute per value
- ☐ Ignore multinomial attributes
- ☐ Remove all discrete attributes
- ☐ Treat as ordinal
- ☐ Divide by number of values

**Continuous Attributes** ②

- ☒ Leave them as they are
- ☐ Normalize by span
- ☐ Normalize by standard deviation

**Discrete Class Attribute** ③

- ☒ Leave it as it is
- ☐ Treat as ordinal
- ☐ Divide by number of values
- ☐ One class per value

**Value Range** ④

- ☐ From -1 to 1
- ☒ From 0 to 1

**Report** ⑤

☒ ⑥ Apply Automatically



- If *Apply automatically* is ticked, changes are committed automatically. Otherwise, you have to press *Apply* after each change.

## Examples

First, let's see what is the output of the **Continue** widget. We feed the original data (the *Heart disease* data set) into the *Data Table* and see how they look like. Then we continue the discrete values and observe them in another *Data Table*.

The screenshot shows the Orange Visual Programming interface. A workflow is visible in the left sidebar: a **Data** widget feeds into a **Data Table** widget, which then feeds into a **Continue** widget, which finally feeds into another **Data Table** widget labeled "Data Table (Continued)".

The **Continue** widget's settings are shown in the top-right panel. It has the following options:

- Info:** 303 instances, 13 features (0.2% missing values), Discrete class with 2 values (no missing values), No meta attributes.
- Variables:**
  - ☒ Show variable labels (if present)
  - ☒ Visualize continuous values
  - ☒ Color by instance classes
- Selection:**
  - ☒ Select full rows
- Buttons:** Restore Original Order, Report, Send Automatically (checked).

The **Data Table** widget shows the original data with columns: diameter, narrowing, age, gender, chest pain, and rest SBP. The data is displayed in a table with 14 rows.

The **Data Table (Continued)** widget shows the data after the **Continue** operation. The discrete attributes have been converted to continuous ones. For example, the **gender** attribute is now represented by a single column **gender=** with values 1.000 for male and 0.000 for female. The **chest pain** attribute is now represented by two columns: **chest pain=atypical ang** and **chest pain=non-anginal**, with values 1.000 and 0.000 respectively.

In the second example, we show a typical use of this widget - in order to properly plot the linear projection of the data, discrete attributes need to be converted to continuous ones and that is why we put the data through the **Continue** widget before drawing it. The attribute “*chest pain*” originally had four values and was transformed into three continuous attributes; similar happened to gender, which was transformed into a single attribute “*gender=female*”.

## Randomize

Shuffles classes, attributes and/or metas of an input data set.

## Signals

Inputs:



- **Data**

Data set.

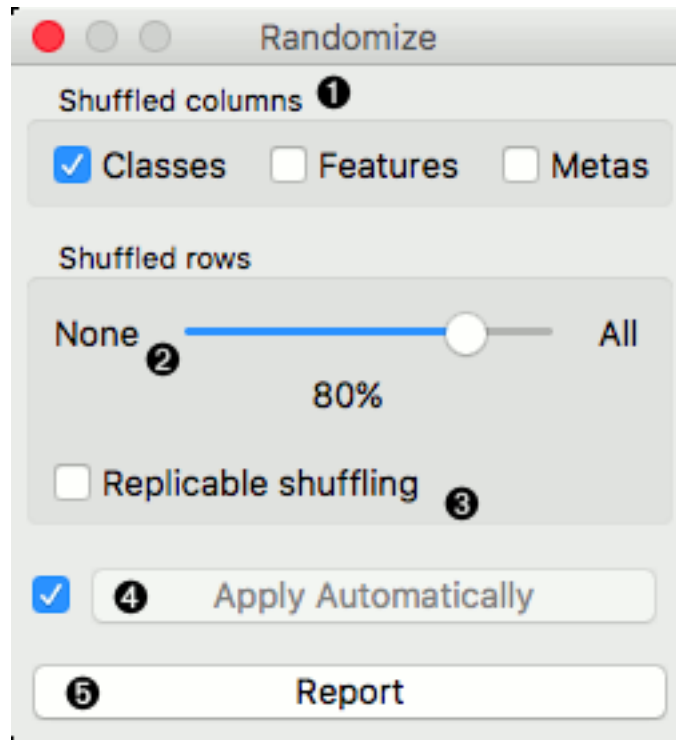
**Outputs:**

- **Data**

Randomized data set.

## Description

The **Randomize** widget receives a data set in the input and outputs the same data set in which the classes, attributes or/and metas are shuffled.



1. Select group of columns of the data set you want to shuffle.
2. Select proportion of the data set you want to shuffle.
3. Produce replicable output.
4. If *Apply automatically* is ticked, changes are committed automatically. Otherwise, you have to press *Apply* after each change.
5. Produce a report.

## Example

The **Randomize** widget is usually placed right after (e.g. *File* widget). The basic usage is shown in the following workflow, where values of class variable of Iris data set are randomly shuffled.

In the next example we show how shuffling class values influences model performance on the same data set as above.

The screenshot shows the Orange Visual Programming interface with a workflow consisting of a File widget, a Data widget, a Randomize widget, and a Data Table widget. The Data Table widget is open, displaying the Iris dataset. The Randomize widget is also open, showing settings for shuffled columns and rows.

**Data Table**

Info

- 150 instances (no missing values)
- 4 features (no missing values)
- Discrete class with 3 values (no missing values)
- No meta attributes

Variables

- ☒ Show variable labels (if present)
- ☐ Visualize continuous values
- ☒ Color by instance classes

Selection

- ☒ Select full rows

Buttons: Restore Original Order, Report, Send Automatically

	iris	sepal length	sepal width	petal length	petal width
1	Iris-setosa	5.100	3.500	1.400	0.200
2	Iris-setosa	4.900	3.000	1.400	0.200
3	Iris-setosa	4.700	3.200	1.300	0.200
4	Iris-setosa	4.600	3.100	1.500	0.200
5	Iris-setosa	5.000	3.600	1.400	0.200
6	Iris-setosa	5.400	3.900	1.700	0.400
7	Iris-setosa	4.600	3.400	1.400	0.300
8	Iris-setosa	5.000	3.400	1.500	0.200
9	Iris-setosa	4.400	2.900	1.400	0.200
10	Iris-setosa	4.900	3.100	1.500	0.100
11	Iris-setosa	5.400	3.700	1.500	0.200
12	Iris-setosa	4.800	3.400	1.600	0.200
13	Iris-setosa	4.800	3.000	1.400	0.100
14	Iris-setosa	4.300	3.000	1.100	0.100
15	Iris-setosa	5.800	4.000	1.200	0.200
16	Iris-setosa	5.700	4.400	1.500	0.400
17	Iris-setosa	5.400	3.900	1.300	0.400
18	Iris-setosa	5.100	3.500	1.400	0.300
19	Iris-setosa	5.700	3.800	1.700	0.300

**Randomize**

Shuffled columns

- ☒ Classes
- ☐ Features
- ☐ Metas

Shuffled rows

None  100% All

- ☒ Replicable shuffling

Buttons: Apply Automatically, Report

**Data Table (1)**

Info

- 150 instances (no missing values)
- 4 features (no missing values)
- Discrete class with 3 values (no missing values)
- No meta attributes

Variables

- ☒ Show variable labels (if present)
- ☐ Visualize continuous values
- ☒ Color by instance classes

Selection

- ☒ Select full rows

Buttons: Restore Original Order, Report, Send Automatically

	iris	sepal length	sepal width	petal length	petal width
1	Iris-setosa	5.100	3.500	1.400	0.200
2	Iris-versicolor	4.900	3.000	1.400	0.200
3	Iris-versicolor	4.700	3.200	1.300	0.200
4	Iris-setosa	4.600	3.100	1.500	0.200
5	Iris-virginica	5.000	3.600	1.400	0.200
6	Iris-versicolor	5.400	3.900	1.700	0.400
7	Iris-virginica	4.600	3.400	1.400	0.300
8	Iris-setosa	5.000	3.400	1.500	0.200
9	Iris-setosa	4.400	2.900	1.400	0.200
10	Iris-virginica	4.900	3.100	1.500	0.100
11	Iris-versicolor	5.400	3.700	1.500	0.200
12	Iris-setosa	4.800	3.400	1.600	0.200
13	Iris-virginica	4.800	3.000	1.400	0.100
14	Iris-versicolor	4.300	3.000	1.100	0.100
15	Iris-versicolor	5.800	4.000	1.200	0.200
16	Iris-setosa	5.700	4.400	1.500	0.400
17	Iris-versicolor	5.400	3.900	1.300	0.400
18	Iris-versicolor	5.100	3.500	1.400	0.300
19	Iris-setosa	5.700	3.800	1.700	0.300

The screenshot shows the Orange Visual Programming interface with a workflow consisting of a File widget, a Data widget, a Randomize widget, a Learner widget, and a Test & Score widget. The Test & Score widget is open, displaying evaluation results for Logistic Regression using cross-validation. The Randomize widget is also open, showing settings for shuffled columns and rows.

**Test & Score**

Sampling

- ☒ Cross validation
  - Number of folds: 10
  - ☒ Stratified
- ☐ Random sampling
  - Repeat train/test: 10
  - Training set size: 66 %
  - ☒ Stratified
  - ☐ Leave one out
  - ☐ Test on train data
  - ☐ Test on test data

Target Class

(Average over classes)

Buttons: Report

**Evaluation Results**

Method	AUC	CA	F1	Precision	Recall
Logistic Regression	0.990	0.960	0.960	0.962	0.960

**Randomize**

Shuffled columns

- ☒ Classes
- ☐ Features
- ☐ Metas

Shuffled rows

None  100% All

- ☒ Replicable shuffling

Buttons: Apply Automatically, Report

**Test & Score (Randomize)**

Sampling

- ☒ Cross validation
  - Number of folds: 10
  - ☒ Stratified
- ☐ Random sampling
  - Repeat train/test: 10
  - Training set size: 66 %
  - ☒ Stratified
  - ☐ Leave one out
  - ☐ Test on train data
  - ☐ Test on test data

Target Class

(Average over classes)

Buttons: Report

**Evaluation Results**

Method	AUC	CA	F1	Precision	Recall
Logistic Regression	0.518	0.347	0.335	0.332	0.347

## Concatenate



Concatenates data from multiple sources.

### Signals

#### Inputs:

- **Primary Data**  
A data set that defines the attribute set.
- **Additional Data**  
An additional data set.

#### Outputs:

- **Data**

### Description

The widget concatenates multiple sets of instances (data sets). The merge is “vertical”, in a sense that two sets of 10 and 5 instances yield a new set of 15 instances.

1. Set the attribute merging method.
2. Add the identification of source data sets to the output data set.
3. Produce a report.
4. If *Apply automatically* is ticked, changes are communicated automatically. Otherwise, click *Apply*.

If one of the tables is connected to the widget as the primary table, the resulting table will contain its own attributes. If there is no primary table, the attributes can be either a union of all attributes that appear in the tables specified as *Additional Tables*, or their intersection, that is, a list of attributes common to all the connected tables.

### Example

As shown below, the widget can be used for merging data from two separate files. Let’s say we have two data sets with the same attributes, one containing instances from the first experiment and the other instances from the second experiment and we wish to join the two data tables together. We use the **Concatenate** widget to merge the data sets by attributes (appending new rows under existing attributes).

Below, we used a modified *Zoo* data set. In the *first File* widget, we loaded only the animals beginning with the letters A and B and in the *second* one only the animals beginning with the letter C. Upon concatenation, we observe the new data in the *Data Table* widget, where we see the complete table with animals from A to C.

### Paint Data

Paints data on a 2D plane. You can place individual data points or use a brush to paint larger data sets.

**Concatenate**

**Domain Merging ①**

When there is no primary table, the domain should be:

- ☒ Union of attributes appearing in all tables
- ☐ Intersection of attributes in all tables

The resulting table will have a class only if there is no conflict between input classes.

**Source Identification ②**

☐ Append data source IDs

Feature name:

Place:

**③ Report** ☒ **④ Apply Automatically**

The screenshot displays the Orange Visual Programming interface with a workflow involving the **Concatenate** widget. The workflow starts with two data sources, **ZOO A+B** and **ZOO C**, which are connected to the **Concatenate** widget. The output of the widget is a **Data Table (A+B+C)**.

The **Concatenate** widget's settings are shown in the bottom right, indicating the **Domain Merging** strategy is set to **Union of attributes appearing in all tables**. The **Source Identification** section shows **Append data source IDs** is unchecked, and the **Place** is set to **Class attribute**.

The **Data Table (A+B)** window shows the following data:

	type	name	hair	feathers	eggs	milk
1	mammal	aardvark	1	0	0	1
2	mammal	antelope	1	0	0	1
3	fish	bass	0	0	1	0
4	mammal	bear	1	0	0	1
5	mammal	boar	1	0	0	1
6	mammal	buffalo	1	0	0	1

The **Data Table (C)** window shows the following data:

	type	name	hair	feathers	eggs	milk
1	mammal	calf	1	0	0	1
2	fish	carp	0	0	1	0
3	fish	catfish	0	0	1	0
4	mammal	cavy	1	0	0	1
5	mammal	cheetah	1	0	0	1
6	bird	chicken	0	1	1	0
7	fish	chub	0	0	1	0
8	invertebrate	clam	0	0	1	0

The **Data Table (A+B+C)** window shows the combined data from both sources:

	type	name	hair	feathers	eggs	milk
1	mammal	aardvark	1	0	0	1
2	mammal	antelope	1	0	0	1
3	fish	bass	0	0	1	0
4	mammal	bear	1	0	0	1
5	mammal	boar	1	0	0	1
6	mammal	buffalo	1	0	0	1
7	mammal	calf	1	0	0	1
8	fish	carp	0	0	1	0
9	fish	catfish	0	0	1	0
10	mammal	cavy	1	0	0	1
11	mammal	cheetah	1	0	0	1
12	bird	chicken	0	1	1	0
13	fish	chub	0	0	1	0
14	invertebrate	clam	0	0	1	0
15	invertebrate	crab	0	0	1	0
16	invertebrate	crayfish	0	0	1	0
17	bird	crow	0	1	1	0



## Signals

### Inputs

- (None)

### Outputs

- **Data**

Attribute-valued data set created in the widget

## Description

The widget supports the creation of a new data set by visually placing data points on a two-dimension plane. Data points can be placed on the plane individually (*Put*) or in a larger number by brushing (*Brush*). Data points can belong to classes if the data is intended to be used in supervised learning.

1. Name the axes and select a class to paint data instances. You can add or remove classes. Use only one class to create classless, unsupervised data sets.
2. Drawing tools. Paint data points with *Brush* (multiple data instances) or *Put* (individual data instance). Select data points with *Select* and remove them with the Delete/Backspace key. Reposition data points with *Jitter* (spread) and *Magnet* (focus). Use *Zoom* and scroll to zoom in or out. Below, set the radius and intensity for Brush, Put, Jitter and Magnet tools.
3. Reset to Input Data.
4. *Save Image* saves the image to your computer in a .svg or .png format.
5. Produce a report.
6. Tick the box on the left to automatically commit changes to other widgets. Alternatively, press *Send* to apply them.

## Example

In the example below, we have painted a data set with 4 classes. Such data set is great for demonstrating k-means and hierarchical clustering methods. In the screenshot, we see that *k-means*, overall, recognizes clusters better than *hierarchical clustering*. It returns a score rank, where the best score (the one with the highest value) means the most likely number of clusters. Hierarchical clustering, however, doesn't group the right classes together. This is a great tool for learning and exploring statistical concepts.

## Python Script

Extends functionalities through Python scripting.

## Signals

### Inputs:

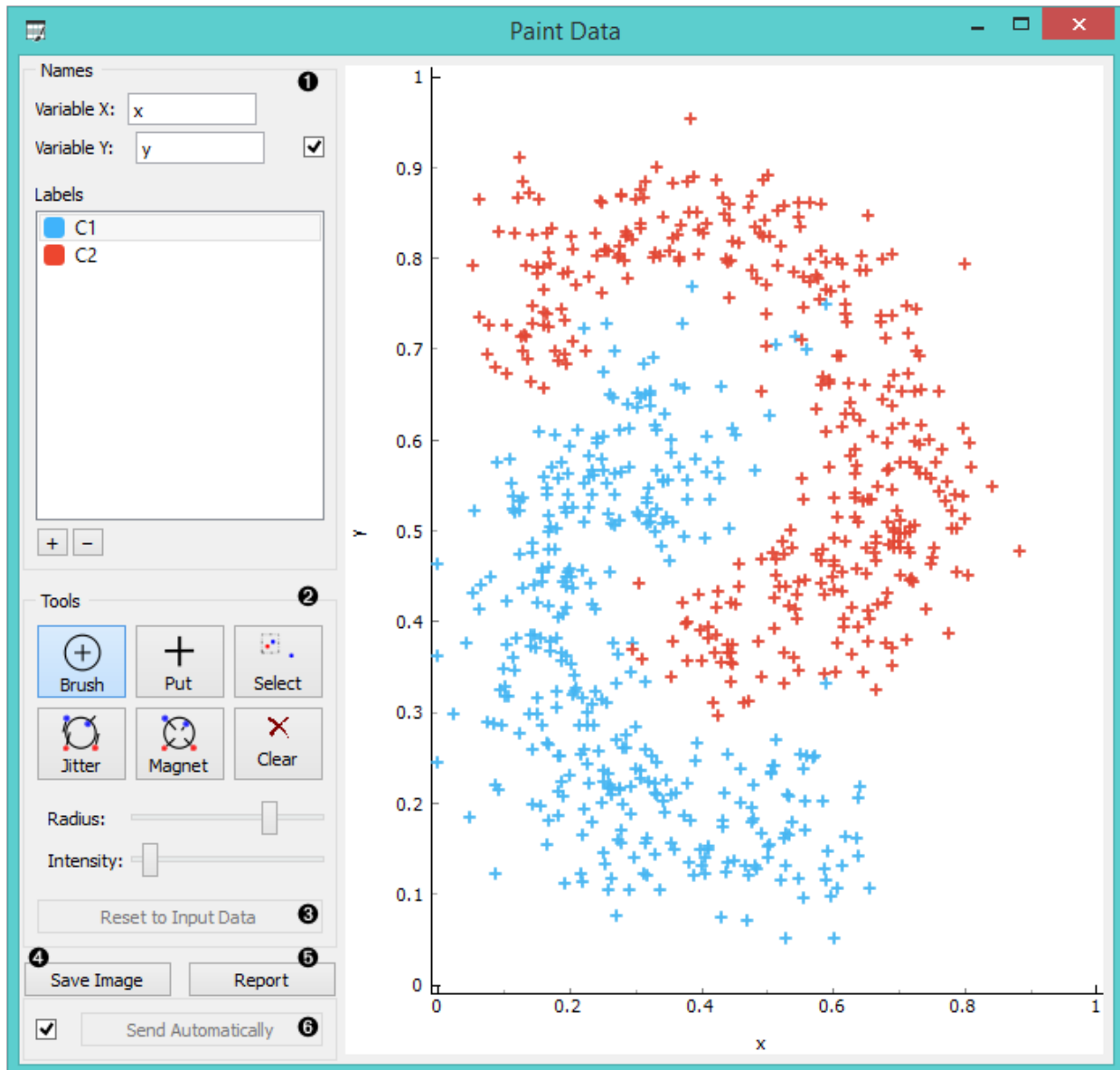
- **in\_data (Orange.data.Table)**

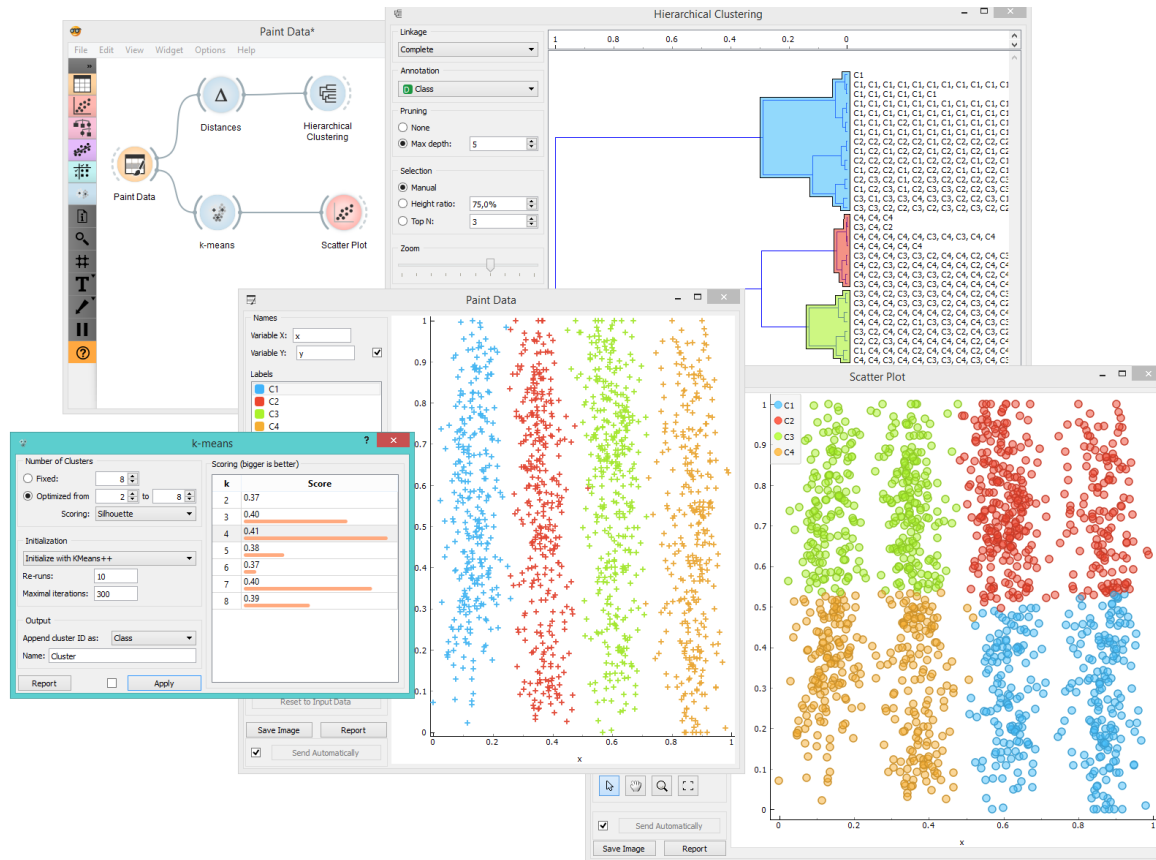
Input data set bound to `in_data` variable in the script's local namespace.

- **in\_distance (Orange.core.SymMatrix)**

Input symmetric matrix bound to `in_distance` variable in the script's local namespace.







- **in\_learner (Orange.classification.Learner)**

Input learner bound to `in_learner` variable in the script's local namespace.

- **in\_classifier (Orange.classification.Learner)**

Input classifier bound to `in_classifier` variable in the script's local namespace.

- **in\_object (object)**

Input python object bound to `in_object` variable in the script's local namespace.

**Outputs:**

- **out\_data (Orange.data.Table)**

Data set retrieved from `out_data` variable in the script's local namespace after execution.

- **out\_distance (Orange.core.SymMatrix)**

Symmetric matrix retrieved from `out_distance` variable in the script's local namespace after execution.

- **out\_learner (Orange.classification.Learner)**

Learner retrieved from `out_learner` variable in the script's local namespace.

- **out\_classifier (Orange.classification.Learner)**

Classifier retrieved from `out_classifier` variable in the script's local namespace after execution.

- **out\_object (object)**

Python object retrieved from `out_object` variable in the script's local namespace after execution.

## Description

**Python Script** widget can be used to run a python script in the input, when a suitable functionality is not implemented in an existing widget. The script has `in_data`, `in_distance`, `in_learner`, `in_classifier` and `in_object` variables (from input signals) in its local namespace. If a signal is not connected or it did not yet receive any data, those variables contain `None`.

After the script is executed, `out_data`, `out_distance`, ... variables from the script's local namespace are extracted and used as outputs of the widget. The widget can be further connected to other widgets for visualizing the output.

For instance the following script would simply pass on all signals it receives:

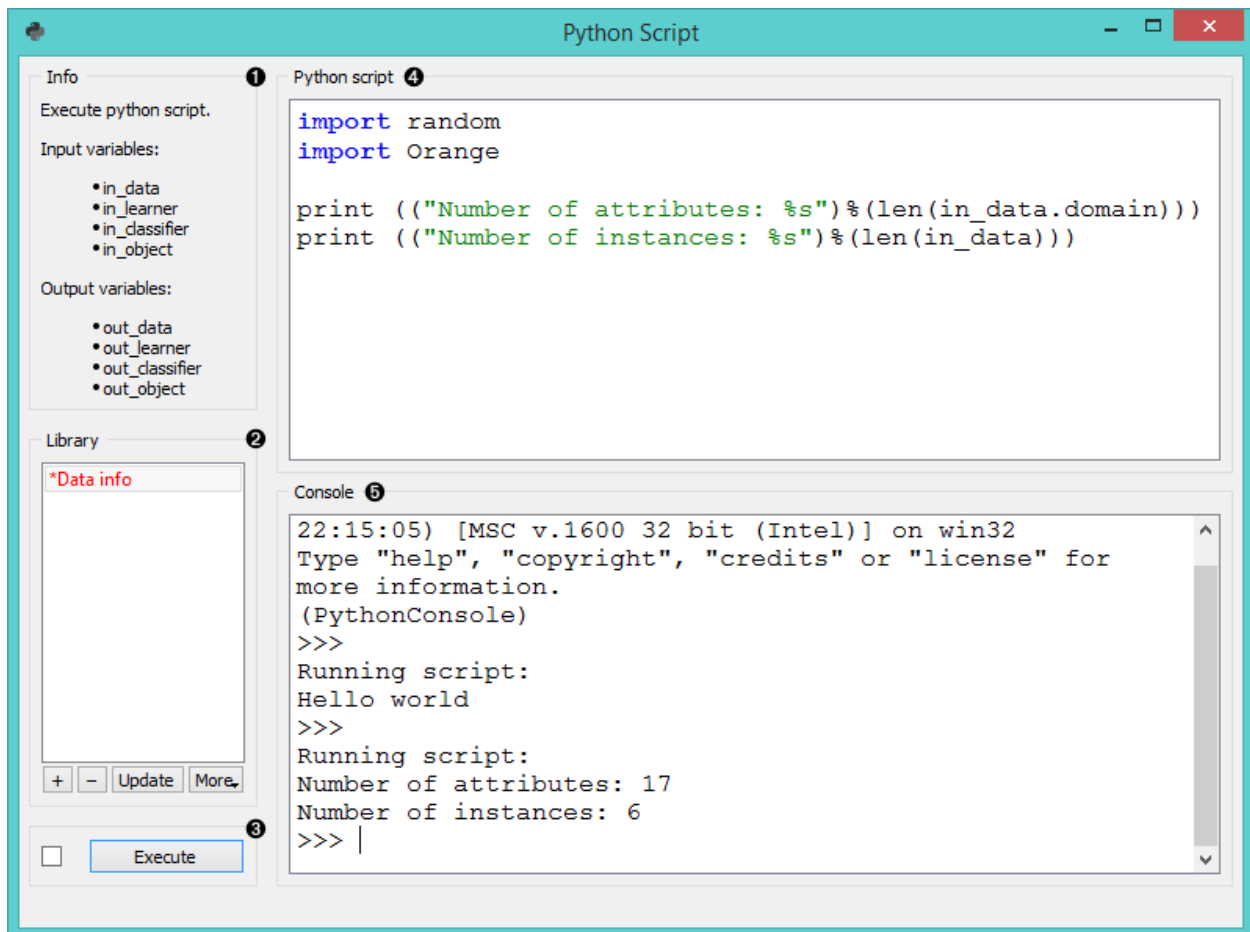
```
out_data = in_data
out_distance = in_distance
out_learner = in_learner
out_classifier = in_classifier
out_object = in_object
```

---

**Note:** You should not modify the input objects in place.

---

1. Info box contains names of basic operators for Orange Python script.
2. The *Library* control can be used to manage multiple scripts. Pressing “+” will add a new entry and open it in the *Python script* editor. When the script is modified, its entry in the *Library* will change to indicate it has unsaved changes. Pressing *Update* will save the script (keyboard shortcut `ctrl + s`). A script can be removed by selecting it and pressing the “-” button.



3. Pressing *Execute* in the *Run* box executes the script (using `exec`). Any script output (from `print`) is captured and displayed in the *Console* below the script. If *Auto execute* is checked, the script is run any time inputs to the widget change.
4. The *Python script* editor on the left can be used to edit a script (it supports some rudimentary syntax highlighting).
5. Console displays the output of the script.

## Examples

Python Script widget is intended to extend functionalities for advanced users.

One can, for example, do batch filtering by attributes. We used `zoo.tab` for the example and we filtered out all the attributes that have more than 5 discrete values. This in our case removed only 'leg' attribute, but imagine an example where one would have many such attributes.

```
from Orange.data import Domain, Table
domain = Domain([attr for attr in in_data.domain.attributes
                  if attr.is_continuous or len(attr.values) <= 5],
                in_data.domain.class_vars)
out_data = Table(domain, in_data)
```

The second example shows how to round all the values in a few lines of code. This time we used `wine.tab` and rounded all the values to whole numbers.

```
import numpy as np
out_data = in_data.copy()
#copy, otherwise input data will be overwritten
np.round(out_data.X, 0, out_data.X)
```

The third example introduces some gaussian noise to the data. Again we make a copy of the input data, then walk through all the values with a double for loop and add random noise.

```
import random
from Orange.data import Domain, Table
new_data = in_data.copy()
for inst in new_data:
    for f in inst.domain.attributes:
        inst[f] += random.gauss(0, 0.02)
out_data = new_data
```

The final example uses Orange3-Text add-on. **Python Script** is very useful for custom preprocessing in text mining, extracting new features from strings, or utilizing advanced nltk or gensim functions. Below, we simply tokenized our input data from `deerwester.tab` by splitting them by whitespace.

```
print('Running Preprocessing ...')
tokens = [doc.split(' ') for doc in in_data.documents]
print('Tokens:', tokens)
out_object = in_data
out_object.store_tokens(tokens)
```

You can add a lot of other preprocessing steps to further adjust the output. The output of **Python Script** can be used with any widget that accepts the type of output your script produces. In this case, connection is green, which signalizes the right type of input for Word Cloud widget.



**Python Script**

Info

Execute python script.

Input variables:

- in\_data
- in\_learner
- in\_classifier
- in\_object

Output variables:

- out\_data
- out\_learner
- out\_classifier
- out\_object

Library

filtering

+ - Update More

☐ Execute

Python Script

```

from Orange.data import Domain, Table

domain = Domain([attr for attr in in_data.domain.attributes
                  if attr.is_continuous or len(attr.values)
                  <= 5], in_data.domain.class_vars)

out_data = Table(domain, in_data)

print(out_data.domain)
  
```

Console

```

[1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1 | mammal],
[1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0 | insect],
[1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1 | mammal],
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0 |
invertebrate],
[0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0 | bird]
>>>
Running script:
[hair, feathers, eggs, milk, airborne, aquatic, predator,
toothed, backbone, breathes, venomous, fins, tail,
domestic, catsize | type]
>>>
  
```



**Python Script**

Info

Execute python script.

Input variables:

- in\_data
- in\_learner
- in\_classifier
- in\_object

Output variables:

- out\_data
- out\_learner
- out\_classifier
- out\_object

Library

round

+ - Update More

☐ Execute

Python Script

```

import numpy as np

out_data = in_data.copy()
#copy, otherwise input data will be overwritten

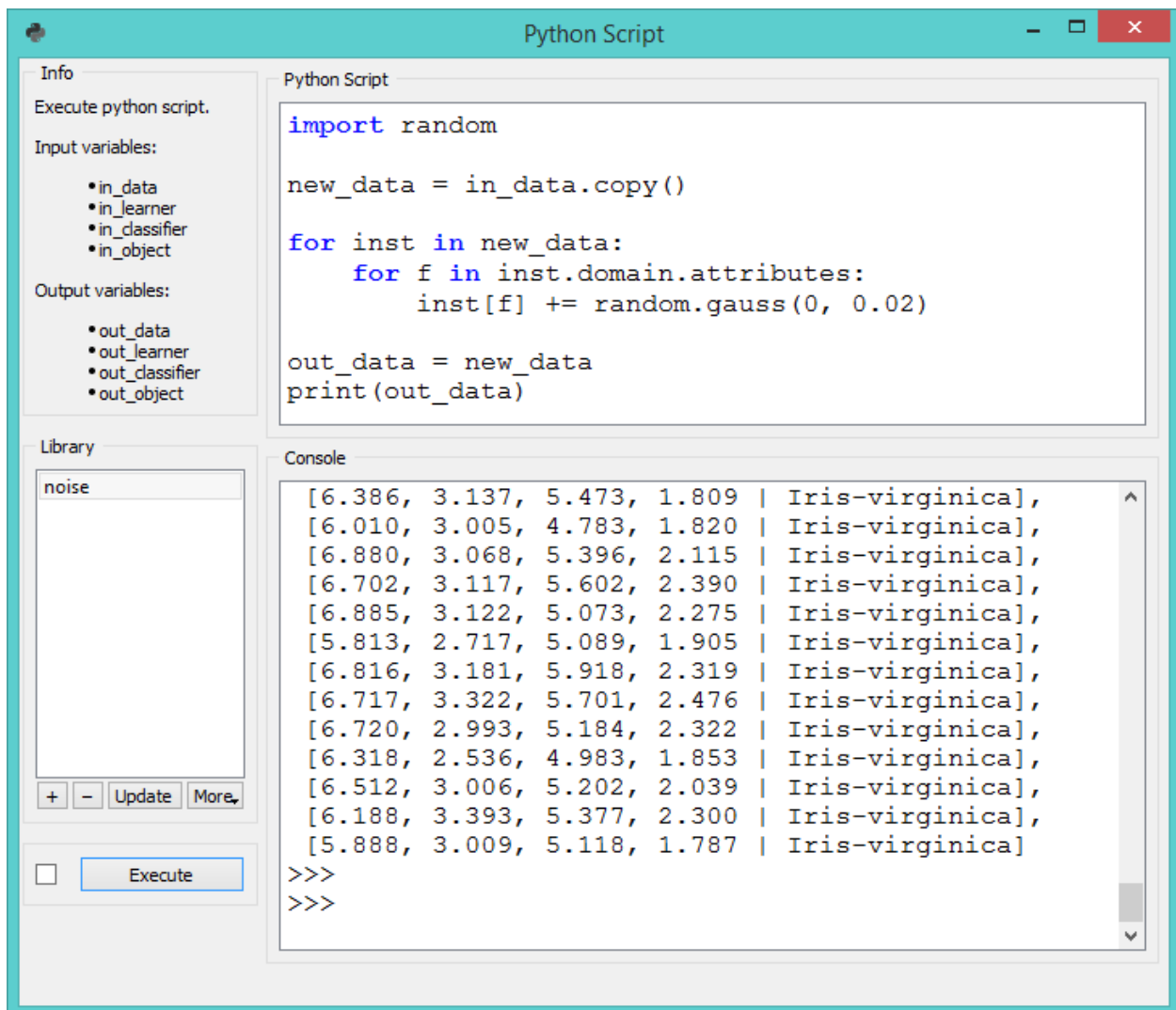
np.round(out_data.X, 0, out_data.X)

print(out_data)
  
```

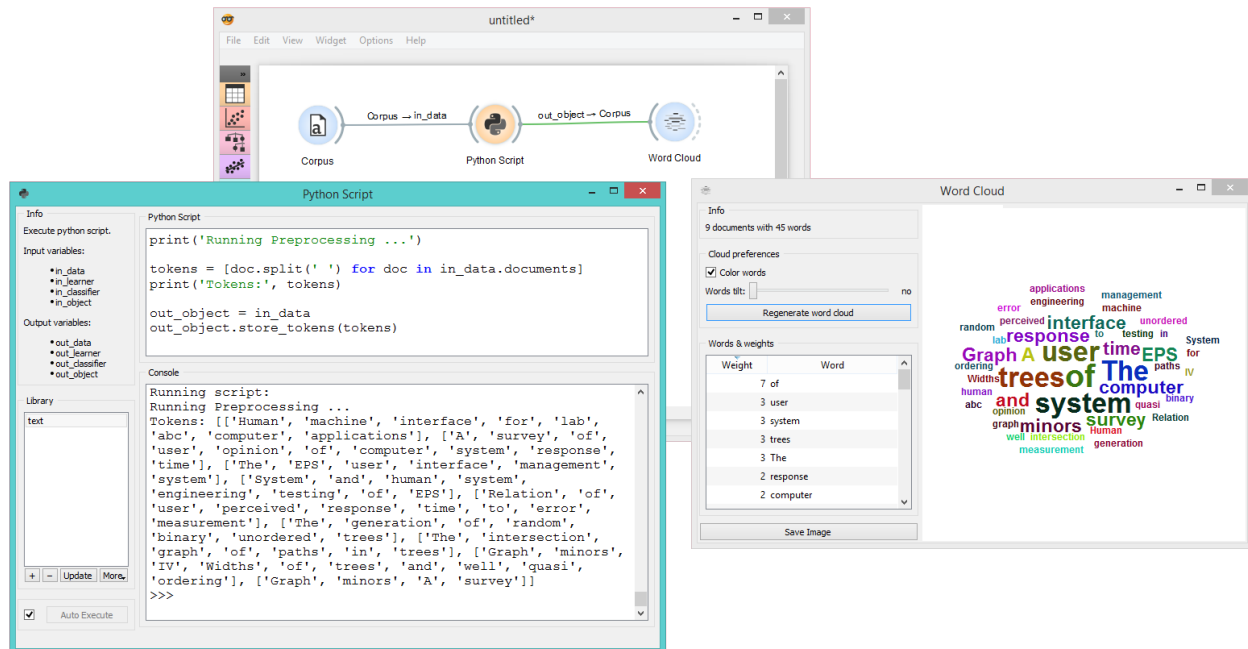
Console

```

[14.000, 6.000, 2.000, 20.000, 95.000, 2.000, 1.000,
1.000, 1.000, 8.000, 1.000, 2.000, 740.000 | 3],
[13.000, 4.000, 2.000, 23.000, 102.000, 2.000, 1.000,
0.000, 1.000, 7.000, 1.000, 2.000, 750.000 | 3],
[13.000, 4.000, 2.000, 20.000, 120.000, 2.000, 1.000,
0.000, 1.000, 10.000, 1.000, 2.000, 835.000 | 3],
[13.000, 3.000, 2.000, 20.000, 120.000, 2.000, 1.000,
1.000, 1.000, 9.000, 1.000, 2.000, 840.000 | 3],
[14.000, 4.000, 3.000, 24.000, 96.000, 2.000, 1.000,
1.000, 1.000, 9.000, 1.000, 2.000, 560.000 | 3]
>>>
  
```







## Feature Constructor



Add new features to your data set.

## Signals

**Inputs:**

- **Data**

## A data set

### Outputs:

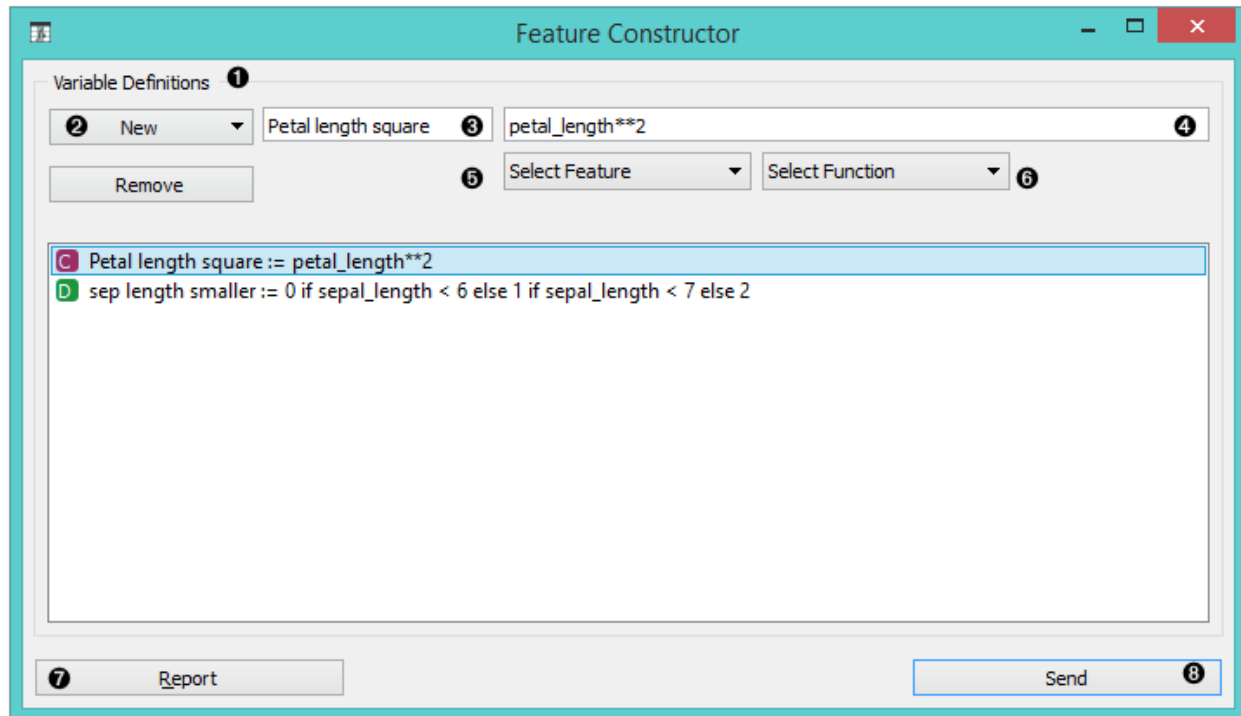
- **Data**

### A modified data set

### Description

The **Feature Constructor** allows you to manually add features (columns) into your data set. The new feature can be a computation of an existing one or a combination of several (addition, subtraction, etc.). You can choose what type of feature it will be (discrete, continuous or string) and what its parameters are (name, value, expression). For continuous variables you only have to construct an expression in Python.

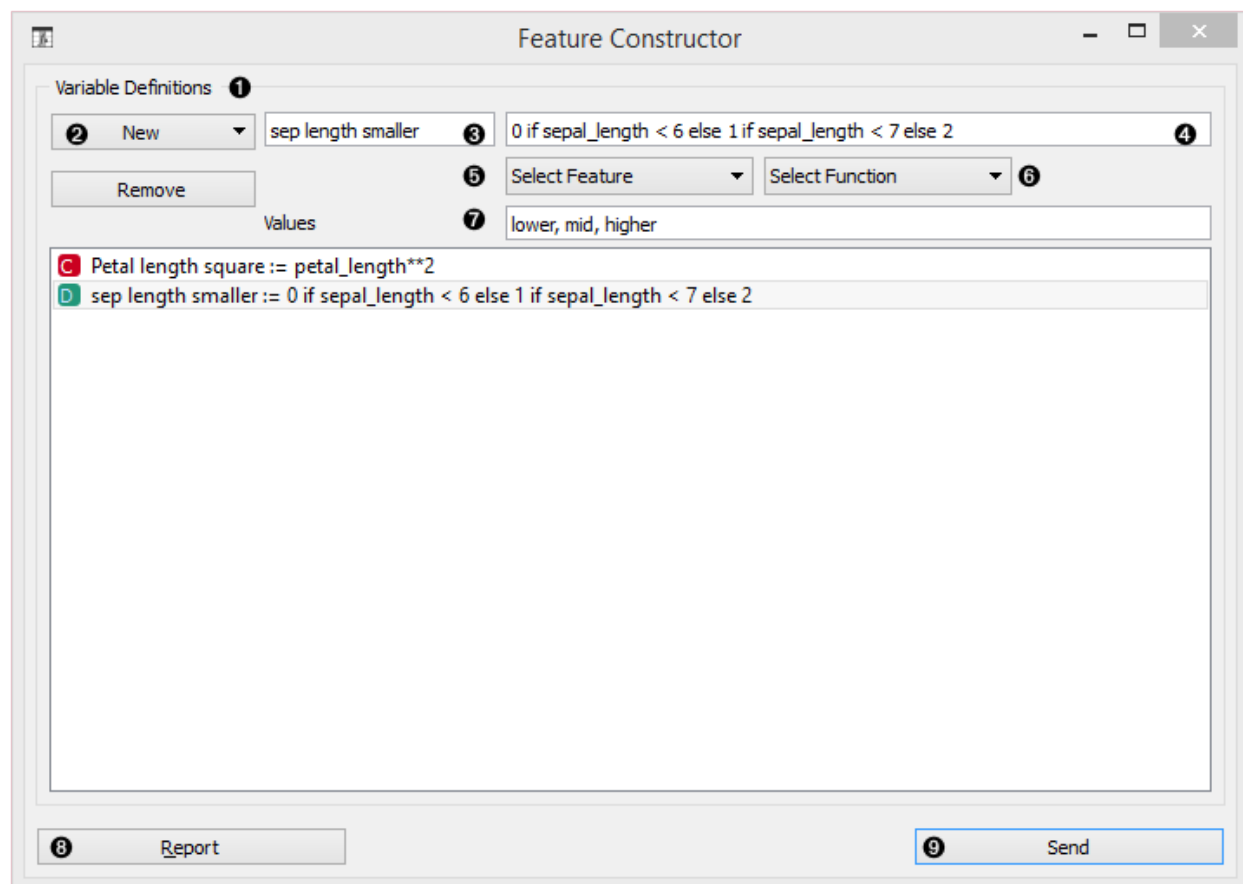
1. List of constructed variables
2. Add or remove variables.
3. New feature name



4. Expression in Python
5. Select a feature.
6. Select a function.
7. Produce a report.
8. Press *Send* to communicate changes.

For discrete variables, however, there's a bit more work. First add or remove the values you want for the new feature. Then select the base value and the expression. In the example below, we have constructed an expression with 'if lower than' and defined three conditions; the program ascribes 0 (which we renamed to lower) if the original value is lower than 6, 1 (mid) if it is lower than 7 and 2 (higher) for all the other values. Notice that we use an underscore for the feature name (e.g. `petal_length`).

1. List of variable definitions
2. Add or remove variables
3. New feature name
4. Expression in Python
5. Select a feature.
6. Select a function.
7. Assign values.
8. Produce a report.
9. Press *Send* to communicate changes.



## Example

With the **Feature Constructor** you can easily adjust or combine existing features into new ones. Below, we added one new discrete feature to the *Titanic* data set. We created a new attribute called *Financial status* and set the values to be *rich* if the person belongs to the first class (status = first) and *not rich* for everybody else. We can see the new data set with *Data Table* widget.

The screenshot shows the Orange3 interface with the following components:

- Feature Constructor\* Workflow:** A workflow diagram showing a 'File' widget connected to a 'Data Table' widget, which is then connected to a 'Feature Constructor' widget, and finally to a 'Data Table (New)' widget.
- Feature Constructor Configuration:**
  - Variable Definitions:** A new variable 'Financial status' is defined with the expression `0 if status=="first" else 1`. The values are listed as 'rich, not rich'.
  - Info:** 2201 instances (no missing values), 4 features (no missing values), Discrete class with 2 values (no missing values), No meta attributes.
  - Variables:** ☒ Show variable labels (if present), ☒ Visualize continuous values, ☒ Color by instance classes.
  - Selection:** ☒ Select full rows.
  - Buttons:** Restore Original Order, Report, Send Automatically.
- Data Table Configuration:**
  - Info:** 2201 instances (no missing values), 3 features (no missing values), Discrete class with 2 values (no missing values), No meta attributes.
  - Variables:** ☒ Show variable labels (if present), ☒ Visualize continuous values, ☒ Color by instance classes.
  - Selection:** ☒ Select full rows.
  - Buttons:** Restore Original Order, Report, Send Automatically.
- Data Table (New) Configuration:**
  - Info:** 2201 instances (no missing values), 4 features (no missing values), Discrete class with 2 values (no missing values), No meta attributes.
  - Variables:** ☒ Show variable labels (if present), ☒ Visualize continuous values, ☒ Color by instance classes.
  - Selection:** ☒ Select full rows.
  - Buttons:** Restore Original Order, Report, Send Automatically.

## Hints

If you are unfamiliar with Python math language, here's a quick introduction.

- +, - to add, subtract
- \* to multiply
- / to divide
- % to divide and return the remainder
- \*\* for exponent (for square root square by 0.5)
- // for floor division
- <, >, <=, >= less than, greater than, less or equal, greater or equal
- == for equal
- != for not equal

As in the example: (value) if (feature name) < (value), else (value) if (feature name) < (value), else (value)

[Use value 1 if feature is less than specified value, else use value 2 if feature is less than specified value 2, else use value 3.]

See more [here](#).

## Edit Domain



### Signals

#### Inputs:

- **Data**  
An input data set

#### Outputs:

- **Data**  
An edited output data set

### Description

This widget can be used to edit/change a data set's domain.

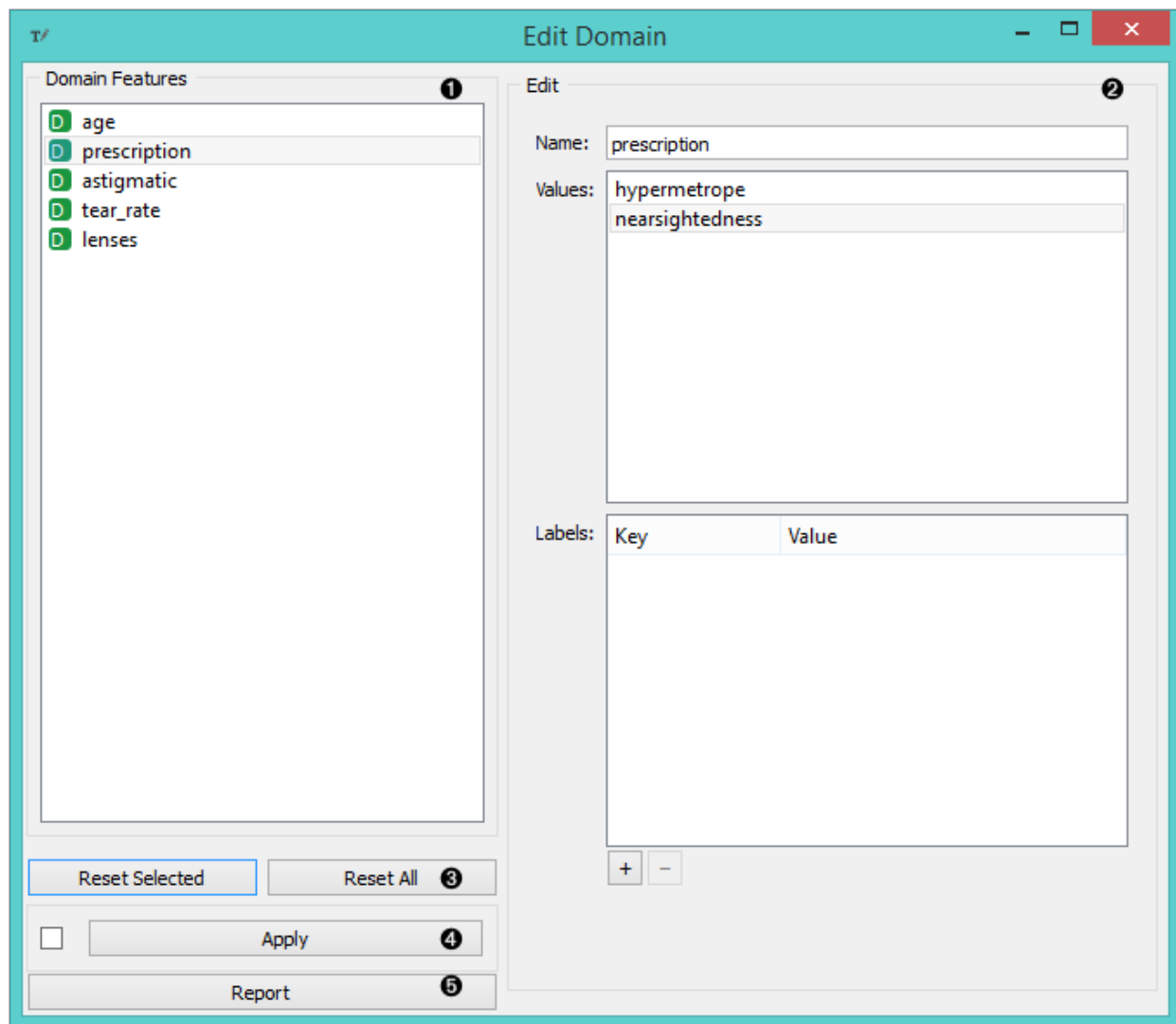
1. All features (including meta attributes) from the input data set are listed in the *Domain Features* list in the box on the left. Selecting one feature displays an editor on the right.
2. The name of the feature can be changed in the *Name* line edit. For *Discrete* features, value names can also be changed in the *Values* list box. Additional feature annotations can be added/removed/edited in the *Labels* box. To add a new label, click the “+” button and add the *Key* and *Value* columns for the new entry. Selecting an existing label and pressing “-” will remove the annotation.
3. To revert the changes made to the feature, press the *Reset Selected* button in the *Reset* box while the feature is selected in the *Domain Features* list. Pressing *Reset All* will reset all features in the domain at the same time.
4. Pressing the *Apply* button will send the changed domain data set to the output channel.

### Example

Below, we demonstrate how to simply edit an existing domain. We selected the *lenses.tab* data set and edited the *perscription* attribute. Where in the original we had the values *myope* and *hypermetrope*, we changed it into *nearsightedness* and *farsightedness* instead. For an easier comparison, we fed both the original and edited data into the [Data Table](#) widget.

## Image Viewer

Displays images that come with a data set.



The screenshot illustrates the 'Edit Domain' workflow in Orange. A central 'Data Table' widget is connected to an 'Edit Domain' widget, which in turn is connected to a 'Data Table (Edited Domain)' widget. The 'Edit Domain' widget's interface is shown in a detailed view, displaying a list of domain features (age, prescription, astigmatic, tear\_rate, lenses) and a table of values for the selected feature 'prescription'.

**Edit Domain\* Interface:**

- Domain Features:**
  - age
  - prescription
  - astigmatic
  - tear\_rate
  - lenses
- Edit:**
  - Name: prescription
  - Values: farsightedness, nearsightedness
  - Labels: Key, Value
- Buttons:** Reset Selected, Reset All, Apply Automatically, Report.

**Data Table Interface:**

	lenses	age	prescription	astigmatic	tear_rate
1	none	young	myope	no	reduced
2	soft	young	myope	no	normal
3	none	young	myope	yes	reduced
4	hard	young	myope	yes	normal
5	none	young	hypermetrope	no	reduced
6	soft	young	hypermetrope	no	normal
7	none	young	hypermetrope	yes	reduced
8	hard	young	hypermetrope	yes	normal
9	none	pre-presbyopic	myope	no	reduced
10	soft	pre-presbyopic	myope	no	normal
11	none	pre-presbyopic	myope	yes	reduced
12	hard	pre-presbyopic	myope	yes	normal
13	none	pre-presbyopic	hypermetrope	no	reduced
14	soft	pre-presbyopic	hypermetrope	no	normal

**Data Table (Edited Domain) Interface:**

	lenses	age	prescription	astigmatic	tear_rate
1	none	young	nearsightedness	no	reduced
2	soft	young	nearsightedness	no	normal
3	none	young	nearsightedness	yes	reduced
4	hard	young	nearsightedness	yes	normal
5	none	young	farsightedness	no	reduced
6	soft	young	farsightedness	no	normal
7	none	young	farsightedness	yes	reduced
8	hard	young	farsightedness	yes	normal
9	none	pre-presbyopic	nearsightedness	no	reduced
10	soft	pre-presbyopic	nearsightedness	no	normal
11	none	pre-presbyopic	nearsightedness	yes	reduced
12	hard	pre-presbyopic	nearsightedness	yes	normal
13	none	pre-presbyopic	farsightedness	no	reduced
14	soft	pre-presbyopic	farsightedness	no	normal



## Signals

### Inputs:

- **Data**

A data set with images.

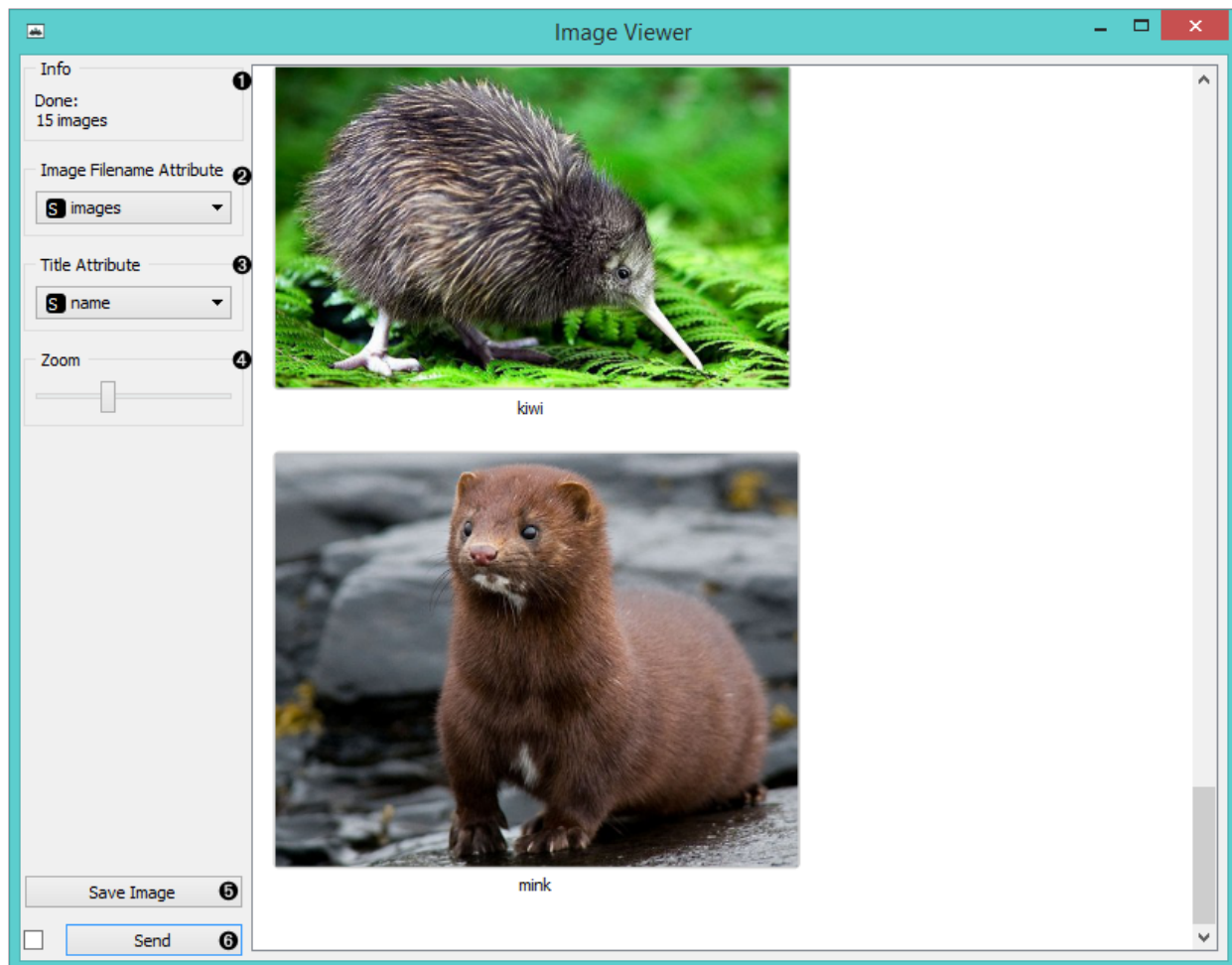
### Outputs:

- **Data**

Images that come with the data.

## Description

The **Image Viewer** widget can display images from a data set, which are stored locally or on the internet. It can be used for image comparison, while looking for similarities or discrepancies between selected data instances (e.g. bacterial growth or bitmap representations of handwriting).



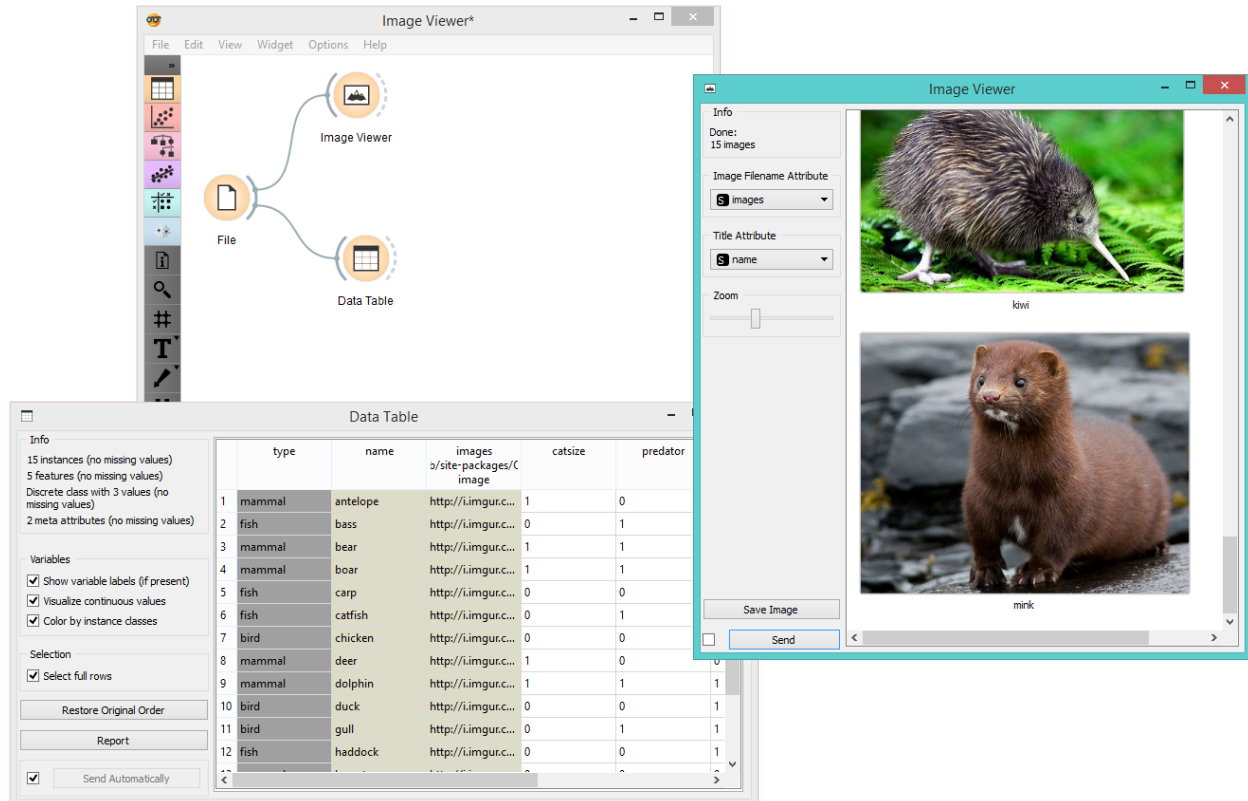
1. Information on the data set
2. Select the column with image data (links).
3. Select the column with image titles.



4. Zoom in or out.
5. Saves the visualization in a file.
6. Tick the box on the left to commit changes automatically. Alternatively, click *Send*.

## Examples

A very simple way to use this widget is to connect the *File* widget with **Image Viewer** and see all the images that come with your data set.



Alternatively, you can visualize only selected instances, as shown in the example below.

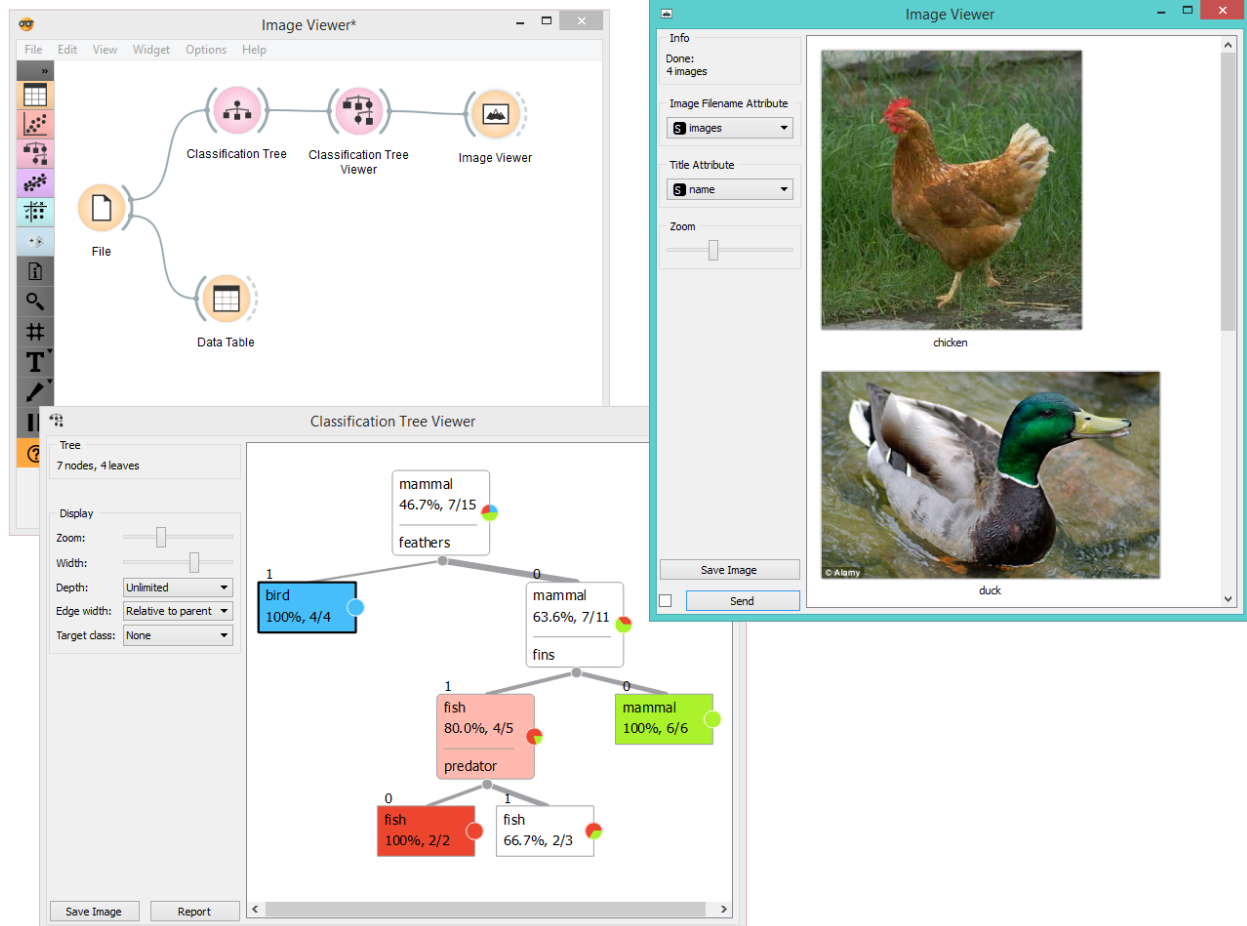
## Impute

Replaces unknown values in the data.

## Signals

### Inputs

- **Data**  
A data set.
- **Learner for Imputation**



A learning algorithm to be used when values are imputed with a predictive model. This algorithm, if given, substitutes the default (1-NN).

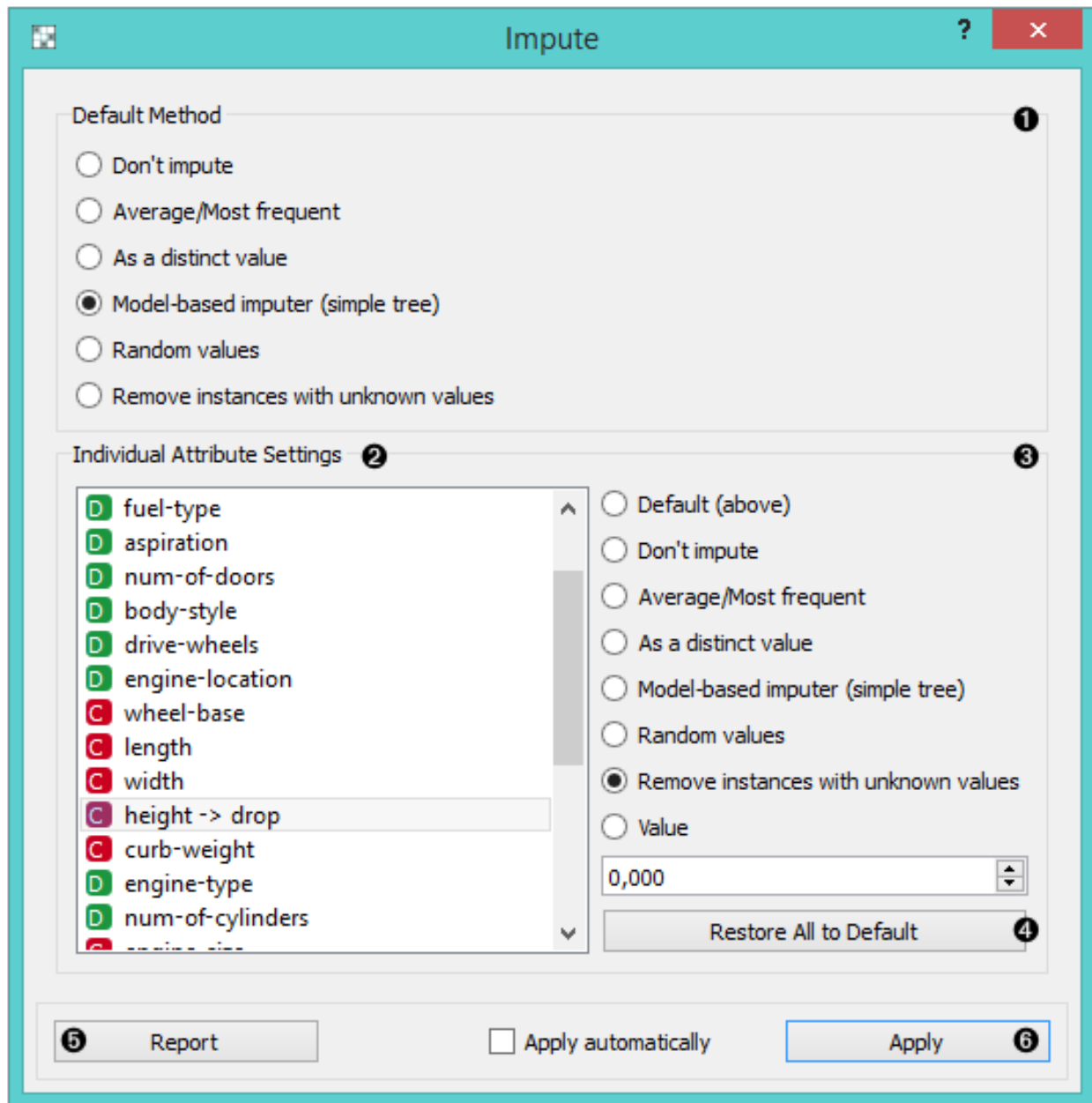
### Outputs

- **Data**

The same data set as in the input, but with the missing values imputed.

### Description

Some Orange's algorithms and visualizations cannot handle unknown values in the data. This widget does what statisticians call imputation: it substitutes missing values by values either computed from the data or set by the user.



1. In the top-most box, *Default method*, the user can specify a general imputation technique for all attributes.
  - **Don't Impute** does nothing with the missing values.
  - **Average/Most-frequent** uses the average value (for continuous attributes) or the most common value (for discrete attributes).
  - **As a distinct value** creates new values to substitute the missing ones.
  - **Model-based imputer** constructs a model for predicting the missing value, based on values of other attributes; a separate model is constructed for each attribute. The default model is 1-NN learner, which takes the value from the most similar example (this is sometimes referred to as hot deck imputation). This algorithm can be substituted by one that the user connects to the input signal Learner for Imputation. Note, however, that if there are discrete and continuous attributes in the data, the algorithm needs to be capable of handling them both; at the moment only 1-NN learner can do that. (In the future, when Orange has more regressors, the Impute widget may have separate input signals for discrete and continuous models.)
  - **Random values** computes the distributions of values for each attribute and then imputes by picking random values from them.
  - **Remove examples with missing values** removes the example containing missing values. This check also applies to the class attribute if *Impute class values* is checked.
2. It is possible to specify individual treatment for each attribute, which overrides the default treatment set. One can also specify a manually defined value used for imputation. In the screenshot, we decided not to impute the values of “*normalized-losses*” and “*make*”, the missing values of “*aspiration*” will be replaced by random values, while the missing values of “*body-style*” and “*drive-wheels*” are replaced by “*hatchback*” and “*fwd*”, respectively. If the values of “*length*”, “*width*” or “*height*” are missing, the example is discarded. Values of all other attributes use the default method set above (model-based imputer, in our case).
3. The imputation methods for individual attributes are the same as default. methods.
4. *Restore All to Default* resets the individual attribute treatments to default.
5. Produce a report.
6. All changes are committed immediately if *Apply automatically* is checked. Otherwise, *Apply* needs to be ticked to apply any new settings.

## Example

To demonstrate how the **Impute** widget works, we played around with the *Iris* data set and deleted some of the data. We used the **Impute** widget and selected the *Model-based imputer* to impute the missing values. In another [Data Table](#), we see how the question marks turned into distinct values (“*Iris-setosa*”, “*Iris-versicolor*”).

## Merge Data

Merges two data sets, based on values of selected attributes.

## Signals

### Inputs:

- **Data A**  
Attribute-valued data set.

The screenshot displays the Orange Visual Programming interface with the following components:

- Workflow:** A 'File' widget is connected to an 'Impute' widget, which is then connected to a 'Data Table' widget. The output of the 'Impute' widget is labeled 'Data Table (Imputed)'.
- Impute Widget Settings:**
  - Default Method:**
    - ☐ Don't impute
    - ☐ Average/Most frequent
    - ☐ As a distinct value
    - ☒ Model-based imputer (simple tree)
    - ☐ Random values
    - ☐ Remove instances with unknown values
  - Individual Attribute Settings:**
    - sepal length:** ☒ (Model-based imputer)
    - sepal width:** ☒ (Model-based imputer)
    - petal length:** ☒ (Model-based imputer)
    - petal width:** ☒ (Model-based imputer)
    - iris -> model (simple tree):** ☒ (Model-based imputer)
  - Other Options:**
    - ☐ Default (above)
    - ☐ Don't impute
    - ☐ Average/Most frequent
    - ☐ As a distinct value
    - ☒ Model-based imputer (simple tree)
    - ☐ Random values
    - ☐ Remove instances with unknown values
    - ☐ Value
  - Buttons:** 'Report', 'Apply automatically', 'Apply', 'Restore All to Default'.
- Data Table Widget:**
  - Info:** 150 instances (no missing values), 4 features (no missing values), Discrete class with 3 values (2.0% missing values), No meta attributes.
  - Variables:**
    - ☒ Show variable labels (if present)
    - ☒ Visualize continuous values
    - ☒ Color by instance classes
  - Selection:**
    - ☒ Select full rows
  - Buttons:** 'Restore Original Order', 'Report', 'Send Automatically'.
- Data Table (Imputed) Widget:**
  - Info:** 150 instances (no missing values), 4 features (no missing values), Discrete class with 3 values (no missing values), No meta attributes.
  - Variables:**
    - ☒ Show variable labels (if present)
    - ☒ Visualize continuous values
    - ☒ Color by instance classes
  - Selection:**
    - ☒ Select full rows
  - Buttons:** 'Restore Original Order', 'Report', 'Send Automatically'.



- **Data B**

Attribute-valued data set.

**Outputs:**

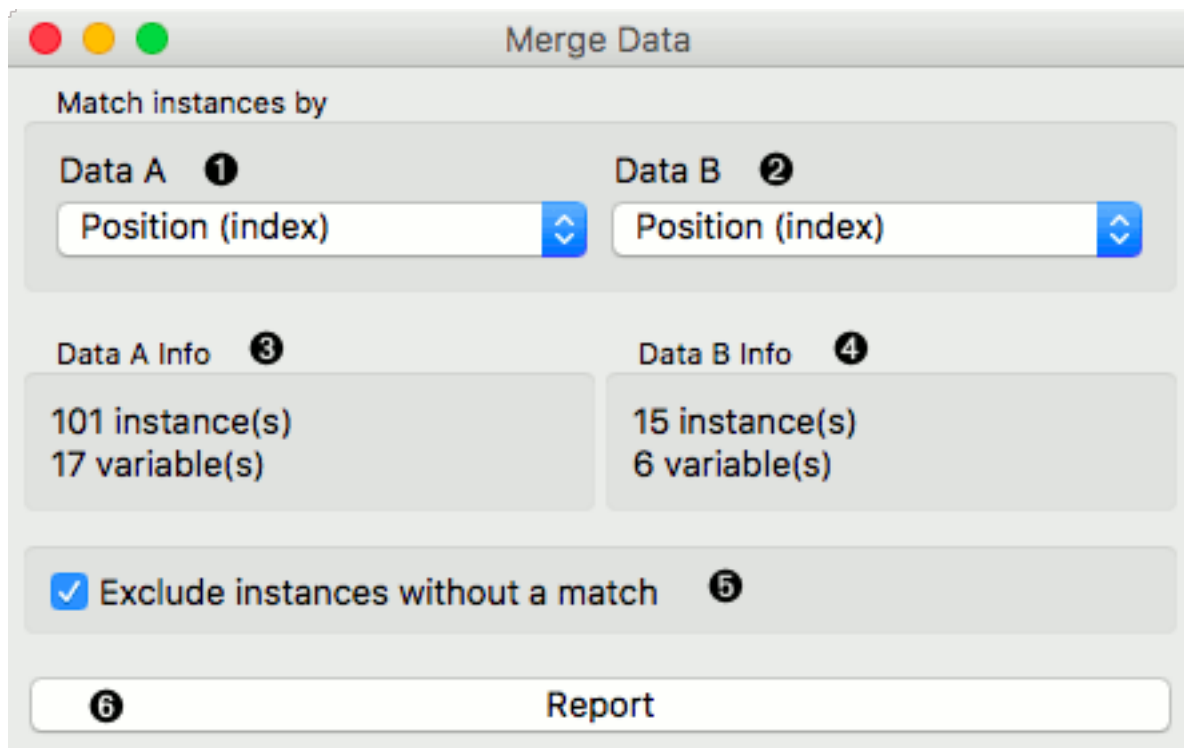
- **Merged Data**

Instances from input data A to which attributes from input data B are added.

## Description

The **Merge Data** widget is used to horizontally merge two data sets, based on values of selected attributes. In the input, two data sets are required, A and B. The widget allows selection of an attribute from each domain, which will be used to perform the merging. The widget produces one output. It corresponds to instances from the input data A to which attributes from B are appended, and B+A to instances from B to which attributes from A are appended.

Merging is done by values of selected (merging) attributes. First, the value of the merging attribute from A is taken and instances from B are searched for matching values. If more than a single instance from B is found, the first one is taken and horizontally merged with the instance from A. If no instance from B matches the criterion, unknown values are assigned to the appended attributes.



1. List of comparable attributes from Data A
2. List of comparable attributes from Data B
3. Information on Data A
4. Information on Data B
5. If checked, instances from B without the match are excluded from the output. If not checked, instances from B without the match are assigned unknown values to the appended attributes.

- Produce a report.

## Example

Merging two data sets results in appending new attributes to the original file, based on a selected common attribute. In the example below, we wanted to merge the **zoo.tab** file containing only factual data with **zoo-with-images.tab** containing images. Both files share a common string attribute *names*. Now, we create a workflow connecting the two files. The **zoo.tab** data is connected to **Data A** input of the **Merge Data** widget, and the **zoo-with-images.tab** data to the **Data B** input. Outputs of the **Merge Data** widget is then connected to the **Data Table** widget. In the latter, the **Merged Data** channels are shown, where image attributes are added to the original data.

The screenshot displays the Orange Visual Programming interface with a workflow for merging two data files. The workflow consists of two 'File' widgets connected to a 'Merge Data' widget, which is then connected to a 'Data Table' widget. The 'Merge Data' widget's settings are shown, indicating it matches instances by the 'name' attribute. The 'Data Table' widget's settings and data preview are also shown.

**Merge Data Widget Settings:**

- Match instances by: Data A (name), Data B (name)
- Data A Info: 101 instance(s), 17 variable(s)
- Data B Info: 15 instance(s), 6 variable(s)
- ☒ Exclude instances without a match
- Report

**Data Table Widget Settings:**

- Info: 15 instances (no missing values), 16 features (no missing values), Discrete class with 7 values (no missing values), 2 meta attributes (no missing values)
- Variables: ☒ Show variable labels (if present), ☐ Visualize continuous values, ☒ Color by instance classes
- Selection: ☒ Select full rows
- Buttons: Restore Original Order, Report
- ☒ Send Automatically

**Data Table Preview:**

	type	name	images na/orange3/Orang image	hair	feathe
1	mammal	antelope	http://i.imgu...	1	0
2	fish	bass	http://i.imgu...	0	0
3	mammal	bear	http://i.imgu...	1	0
4	mammal	boar	http://i.imgu...	1	0
5	fish	carp	http://i.imgu...	0	0
6	fish	catfish	http://i.imgu...	0	0
7	bird	chicken	http://i.imgu...	0	1
8	mammal	deer	http://i.imgu...	1	0
9	mammal	dolphin	http://i.imgu...	0	0
10	bird	duck	http://i.imgu...	0	1
11	bird	gull	http://i.imgu...	0	1
12	fish	haddock	http://i.imgu...	0	0
13	mammal	hamster	http://i.imgu...	1	0
14	bird	kiwi	http://i.imgu...	0	1
15	mammal	mink	http://i.imgu...	1	0

The case where we want to include all instances in the output, even those where no match by attribute *names* was found, is shown in the following workflow.

## Hint

If the two data sets consist of equally-named attributes (other than the ones used to perform the merging), Orange will check by default for consistency of the values of these attributes and report an error in case of non-matching values. In order to avoid the consistency checking, make sure that new attributes are created for each data set: you may use the 'Columns with the same name in different files represent different variables' option in the **File** widget for loading the data.

The screenshot displays the Orange Visual Programming interface with a workflow and three configuration panels.

**Workflow:**

- File (ZOO)** widget outputs **Data A**.
- File (ZOO - images)** widget outputs **Data B**.
- Merge Data** widget takes **Data A** and **Data B** as input and outputs **Merged Data**.
- Data Table** widget takes **Merged Data** as input.

**Merge Data Widget Configuration:**

- Match instances by:** Data A: name, Data B: name.
- Data A Info:** 101 instance(s), 17 variable(s).
- Data B Info:** 15 instance(s), 6 variable(s).
- ☐ Exclude instances without a match.
- Report** button.

**Data Table Widget Configuration:**

- Info:** 101 instances, 16 features (no missing values), Discrete class with 7 values (no missing values), 2 meta attributes (42.6% missing values).
- Variables:**
  - ☒ Show variable labels (if present)
  - ☐ Visualize continuous values
  - ☒ Color by instance classes
- Selection:**
  - ☒ Select full rows
- Buttons:** Restore Original Order, Report.
- ☒ Send Automatically.

**Data Table Data:**

	type	name	images na/orange3/Orang image	hair	feath
1	mammal	aardvark	?	1	0
2	mammal	antelope	http://i.imgu...	1	0
3	fish	bass	http://i.imgu...	0	0
4	mammal	bear	http://i.imgu...	1	0
5	mammal	boar	http://i.imgu...	1	0
6	mammal	buffalo	?	1	0
7	mammal	calf	?	1	0
8	fish	carp	http://i.imgu...	0	0
9	fish	catfish	http://i.imgu...	0	0
10	mammal	cavy	?	1	0
11	mammal	cheetah	?	1	0
12	bird	chicken	http://i.imgu...	0	1
13	fish	chub	?	0	0
14	invertebrate	clam	?	0	0
15	invertebrate	crab	?	0	0
16	invertebrate	crayfish	?	0	0
17	bird	crow	?	0	1



## Outliers



Simple outlier detection by comparing distances between instances.

### Signals

#### Inputs:

- **Data**  
A data set
- **Distances**  
A distance matrix

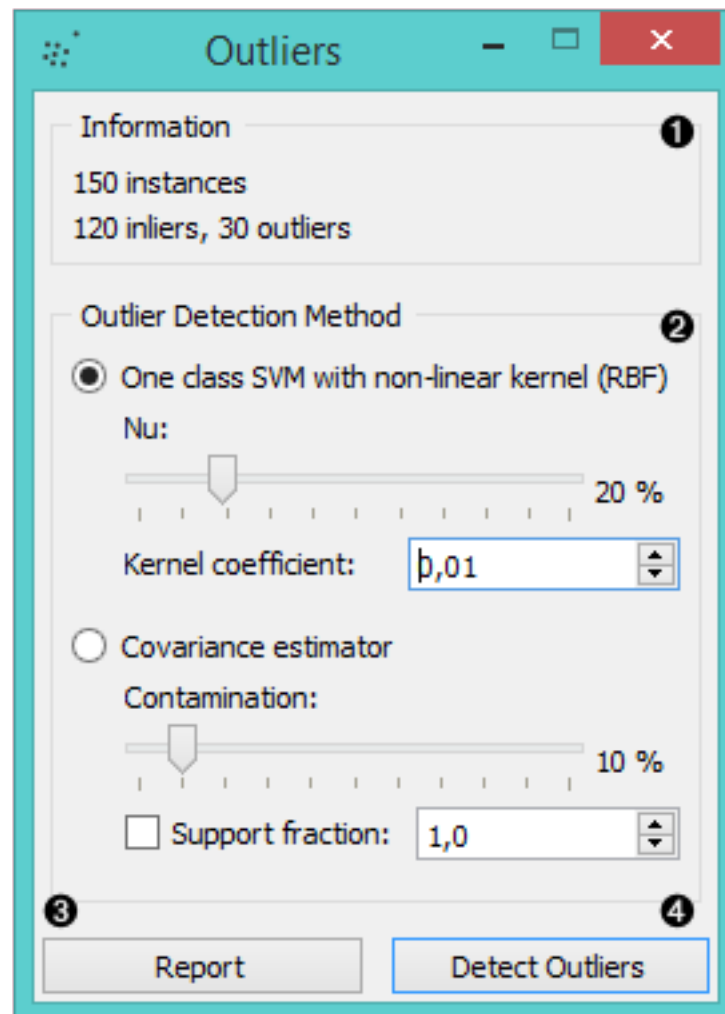
#### Outputs:

- **Outliers**  
A data set containing instances scored as outliers
- **Inliers**  
A data set containing instances not scored as outliers

### Description

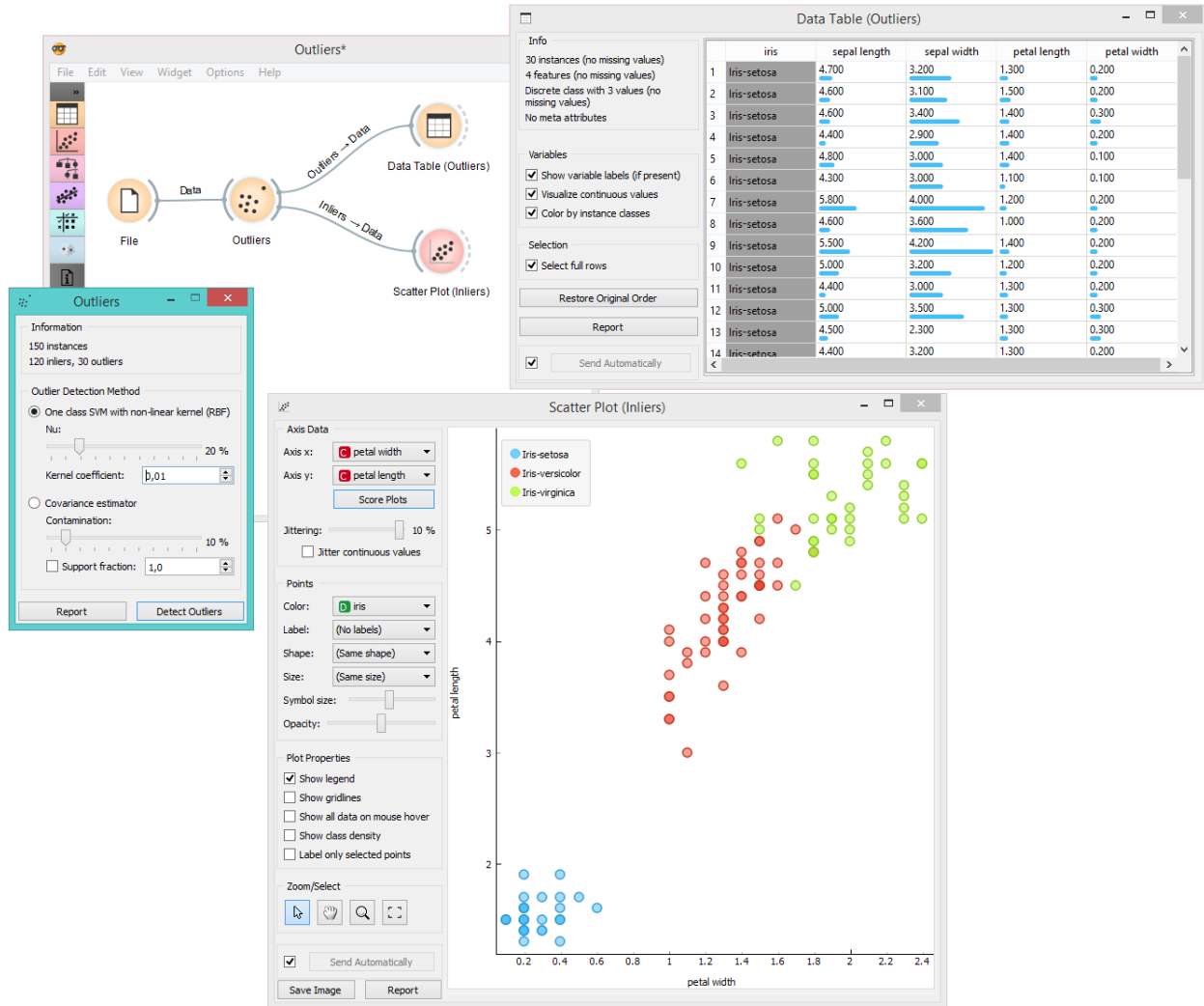
The **Outliers** widget applies one of the two methods for outlier detection. Both methods apply classification to the data set, one with SVM (multiple kernels) and the other with elliptical envelope. *One-class SVM with non-linear kernels (RBF)* performs well with non-Gaussian distributions, while *Covariance estimator* works only for data with Gaussian distribution.

1. Information on the input data, number of inliers and outliers based on the selected model.
2. Select the *Outlier detection method*:
  - **One class SVM with non-linear kernel (RBF)**: classifies data as similar or different from the core class
    - **Nu** is a parameter for the upper bound on the fraction of training errors and a lower bound of the fraction of support vectors
    - **Kernel coefficient** is a gamma parameter, which specifies how much influence a single data instance has
  - **Covariance estimator**: fits ellipsis to central points with Mahalanobis distance metric
    - **Contamination** is the proportion of outliers in the data set
    - **Support fraction** specifies the proportion of points included in the estimate
3. Produce a report.
4. Click *Detect outliers* to output the data.



## Example

Below, is a simple example of how to use this widget. We used the *Iris* data set to detect the outliers. We chose the *one class SVM with non-linear kernel (RBF)* method, with Nu set at 20% (less training errors, more support vectors). Then we observed the outliers in the *Data Table* widget, while we sent the inliers to the *Scatter Plot*.



## Preprocess



Preprocesses data with selected methods.

## Signals

Inputs:

### 2.1. Data

- **Data**

A data set.

**Outputs:**

- **Preprocessor**

A preprocessing method.

- **Preprocessed Data**

Data preprocessed with selected methods.

## Description

Preprocessing is crucial for achieving better-quality analysis results. The **Preprocess** widget offers five preprocessing methods to improve data quality. In this widget, you can immediately discretize continuous values or continuize discrete ones, impute missing values, select relevant features or center and scale them. Basically, this widget combines four separate widgets for simpler processing.

1. List of preprocessors. You drag the preprocessors you wish to use to the right side of the widget.
2. Discretization of continuous values
3. Continuization of discrete values
4. Impute missing values or remove them.
5. Select the most relevant features by information gain, gain ratio, Gini index.
6. Select random features
7. Normalize features
8. Randomize
9. When the box is ticked (*Send Automatically*), the widget will communicate changes automatically. Alternatively, click *Send*.
10. Produce a report.

## Example

In the example below, we have used the *adult* data set and preprocessed the data. We continuized discrete values (age, education and marital status...) as *one attribute per value*, we imputed missing values (replacing ? with average values), selected 10 most relevant attributes by *Information gain*, centered them by mean and scaled by span. We can observe the changes in the [Data Table](#) and compare it to the non-processed data.

## Purge Domain

Removes unused attribute values and useless attributes, sorts the remaining values.

## Signals

**Inputs:**

- **Data**

A data set.

**Preprocessors 1**

- Discretize Continuous Variables**
- Continuize Discrete Variables
- Impute Missing Values
- Select Relevant Features
- Select Random Features
- Normalize Features
- Randomize
- Principal Component Analysis
- CUR Matrix Decomposition

**Discretize Continuous Variables 2**

- ☐ Entropy-MDL discretization
- ☒ Equal frequency discretization
- ☐ Equal width discretization
- Number of intervals (for equal width/frequency):
- ☐ Remove continuous attributes

**Continuize Discrete Variables 3**

- ☒ One attribute per value
- ☐ Most frequent is base
- ☐ Remove all discrete attributes
- ☐ Remove multinomial attributes
- ☐ Treat as ordinal
- ☐ Divide by number of values

**Impute Missing Values 4**

- ☒ Average/Most frequent
- ☐ Replace with random value
- ☐ Remove rows with missing values.

**Select Relevant Features 5**

Score:

Strategy:

- ☒ Fixed:
- ☐ Percentile:

**Select Random Features 6**

Strategy:

- ☒ Fixed:
- ☐ Percentage:

**Normalize Features 7**

Center:

Scale:

The screenshot displays the Orange Visual Programming interface with the following components:

- Preprocess\* Workflow:** A workflow consisting of a **File** widget connected to a **Preprocess** widget, which is then connected to a **Data Table (Preprocessed)** widget.
- Preprocess Widget Configuration:**
  - Preprocessors:**
    - Discretize Continuous Variables
    - Continuize Discrete Variables
    - Impute Missing Values
    - Select Relevant Features
    - Select Random Features
    - Normalize Features
    - Randomize
    - Principal Component Analysis
    - CUR Matrix Decomposition
  - Continuize Discrete Variables:**
    - One attribute per value
    - Most frequent is base
    - Remove all discrete attributes
    - Remove multinomial attributes
    - Treat as ordinal
    - Divide by number of values
  - Impute Missing Values:**
    - Average/Most frequent
    - Replace with random value
    - Remove rows with missing values
  - Select Relevant Features:**
    - Score: Information Gain
    - Strategy: Fixed (10)
    - Percentile: 75,00%
  - Normalize Features:**
    - Center: Center by Mean
    - Scale: Scale by span
- Data Table (Original):**
  - Info:** 32561 instances, 14 features (0.9% missing values), Discrete class with 2 values (no missing values), No meta attributes.
  - Variables:**
    - Show variable labels (if present)
    - Visualize continuous values
    - Color by instance classes
  - Selection:**
    - Select full rows
  - Buttons:** Restore Original Order, Report, Send Automatically.
- Data Table (Preprocessed):**
  - Info:** 32561 instances (no missing values), 10 features (no missing values), Discrete class with 2 values (no missing values), No meta attributes.
  - Variables:**
    - Show variable labels (if present)
    - Visualize continuous values
    - Color by instance classes
  - Selection:**
    - Select full rows
  - Buttons:** Restore Original Order, Report, Send Automatically.



**Outputs:**

- **Data**

A filtered data set

**Description**

Definitions of nominal attributes sometimes contain values which don't appear in the data. Even if this does not happen in the original data, filtering the data, selecting exemplary subsets and alike can remove all examples for which the attribute has some particular value. Such values clutter data presentation, especially various visualizations, and should be removed.

After purging an attribute, it may become single-valued or, in extreme case, have no values at all (if the value of this attribute was undefined for all examples). In such cases, the attribute can be removed.

A different issue is the order of attribute values: if the data is read from a file in a format in which values are not declared in advance, they are sorted "in order of appearance". Sometimes we would prefer to have them sorted alphabetically.

1. Purge attributes.
2. Purge classes.
3. Purge meta attributes.
4. Information on the filtering process
5. Produce a report.
6. If *Apply automatically* is ticked, the widget will output data at each change of widget settings.

Such purification is done by the widget **Purge Domain**. Ordinary attributes and class attributes are treated separately. For each, we can decide if we want the values sorted or not. Next, we may allow the widget to remove attributes with less than two values or remove the class attribute if there are less than two classes. Finally, we can instruct the widget to check which values of attributes actually appear in the data and remove the unused values. The widget cannot remove values if it is not allowed to remove the attributes, since having attributes without values makes no sense.

The new, reduced attributes get the prefix "R", which distinguishes them from the original ones. The values of new attributes can be computed from the old ones, but not the other way around. This means that if you construct a classifier from the new attributes, you can use it to classify the examples described by the original attributes. But not the opposite: constructing a classifier from the old attributes and using it on examples described by the reduced ones won't work. Fortunately, the latter is seldom the case. In a typical setup, one would explore the data, visualize it, filter it, purify it... and then test the final model on the original data.

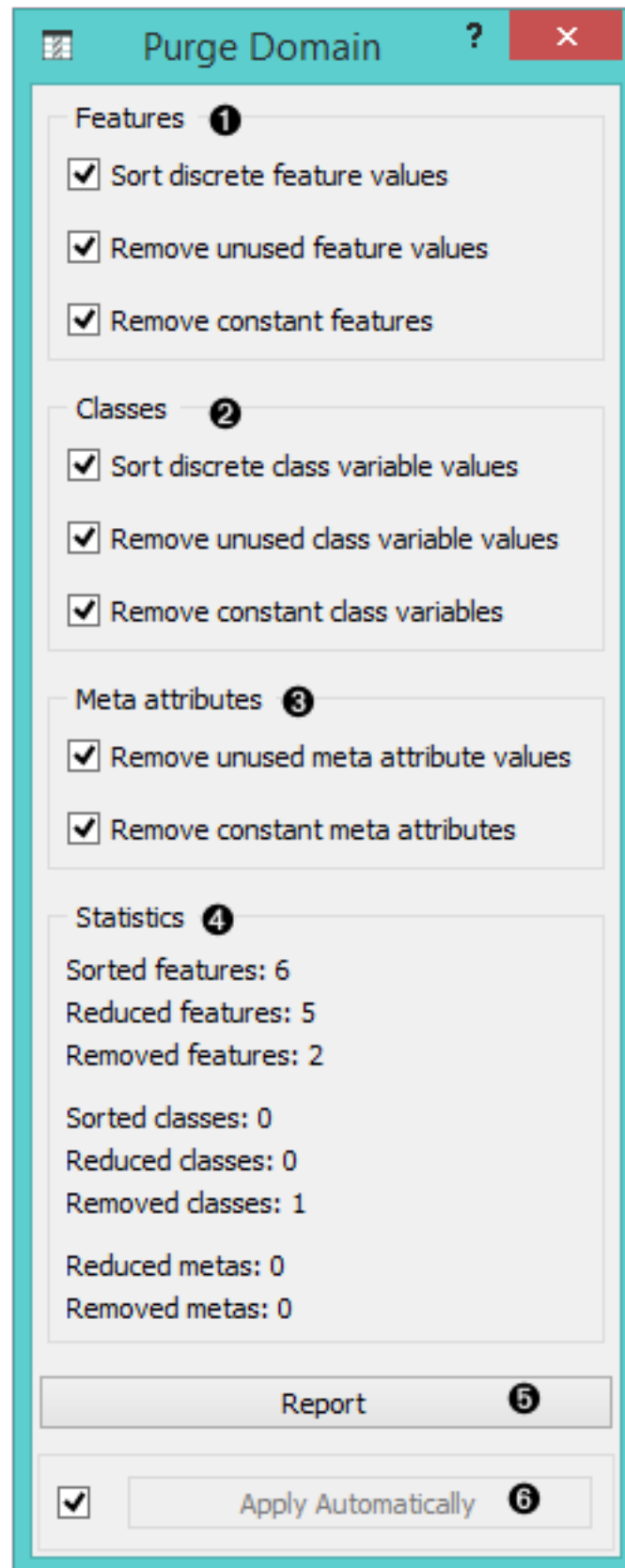
**Example**

The **Purge Domain** widget would typically appear after data filtering, for instance when selecting a subset of visualized examples.

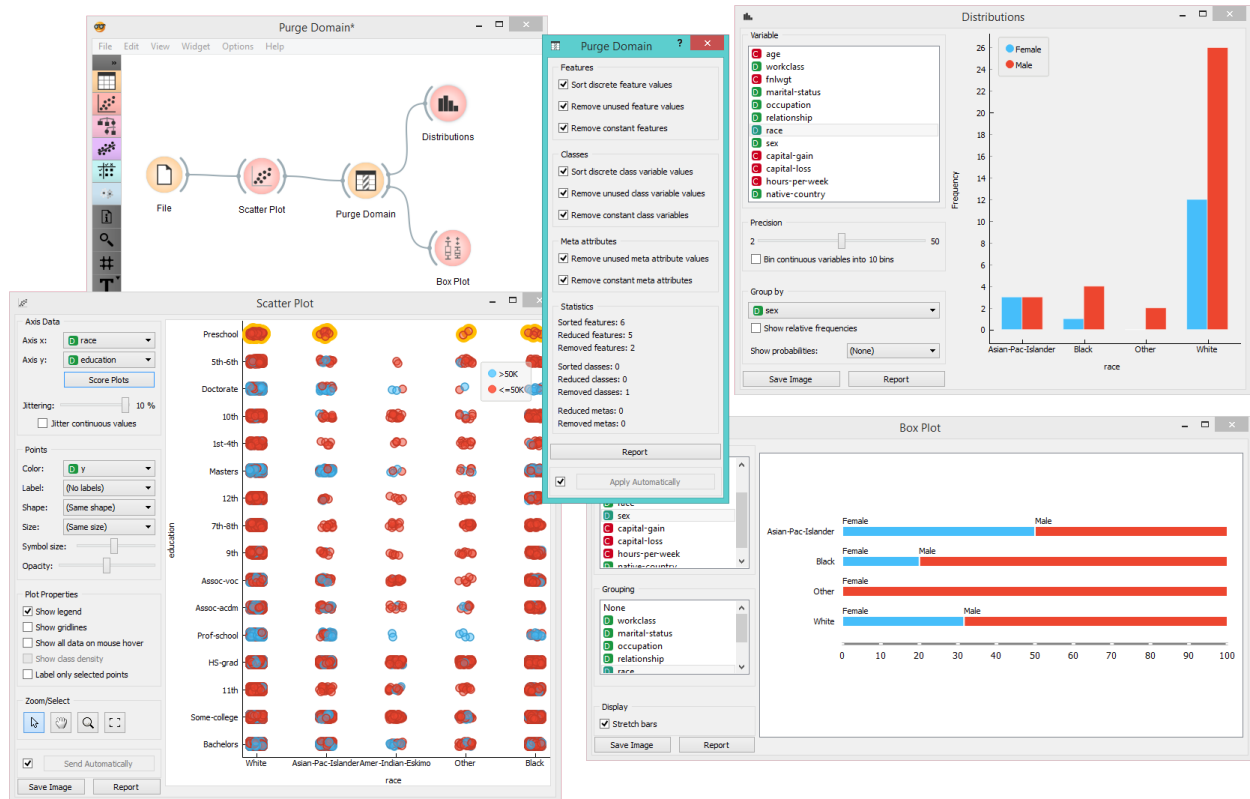
In the above schema, we play with the *adult.tab* data set: we visualize it and select a portion of the data, which contains only four out of the five original classes. To get rid of the empty class, we put the data through **Purge Domain** before going on to the *Box Plot* widget. The latter shows only the four classes which are in the **Purge Data** output. To see the effect of data purification, uncheck *Remove unused class variable values* and observe the effect this has on *Box Plot*.

**Rank**

Ranking of attributes in classification or regression data sets.







## Signals

### Inputs:

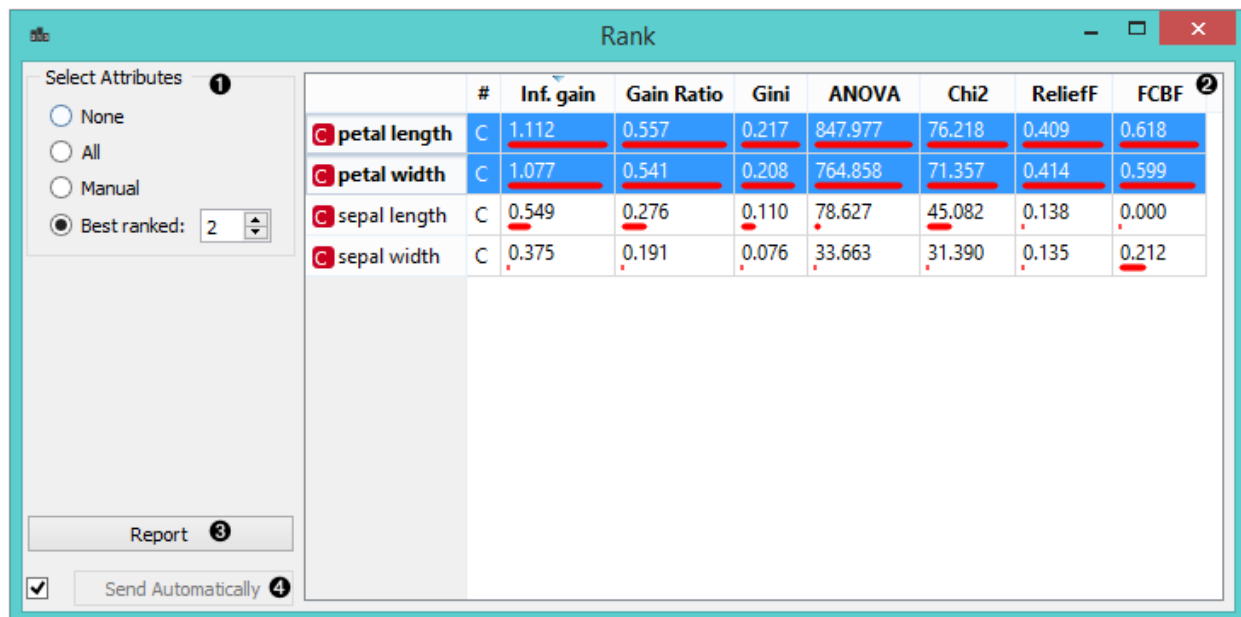
- **Data**  
An input data set.

### Outputs:

- **Reduced Data**  
A data set with selected attributes.

## Description

The **Rank** widget considers class-labeled data sets (classification or regression) and scores the attributes according to their correlation with the class.



1. Select attributes from the data table.
2. Data table with attributes (rows) and their scores by different scoring methods (columns)
3. Produce a report.
4. If 'Send Automatically' is ticked, the widget automatically communicates changes to other widgets.

## Scoring methods

1. **Information Gain**: the expected amount of information (reduction of entropy)
2. **Gain Ratio**: a ratio of the information gain and the attribute's intrinsic information, which reduces the bias towards multivalued features that occurs in information gain
3. **Gini**: the inequality among values of a frequency distribution
4. **ANOVA**: the difference between average values of the feature in different classes

5. **Chi2**: dependence between the feature and the class as measure by the chi-square statistic
6. **ReliefF**: the ability of an attribute to distinguish between classes on similar data instances
7. **FCBF (Fast Correlation Based Filter)**: entropy-based measure, which also identifies redundancy due to pairwise correlations between features

### Example: Attribute Ranking and Selection

Below, we have used the **Rank** widget immediately after the *File* widget to reduce the set of data attributes and include only the most informative ones:



Notice how the widget outputs a data set that includes only the best-scored attributes:

### Example: Feature Subset Selection for Machine Learning

What follows is a bit more complicated example. In the workflow below, we first split the data into a training set and a test set. In the upper branch, the training data passes through the **Rank** widget to select the most informative attributes, while in the lower branch there is no feature selection. Both feature selected and original data sets are passed to their own *Test & Score* widgets, which develop a *Naive Bayes* classifier and score it on a test set.

For data sets with many features, a naive Bayesian classifier feature selection, as shown above, would often yield a better predictive accuracy.

## Color

Set color legend for variables.

## Signals

### Inputs:

- **Data**  
An input data set.

### Outputs:

- **Data**  
A data set with a new color legend.

**Data Table**

Info  
303 instances  
3 features (0.7% missing values)  
Discrete class with 2 values (no missing values)  
No meta attributes

Variables  
☒ Show variable labels (if present)  
☒ Visualize continuous values  
☒ Color by instance classes

Selection  
☒ Select full rows

Restore Original Order

Report

☒ Send Automatically

	diameter narrowin	chest pain	ajor vessels colore	thal
1	0	typical ang	0.000	fixed defect
2	1	asymptomatic	3.000	normal
3	1	asymptomatic	2.000	reversible defect
4	0	non-anginal	0.000	normal
5	0	atypical ang	0.000	normal
6	0	atypical ang	0.000	normal
7	1	asymptomatic	2.000	normal
8	0	asymptomatic	0.000	normal
9	1	asymptomatic	1.000	reversible defect

**Rank**

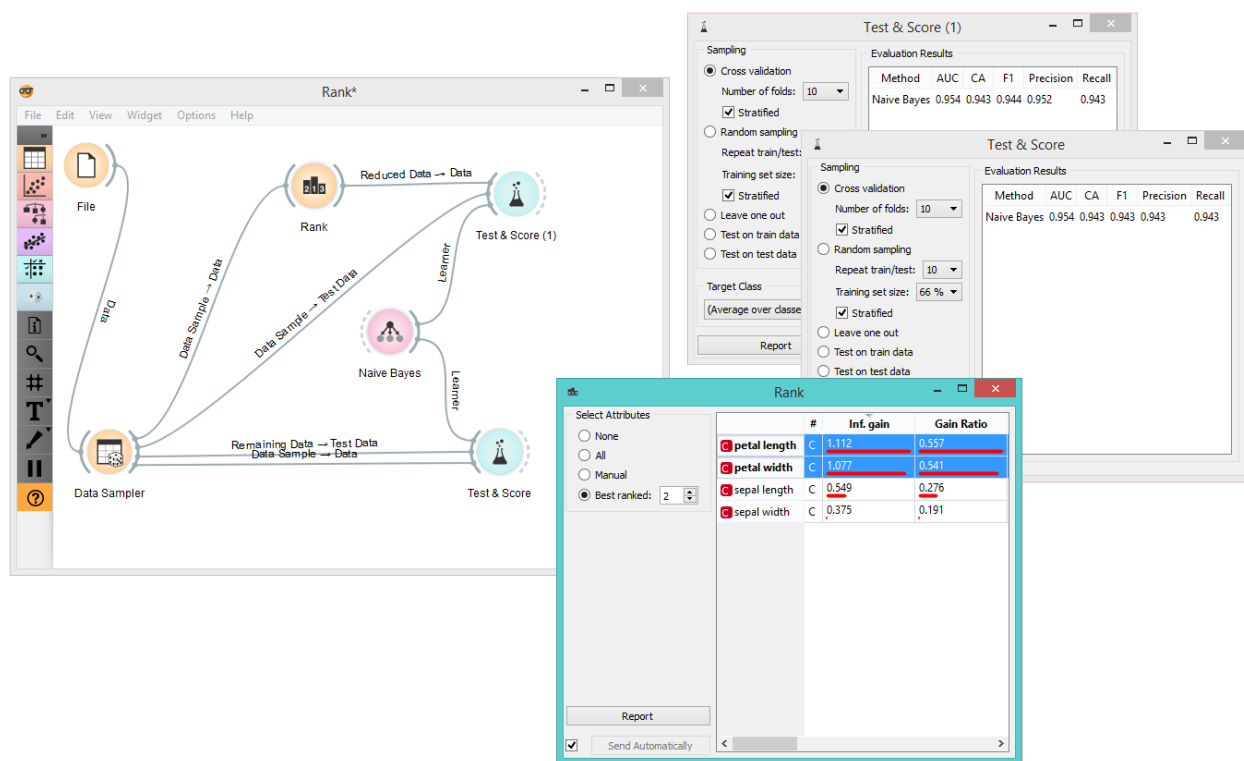
Missing values have been imputed.

Select Attributes  
☐ None  
☐ All  
☐ Manual  
☒ Best ranked: 3

Report

☒ Send Automatically

	#	Inf. gain	Gain Ratio	Gini
<b>D</b> thal	3	0.208	0.167	0.068
<b>D</b> chest pain	4	0.205	0.118	0.067
<b>C</b> major vessels colored	C	0.180	0.115	0.059
<b>C</b> ST by exercise	C	0.145	0.074	0.047
<b>D</b> exerc ind ang	2	0.139	0.153	0.046
<b>C</b> max HR	C	0.123	0.062	0.040
<b>D</b> slope peak exc ST	3	0.112	0.087	0.038
<b>C</b> age	C	0.058	0.029	0.020
<b>D</b> gender	2	0.057	0.063	0.019
<b>D</b> rest ECG	3	0.024	0.022	0.008
<b>C</b> cholesterol	C	0.016	0.008	0.006
<b>C</b> rest SBP	C	0.015	0.008	0.005
<b>D</b> fasting blood sugar > 120	2	0.000	0.001	0.000



## Description

The **Color** widget enables you to set the color legend in your visualizations according to your own preferences. This option provides you with the tools for emphasizing your results and offers a great variety of color options for presenting your data. It can be combined with most visualizations widgets.

1. A list of discrete variables. You can set the color of each variable by double-clicking on it and opening the *Color palette* or the *Select color* window. The widget also enables text-editing. By clicking on a variable, you can change its name.
2. A list of continuous variables. You can customize the color gradients by double-clicking on them. The widget also enables text-editing. By clicking on a variable, you can change its name. If you hover over the right side of the gradient, *Copy to all* appears. You can then apply your customized color gradient to all variables.
3. Produce a report.
4. Apply changes. If *Apply automatically* is ticked, changes will be communicated automatically. Alternatively, just click *Apply*.

## Discrete variables

1. Choose a desired color from the palette of basic colors.
2. Move the cursor to choose a custom color from the color palette.
3. Choose a custom color from your previously saved color choices.
4. Specify the custom color by:
  - entering the red, green, and blue components of the color as values between 0 (darkest) and 255 (brightest)
  - entering the hue, saturation and luminescence components of the color as values in the range 0 to 255
5. Add the created color to your custom colors.
6. Click *OK* to save your choices or *Cancel* to exit the the color palette.

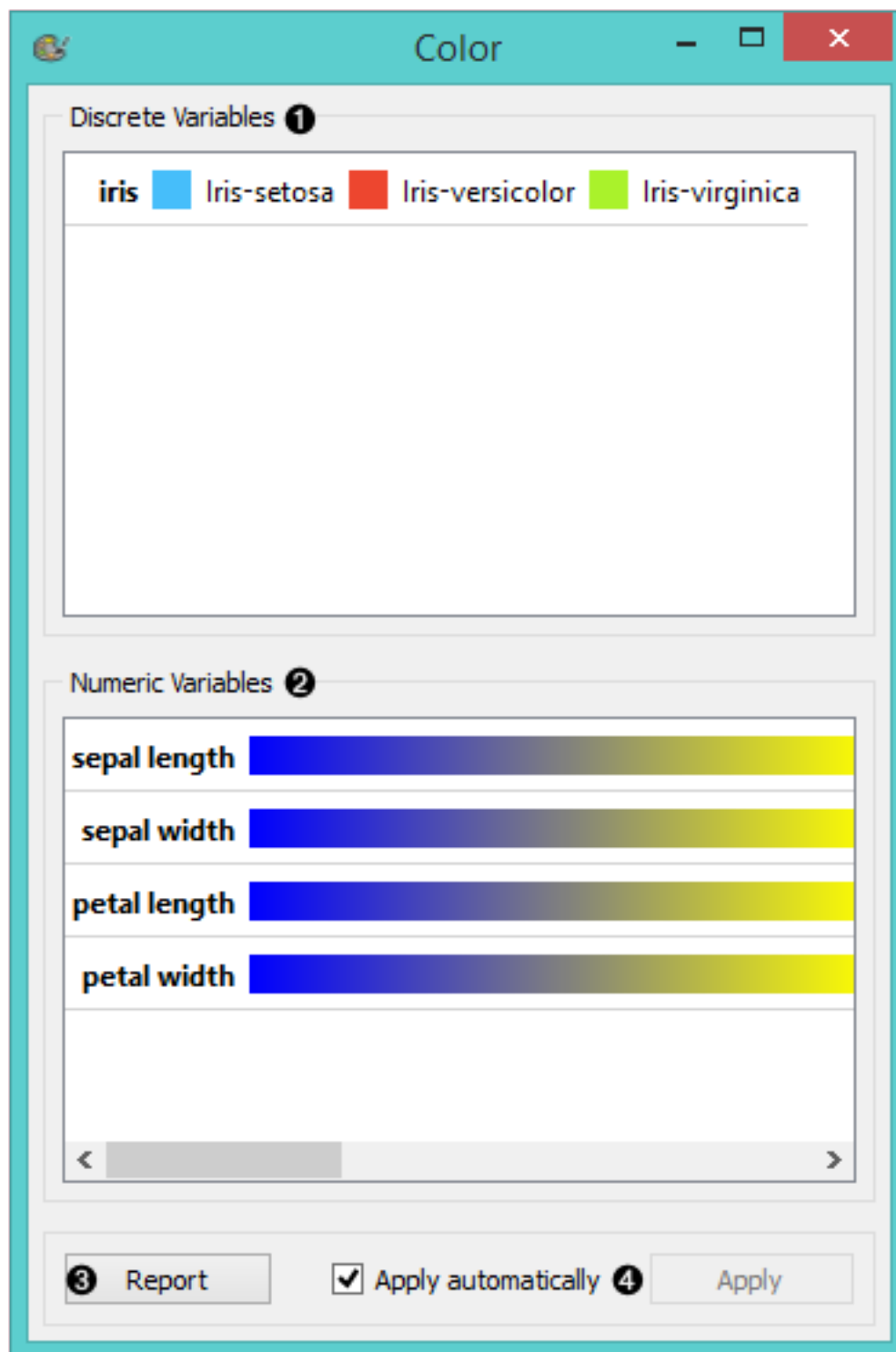
## Numeric variables

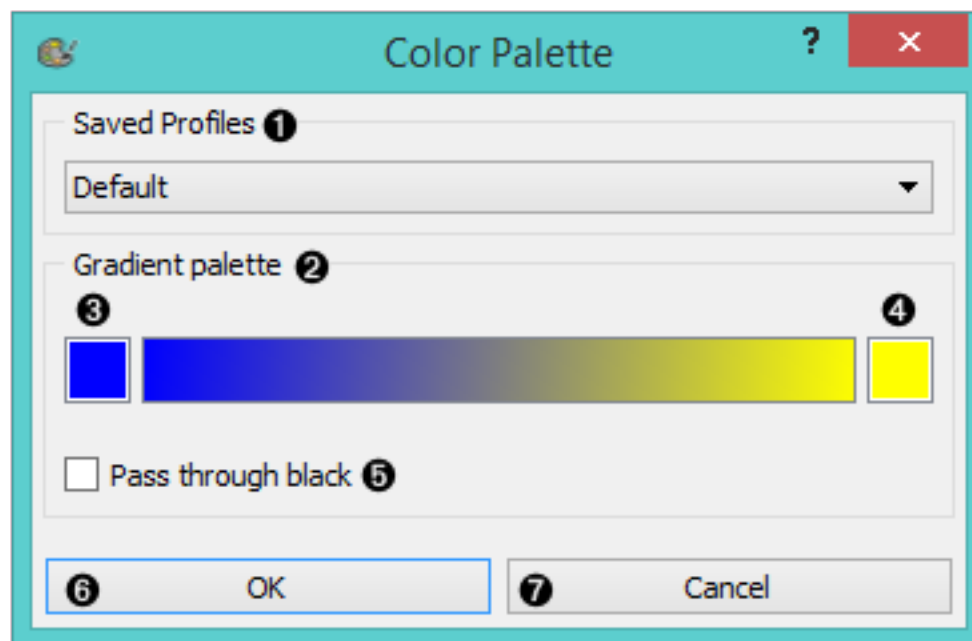
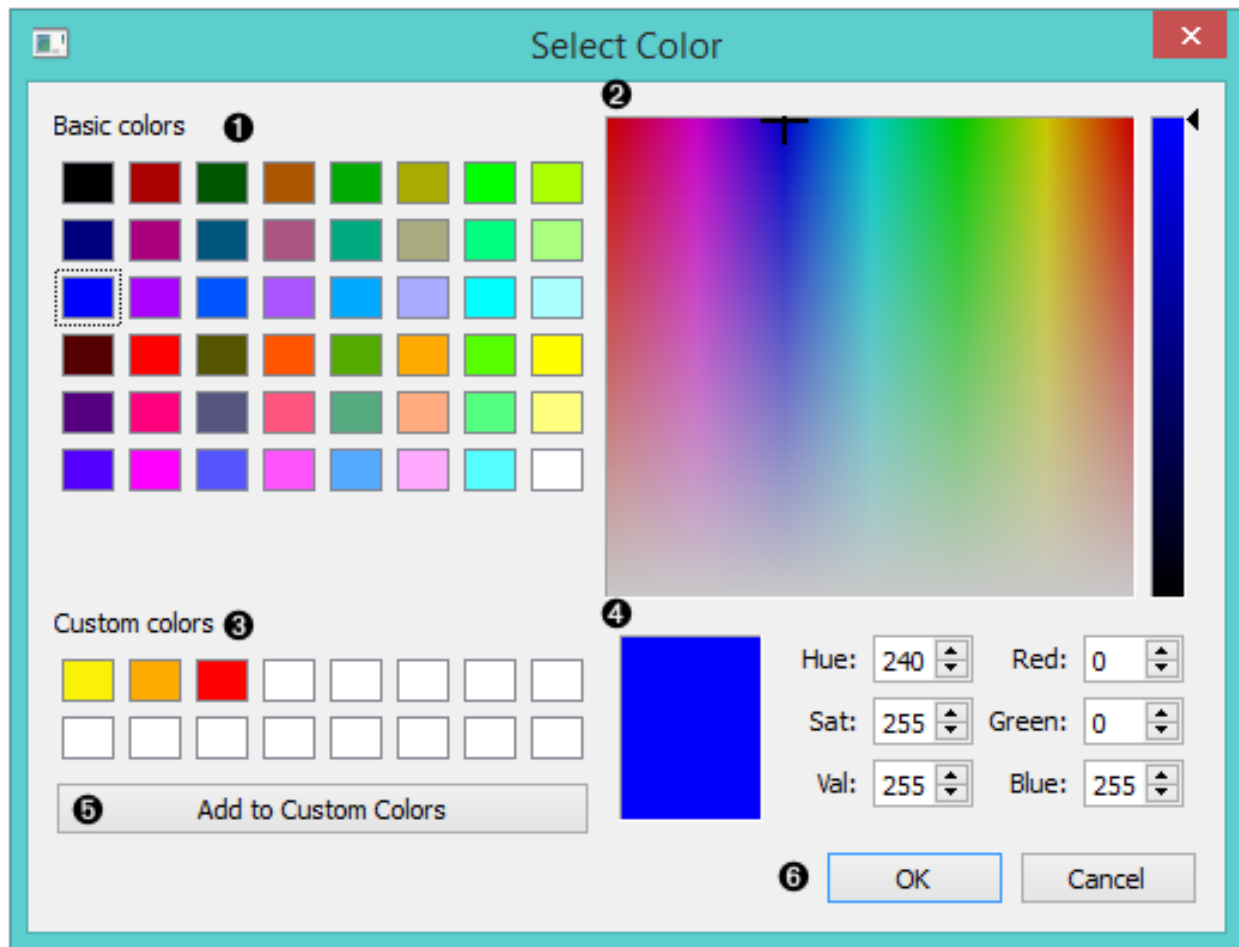
1. Choose a gradient from your saved profiles. The default profile is already set.
2. The gradient palette
3. Select the left side of the gradient. Double clicking the color opens the *Select Color* window.
4. Select the right side of the gradient. Double clicking the color opens the *Select Color* window.
5. Pass through black.
6. Click *OK* to save your choices or *Cancel* to exit the color palette.

## Example

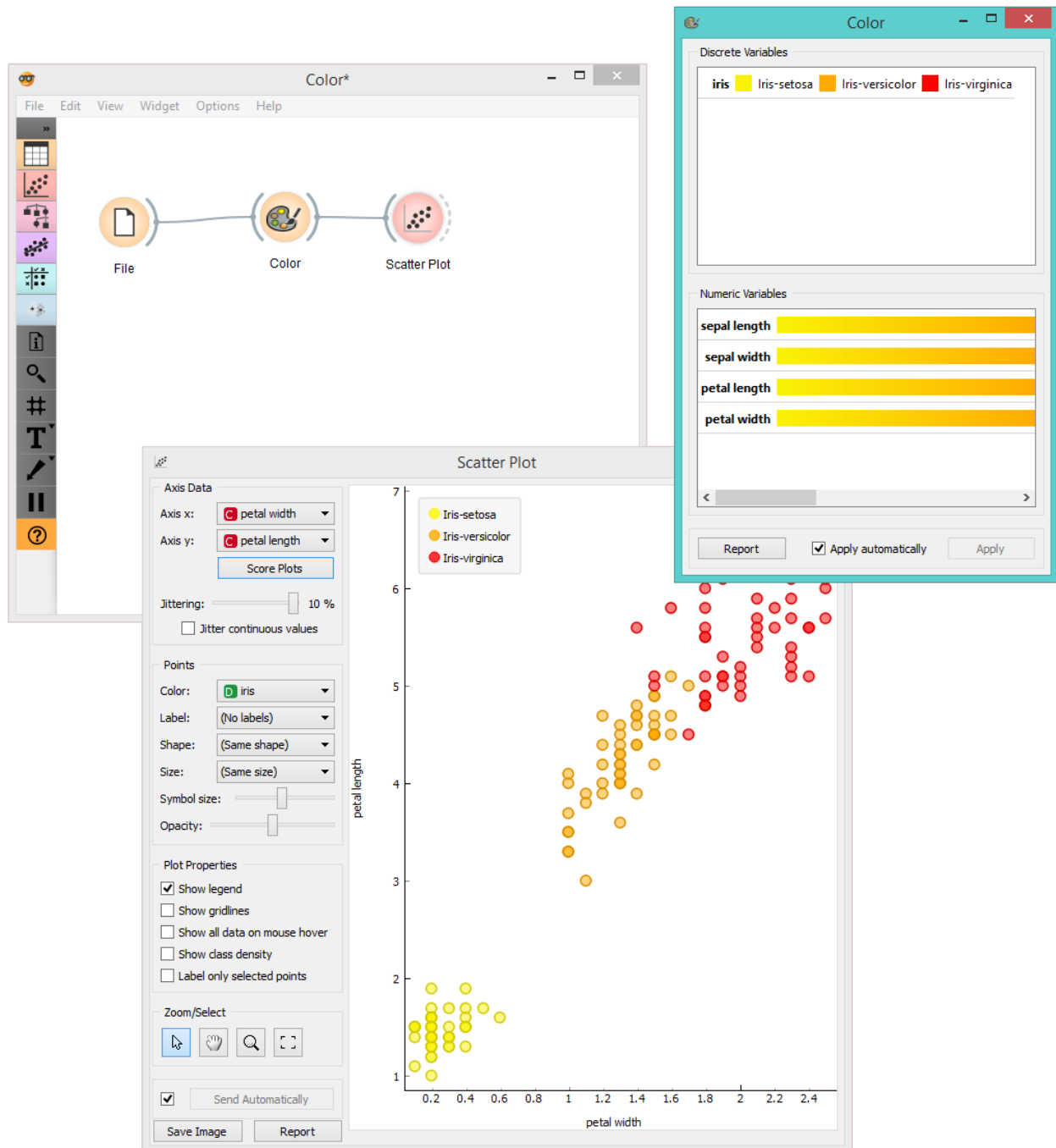
We chose to work with the *Iris* data set. We opened the color palette and selected three new colors for the three types of Irises. Then we opened the *Scatter Plot* widget and viewed the changes made to the scatter plot.

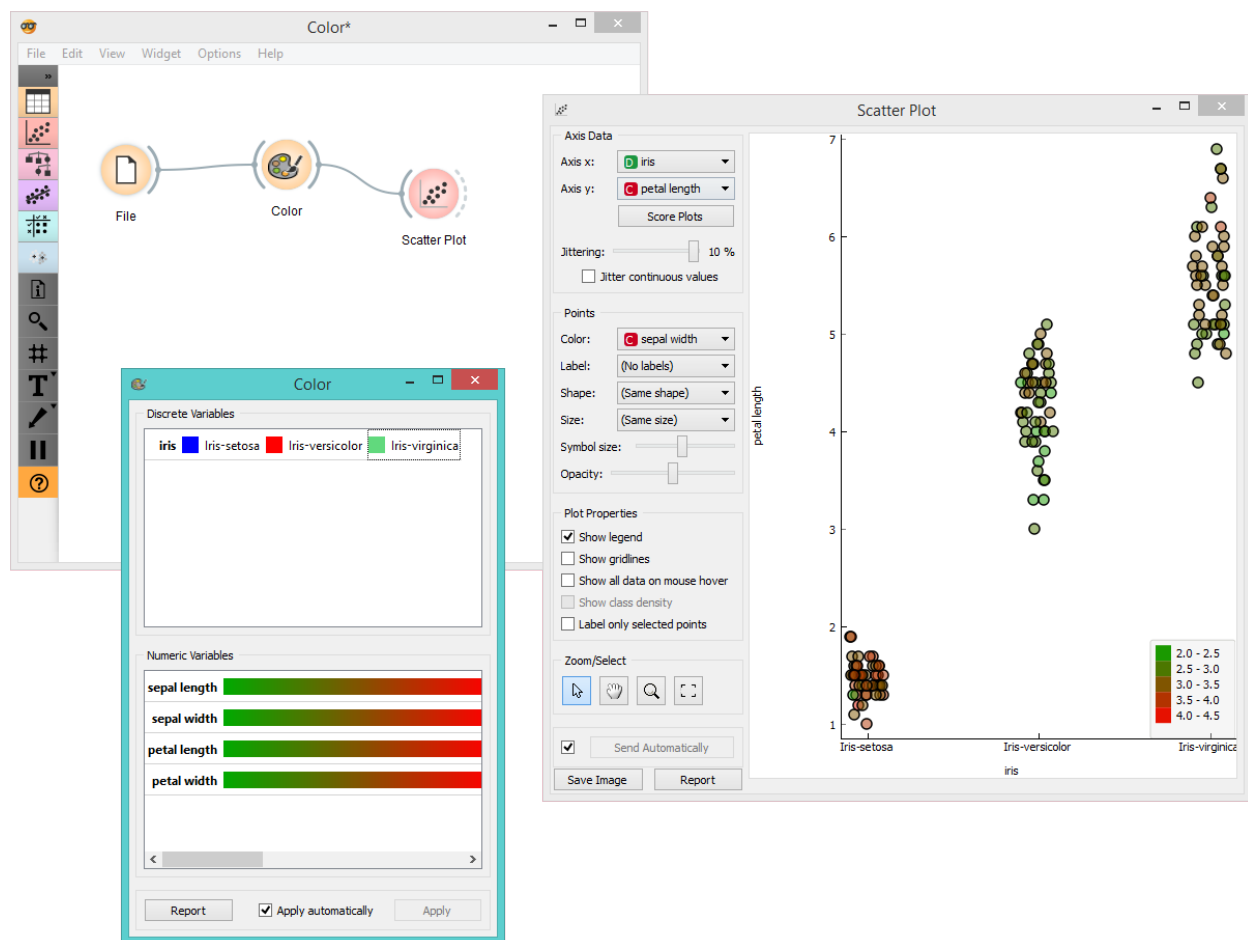
For our second example, we wished to demonstrate the use of the **Color** widget with continuous variables. We put different types of Irises on the x axis and petal length on the y axis. We created a new color gradient and named it *greed* (green + red). In order to show that sepal length is not a deciding factor in differentiating between different types of Irises, we chose to color the points according to sepal width.





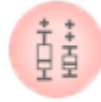






## Visualize

### Box Plot



Shows distribution of attribute values.

### Signals

#### Inputs:

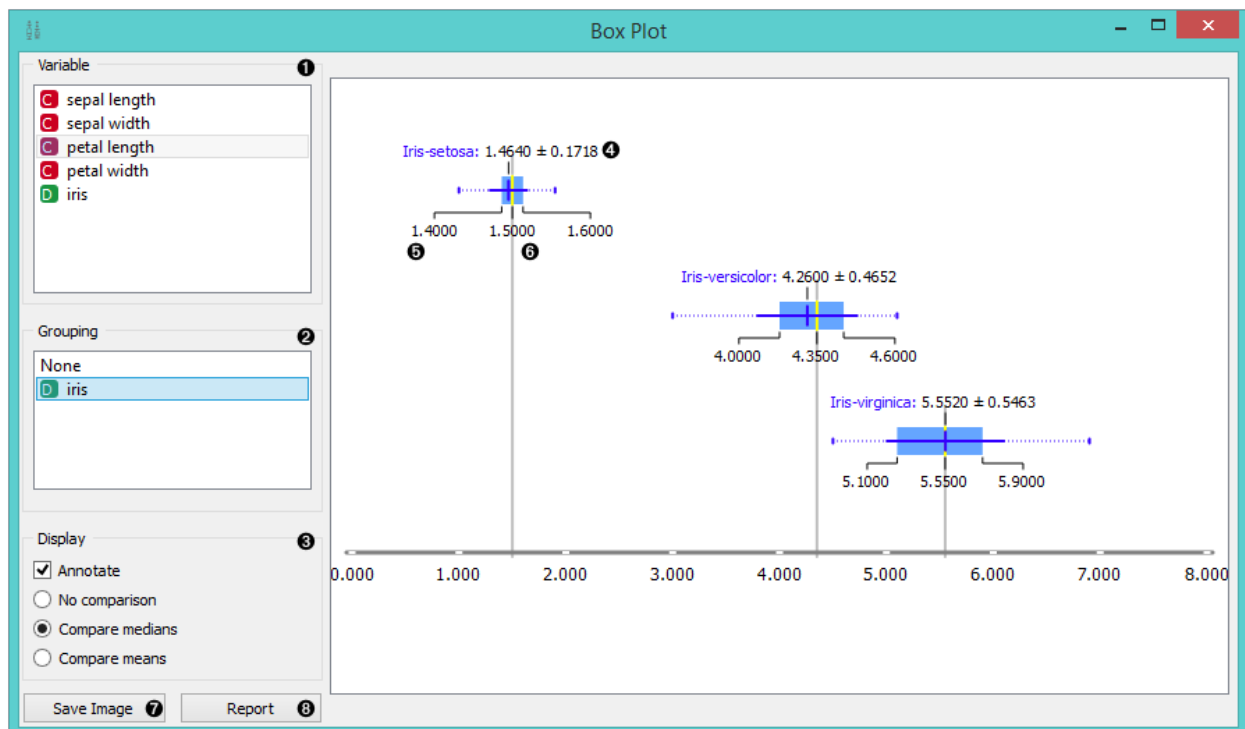
- **Data**  
An input data set

#### Outputs:

- (None)

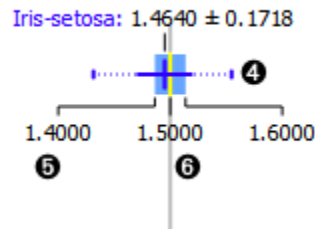
### Description

The **Box Plot** widget shows the distributions of attribute values. It is a good practice to check any new data with this widget to quickly discover any anomalies, such as duplicated values (e.g. gray and grey), outliers, and alike.



1. Select the variable you want to see plotted.

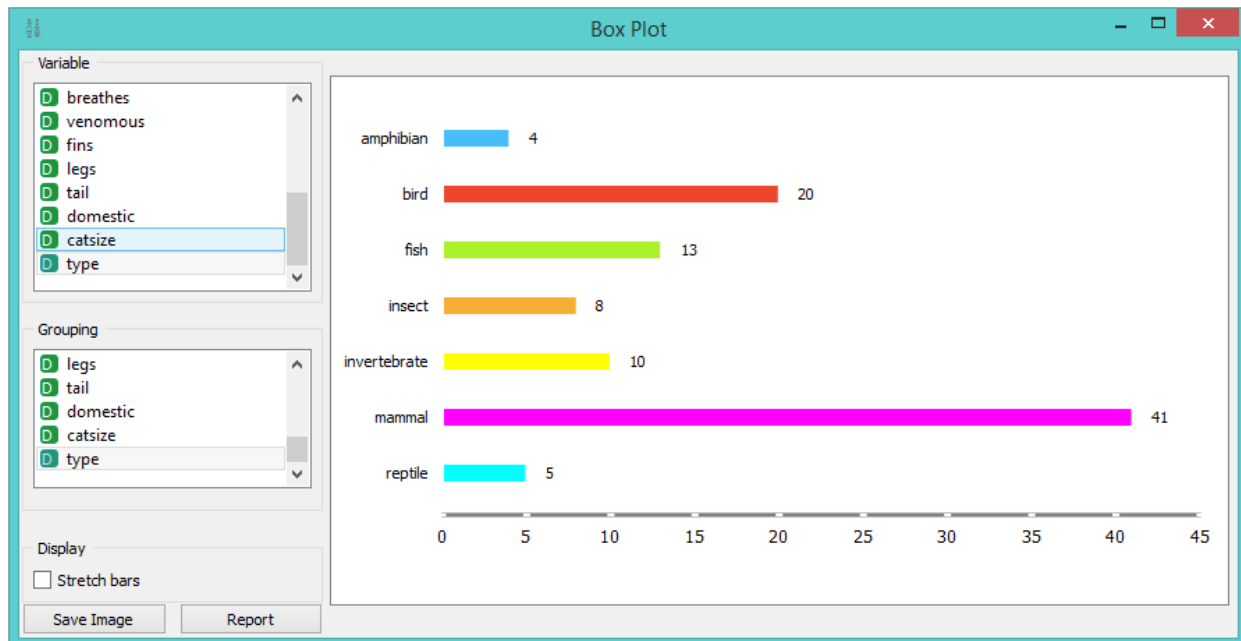
2. Choose *Grouping* to see box plots displayed by class.
3. When instances are grouped by class, you can change the display mode. Annotated boxes will display the end values, the mean and the median, while compare medians and compare means will, naturally, compare the selected value between class groups.



For continuous attributes the widget displays:

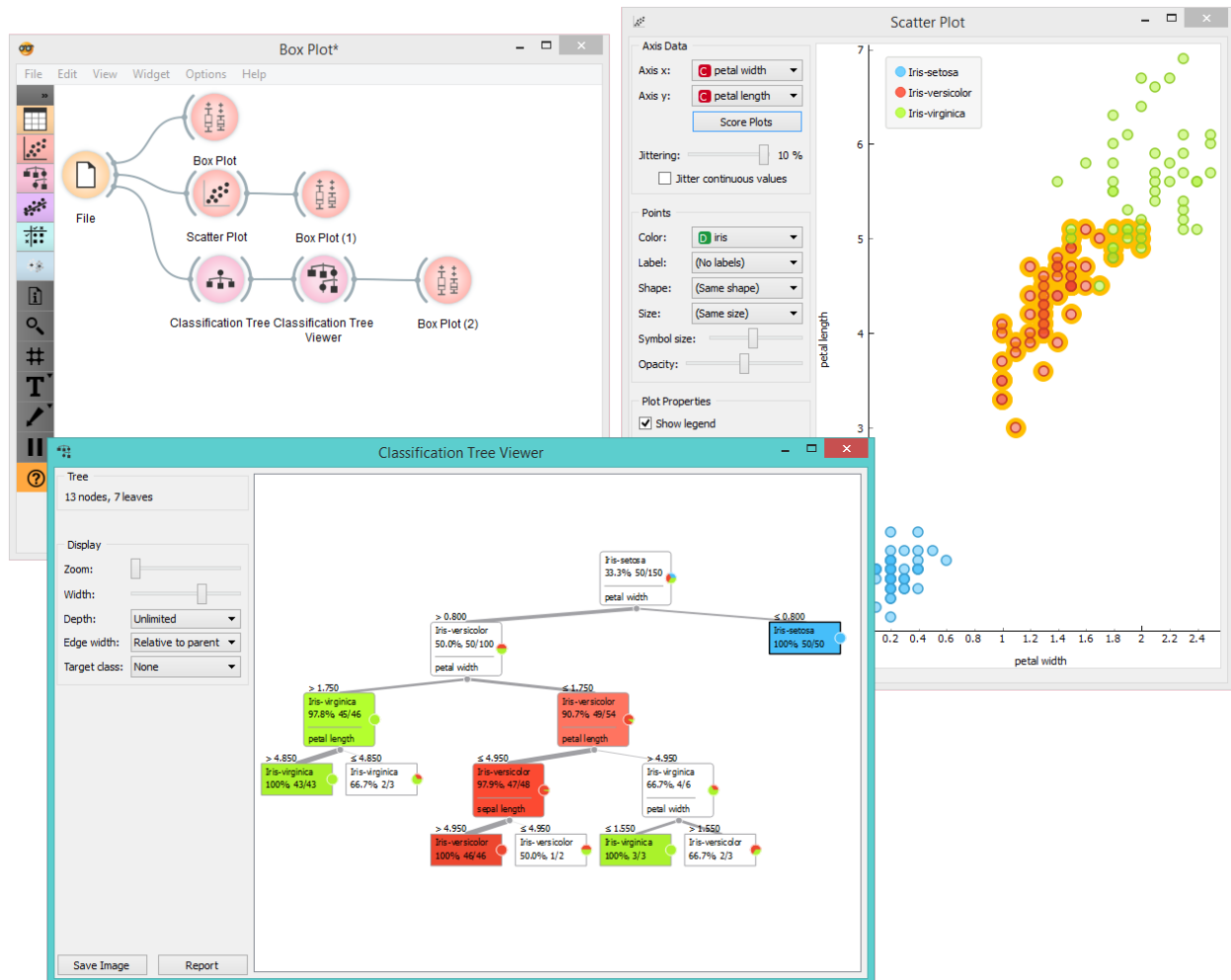
4. The mean (the dark blue vertical line)
5. Border values for the *standard deviation of the mean*. The blue highlighted area is the entire standard deviation of the mean.
6. The median (yellow vertical line). The thin blue line represents the area between the first (25%) and the third (75%) quantile, while the thin dotted line represents the entire range of values (from the lowest to the highest value in the data set for the selected parameter).
7. Save image.
8. Produce a report.

For discrete attributes, the bars represent the number of instances with each particular attribute value. The plot shows the number of different animal types in the *Zoo* data set: there are 41 mammals, 13 fish, 20 birds and so on.



## Example

The **Box Plot** widget is most commonly used immediately after the *File* widget to observe the statistical properties of a data set. It is also useful for finding the properties of a specific data set, for instance a set of instances manually defined in another widget (e.g. *Scatterplot*) or instances belonging to some cluster or a classification tree node, as shown in the schema below.



## Distributions

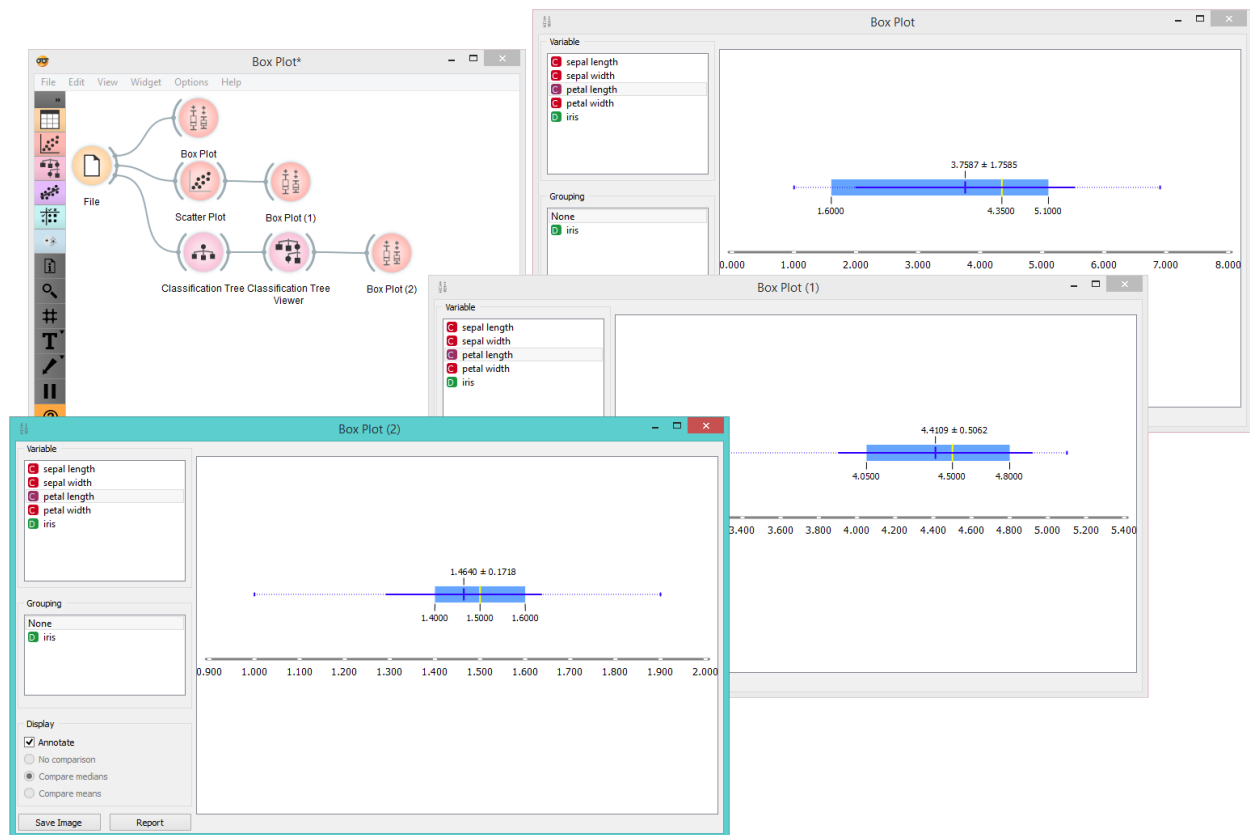
Displays value distributions for a single attribute.

## Signals

### Inputs:

- **Data**  
An input data set.

### Outputs:

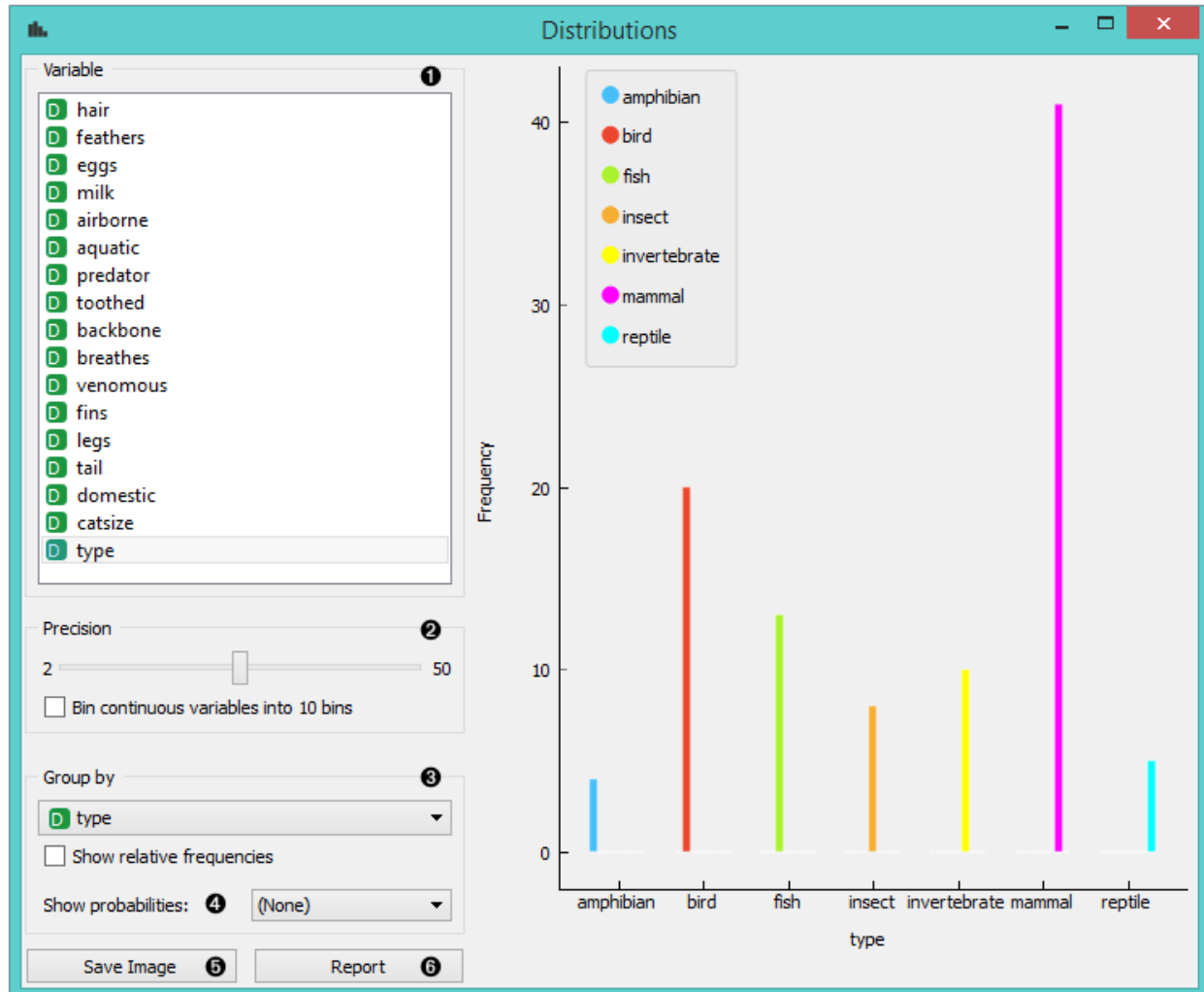


- (None)

## Description

The **Distributions** widget displays the [value distribution](#) of discrete or continuous attributes. If the data contains a class variable, distributions may be conditioned on the class.

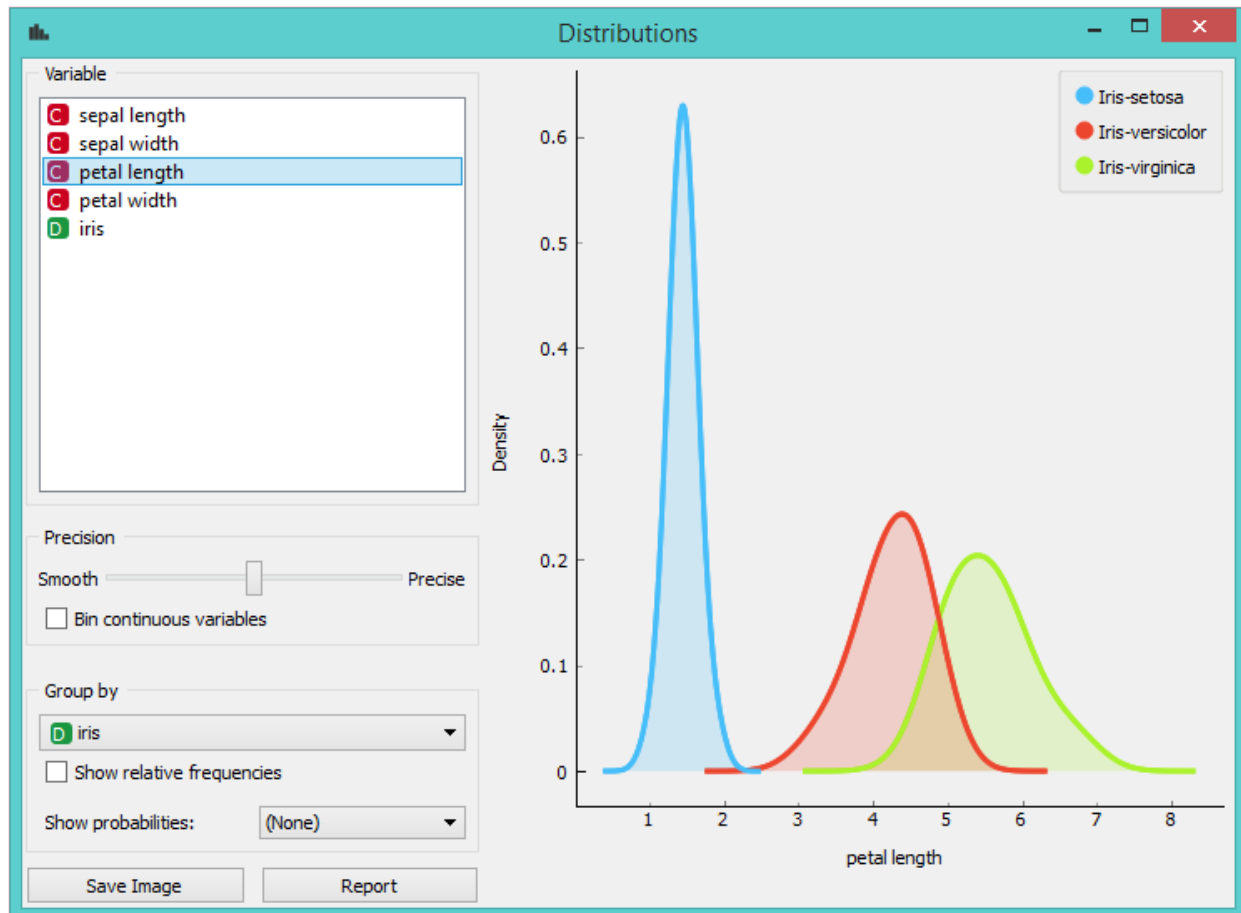
For discrete attributes, the graph displayed by the widget shows how many times (e.g., in how many instances) each attribute value appears in the data. If the data contains a class variable, class distributions for each of the attribute values will be displayed as well (like in the snapshot below). In order to create this graph, we used the *Zoo* data set.



1. A list of variables for distributions display
2. If *Bin continuous variables* is ticked, the widget will discretize continuous variables by assigning them to intervals. The number of intervals is set by precision scale. Alternatively, you can set smoothness for the distribution curves of continuous variables.
3. The widget may be requested to display value distributions only for instances of certain class (*Group by*). *Show relative frequencies* will scale the data by percentage of the data set.
4. Show probabilities.

5. *Save image* saves the graph to your computer in a .svg or .png format.
6. Produce a report.

For continuous attributes, the attribute values are displayed as a function graph. Class probabilities for continuous attributes are obtained with gaussian kernel density estimation, while the appearance of the curve is set with the *Precision* bar (smooth or precise). For the purpose of this example, we used the *Iris* data set.



In class-less domains, the bars are displayed in gray. Here we set *Bin continuous variables into 10 bins*, which distributes variables into 10 intervals and displays averages of these intervals as histograms (see 2. above). We used the *Housing* data set.

## Heat Map

Plots a heat map for a pair of attributes.

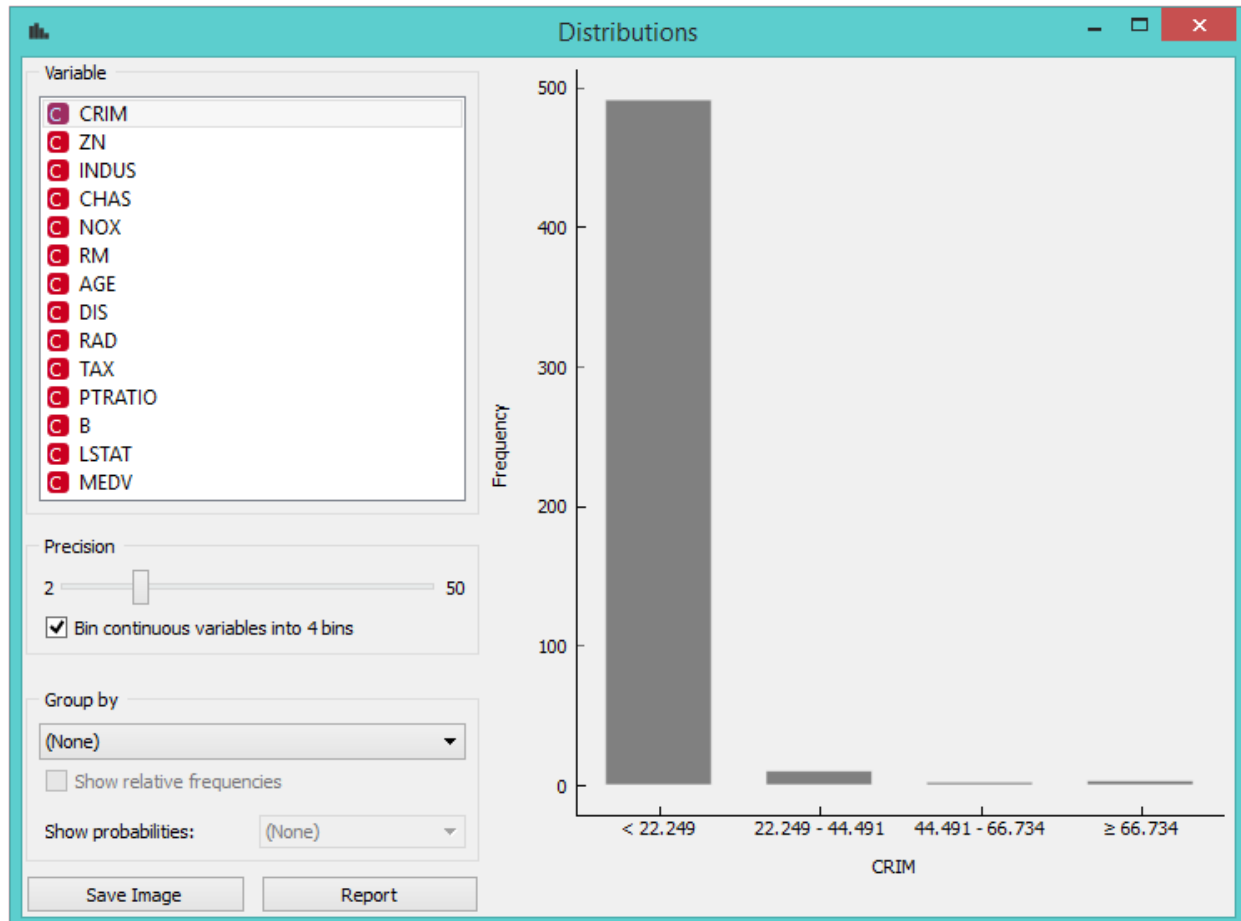
## Signals

### Inputs:

- **Data**  
An input data set.

### Outputs:





- **Selected Data**

A subset of instances that the user has manually selected from the map.

### Description

**Heat map** is a graphical method for visualizing attribute values by class in a two-way matrix. It only works on data sets containing continuous variables. The values are represented by color: the higher a certain value is, the darker the represented color. By combining class and attributes on x and y axes, we see where the attribute values are the strongest and where the weakest, thus enabling us to find typical features (discrete) or value range (continuous) for each class.

1. The color scheme legend. **Low** and **High** are thresholds for the color palette (low for attributes with low values and high for attributes with high values).
2. Merge data.
3. Sort columns and rows: - **No Sorting** (lists attributes as found in the data set) - **Clustering** (clusters data by similarity) - **Clustering with ordered leaves** (maximizes the sum of similarities of adjacent elements)
4. Set what is displayed in the plot in **Annotation & Legend**. - If *Show legend* is ticked, a color chart will be displayed above the map. - If *Stripes with averages* is ticked, a new line with attribute averages will be displayed on the left. - **Row Annotations** adds annotations to each instance on the right. - **Column Label Positions** places column labels in a selected place (None, Top, Bottom, Top and Bottom).
5. If *Keep aspect ratio* is ticked, each value will be displayed with a square (proportionate to the map).
6. If *Send Automatically* is ticked, changes are communicated automatically. Alternatively, click *Send*.
7. *Save image* saves the image to your computer in a .svg or .png format.
8. Produce a report.

### Example

The **Heat Map** below displays attribute values for the *Housing* data set. The aforementioned data set concerns the housing values in the suburbs of Boston. The first thing we see in the map are the 'B' and 'Tax' attributes, which are the only two colored in dark orange. The 'B' attribute provides information on the proportion of blacks by town and the 'Tax' attribute informs us about the full-value property-tax rate per \$10,000. In order to get a clearer heat map, we then use the *Select Columns* widget and remove the two attributes from the data set. Then we again feed the data to the **Heat map**. The new projection offers additional information. By removing 'B' and 'Tax', we can see other deciding factors, namely 'Age' and 'ZN'. The 'Age' attribute provides information on the proportion of owner-occupied units built prior to 1940 and the 'ZN' attribute informs us about the proportion of non-retail business acres per town.

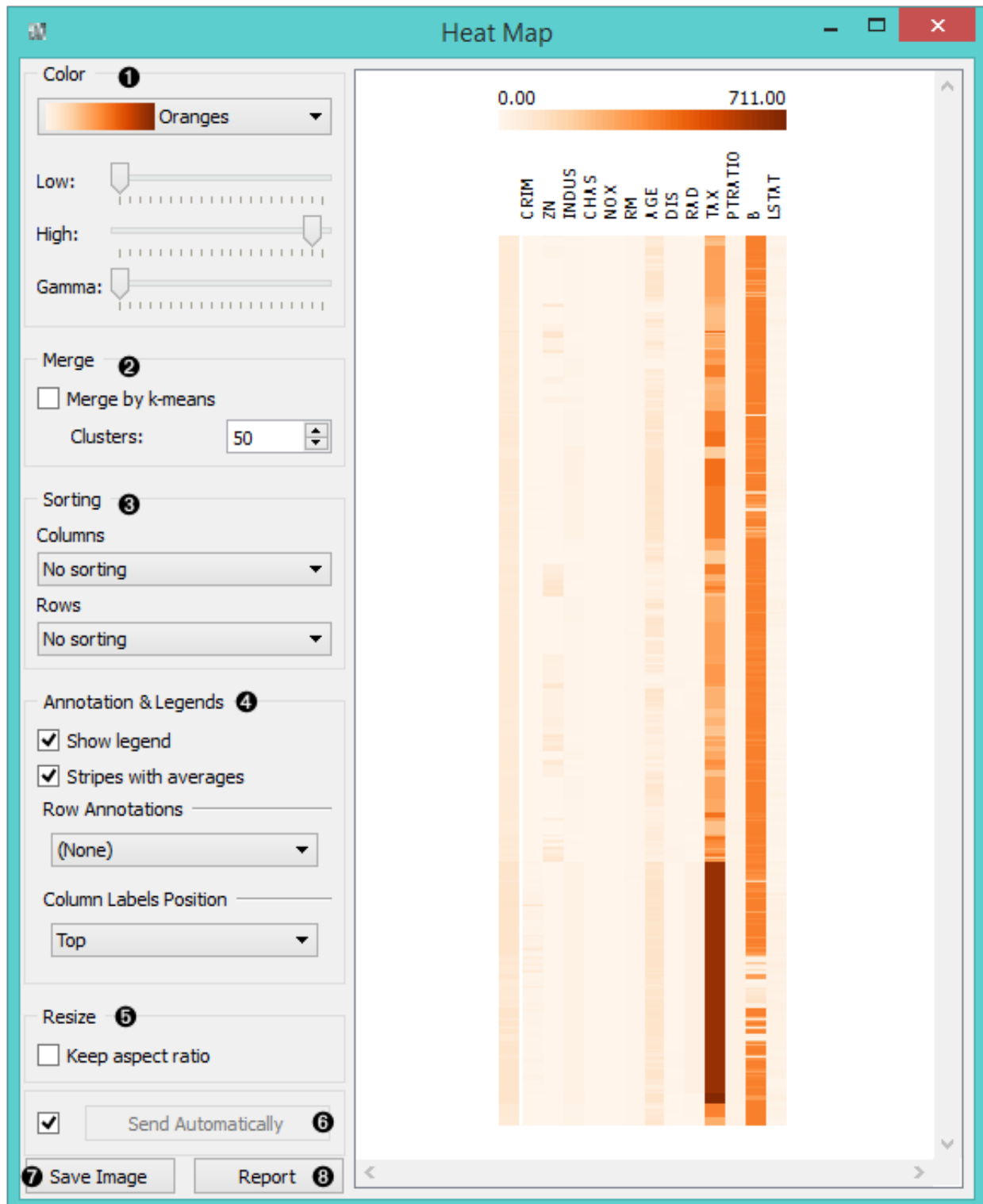
The **Heat Map** widget is a nice tool for discovering relevant features in the data. By removing some of the more pronounced features, we came across new information, which was hiding in the background.

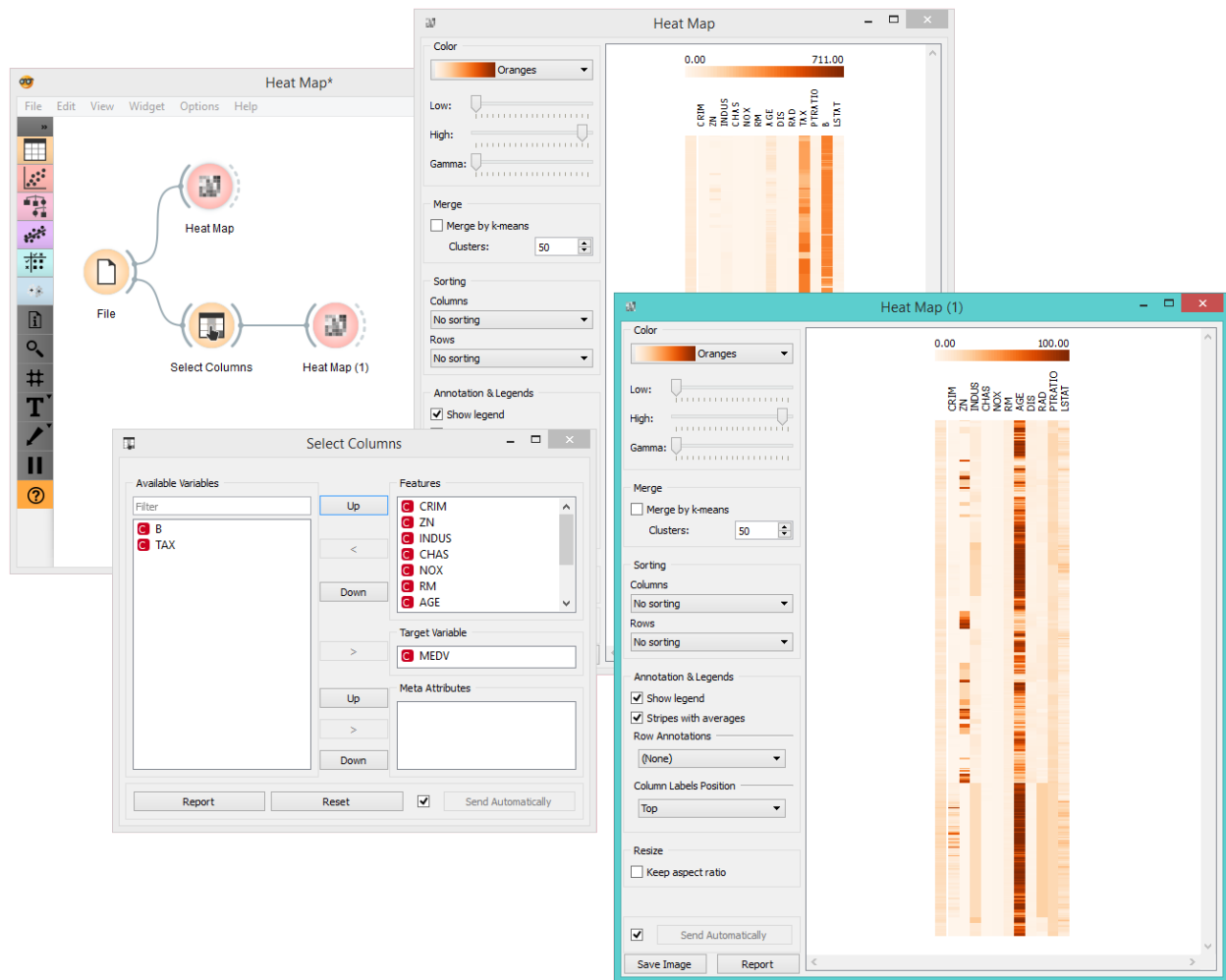
### References

[Housing Data Set](#)

### Scatter Plot

Scatterplot visualization with explorative analysis and intelligent data visualization enhancements.





## Signals

### Inputs:

- **Data**  
An input data set.
- **Data Subset**  
A subset of instances from the input data set.
- **Features**  
A list of attributes.

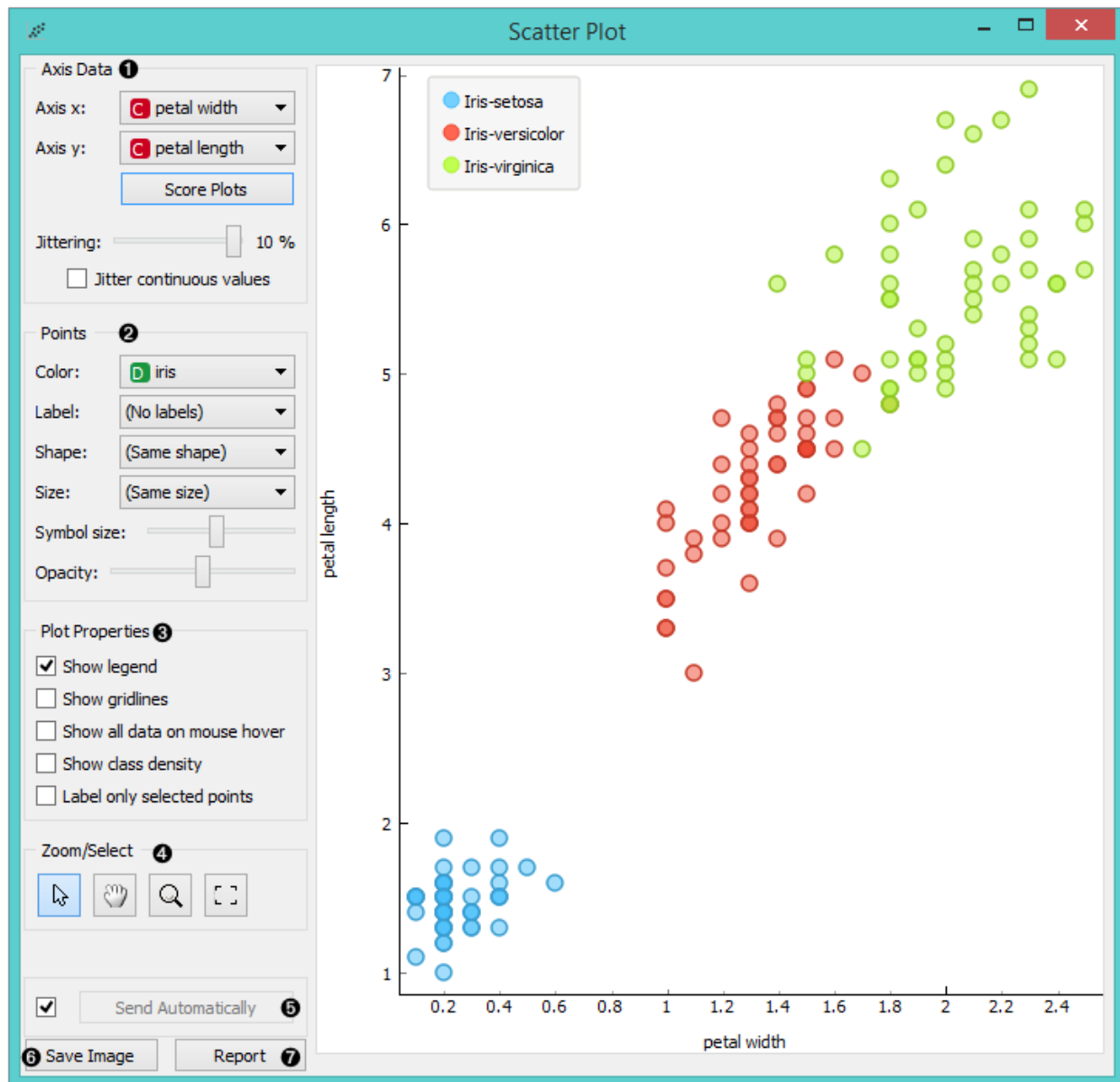
### Outputs:

- **Selected Data**  
A subset of instances that the user manually selected from the scatterplot.
- **Unselected Data**  
All other data (instances not included in the user's selection).

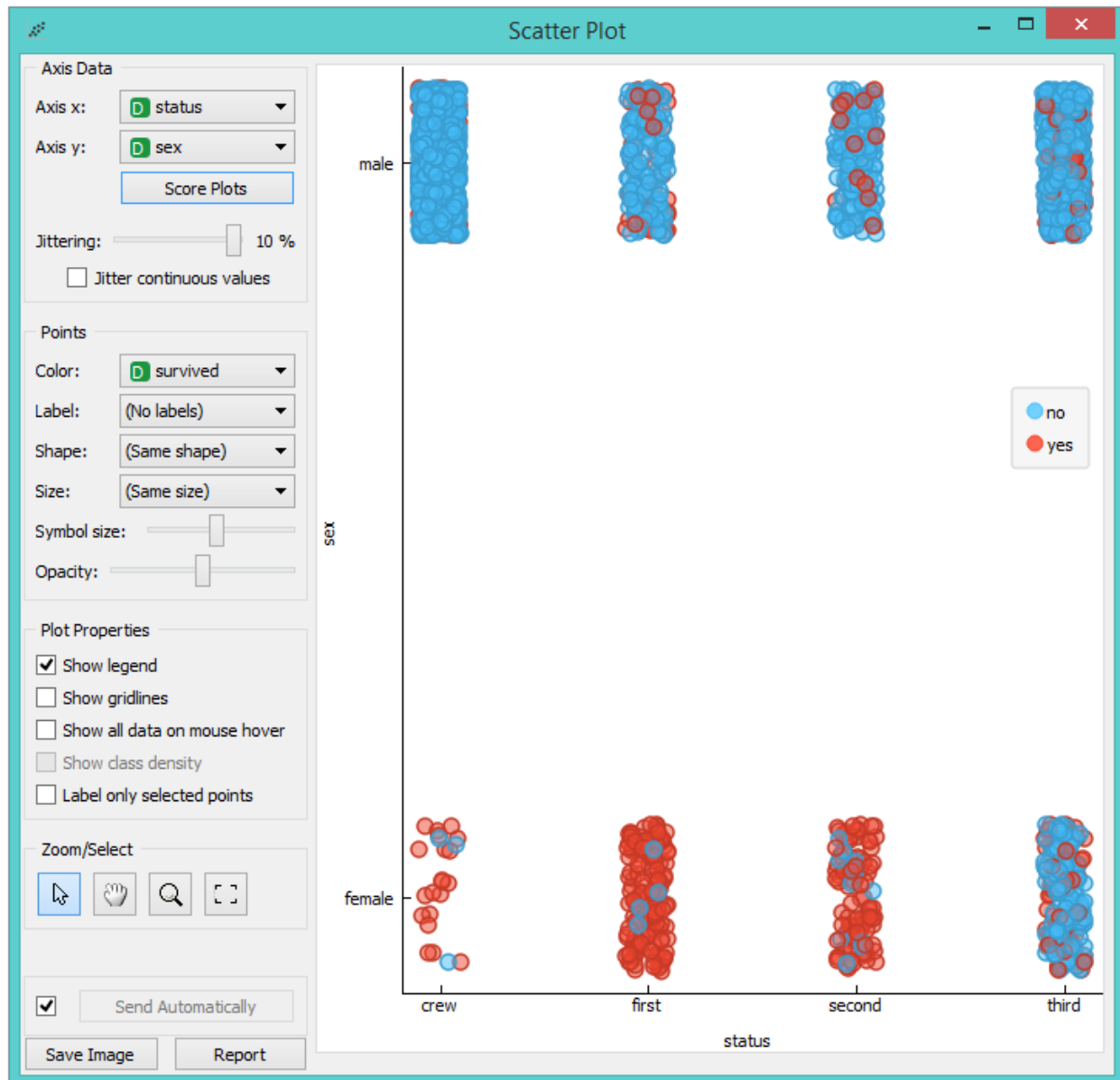
## Description

The **Scatterplot** widget provides a 2-dimensional scatterplot visualization for both continuous and discrete-valued attributes. The data is displayed as a collection of points, each having the value of the x-axis attribute determining the position on the horizontal axis and the value of the y-axis attribute determining the position on the vertical axis. Various properties of the graph, like color, size and shape of the points, axis titles, maximum point size and jittering can be adjusted on the left side of the widget. A snapshot below shows the scatterplot of the *Iris* data set with the coloring matching of the class attribute.

1. Select the x and y attribute. Optimize your projection by using **Rank Projections**. This feature scores attribute pairs by average classification accuracy and returns the top scoring pair with a simultaneous visualization update. Set **jittering** to prevent the dots overlapping. If *Jitter continuous values* is ticked, continuous instances will be dispersed.
2. Set the color of the displayed points (you will get colors for discrete values and grey-scale points for continuous). Set label, shape and size to differentiate between points. Set symbol size and opacity for all data points. Set the desired colors scale.
3. Adjust *plot properties*:
  - *Show legend* displays a legend on the right. Click and drag the legend to move it.
  - *Show gridlines* displays the grid behind the plot.
  - *Show all data on mouse hover* enables information bubbles if the cursor is placed on a dot.
  - *Show class density* colors the graph by class (see the screenshot below).
  - *Label only selected points* allows you to select individual data instances and label them.
4. *Select*, *zoom*, *pan* and *zoom to fit* are the options for exploring the graph. The manual selection of data instances works as an angular/square selection tool. Double click to move the projection. Scroll in or out for zoom.
5. If *Send automatically* is ticked, changes are communicated automatically. Alternatively, press *Send*.
6. *Save Image* saves the created image to your computer in a .svg or .png format.
7. Produce a report.



For discrete attributes, jittering circumvents the overlap of points which have the same value for both axes, and therefore the density of points in the region corresponds better to the data. As an example, the scatterplot for the Titanic data set, reporting on the gender of the passengers and the traveling class is shown below; without jittering, the scatterplot would display only eight distinct points.

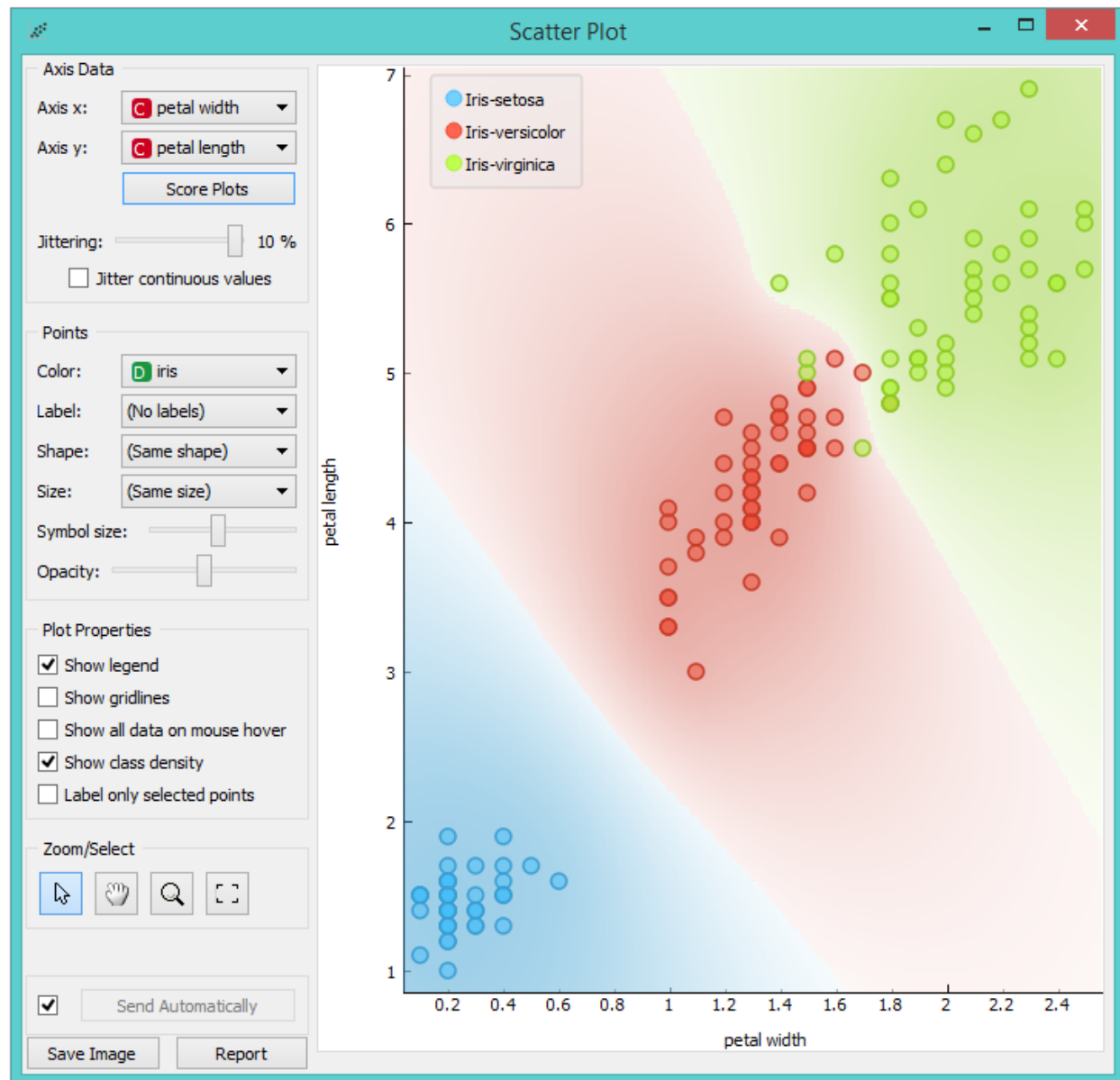


Here is an example of the **Scatter Plot** widget if the *Show class density* box is ticked.

## Intelligent Data Visualization

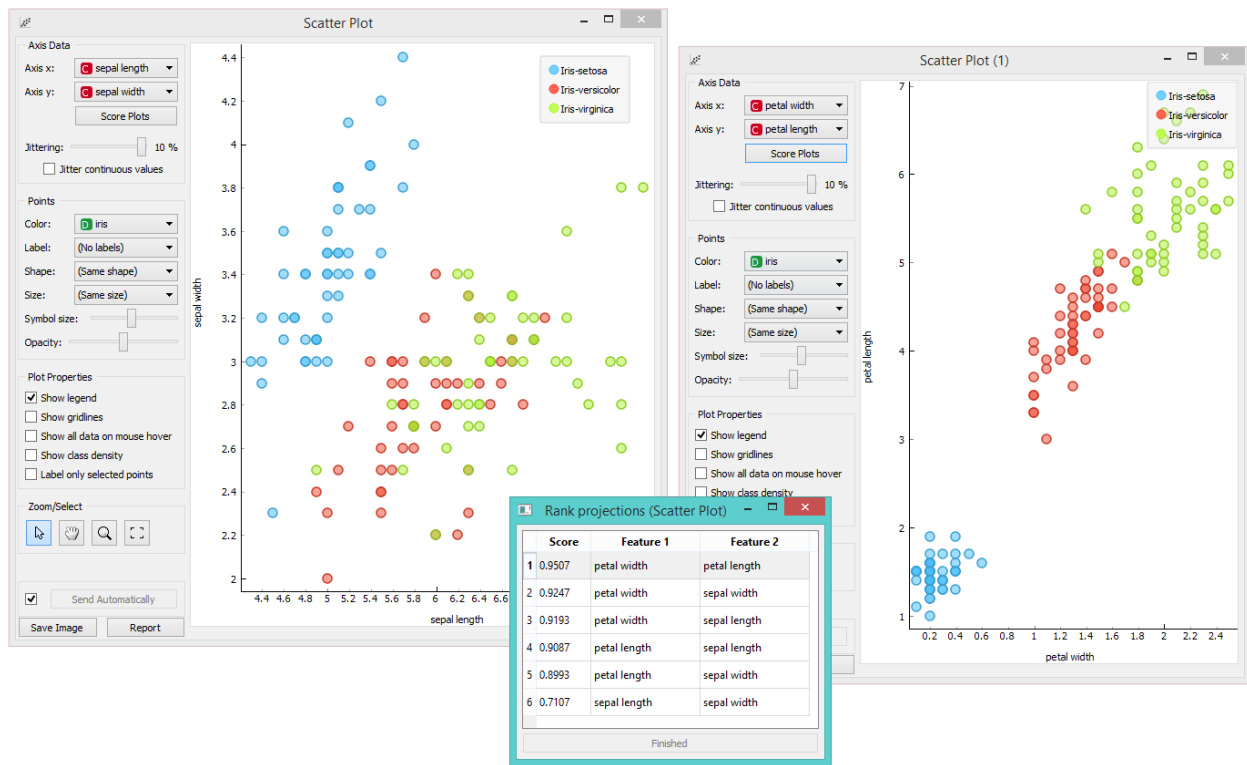
If a data set has many attributes, it is impossible to manually scan through all the pairs to find interesting or useful scatterplots. Orange implements intelligent data visualization with the **Score Plots** option in the widget. The goal of optimization is to find scatterplot projections where instances are well separated.

To use this method, go to the *Score Plots* option in the widget, open the subwindow and press *Start Evaluation*. The feature will return a list of attribute pairs by average classification accuracy score.





Below, there is an example demonstrating the utility of ranking. The first scatterplot projection was set as the default sepal width to sepal length plot (we used the Iris data set for simplicity). Upon running *Score Plots* optimization, the scatterplot converted to a much better projection of petal width to petal length plot.



## Explorative Data Analysis

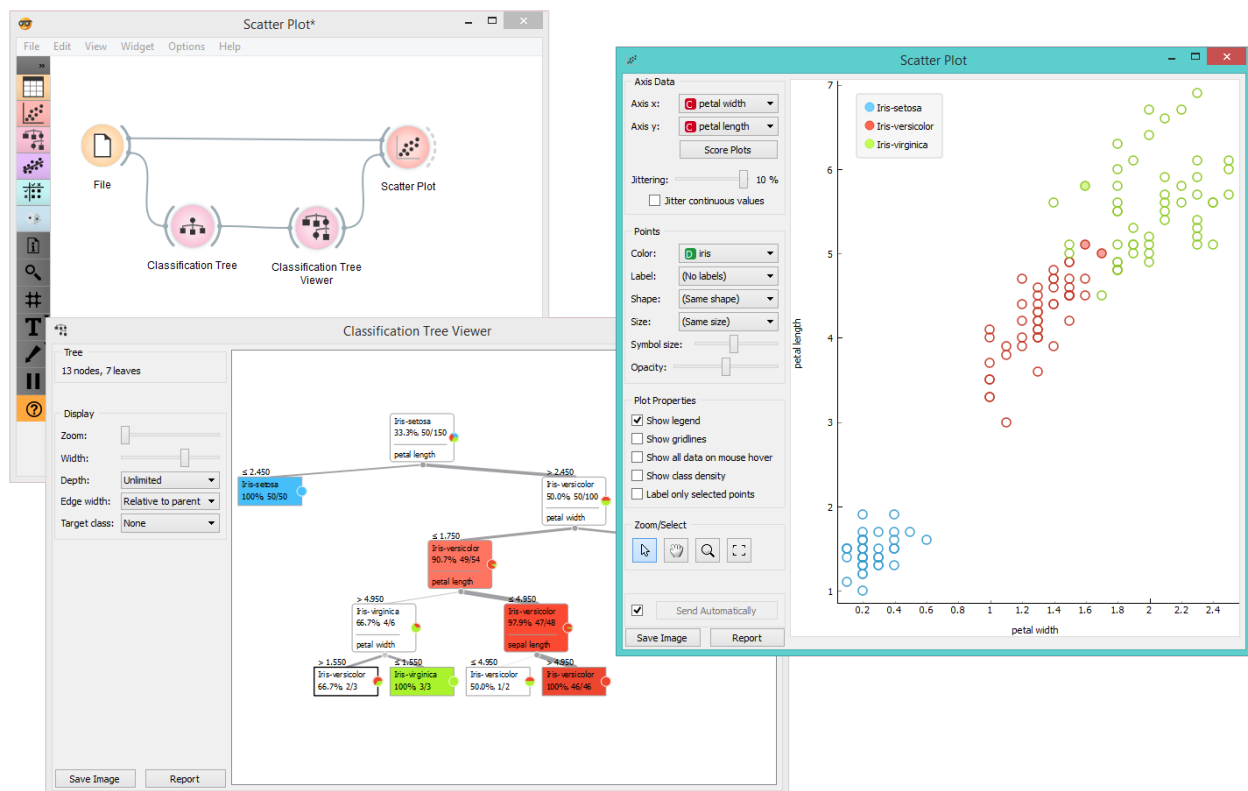
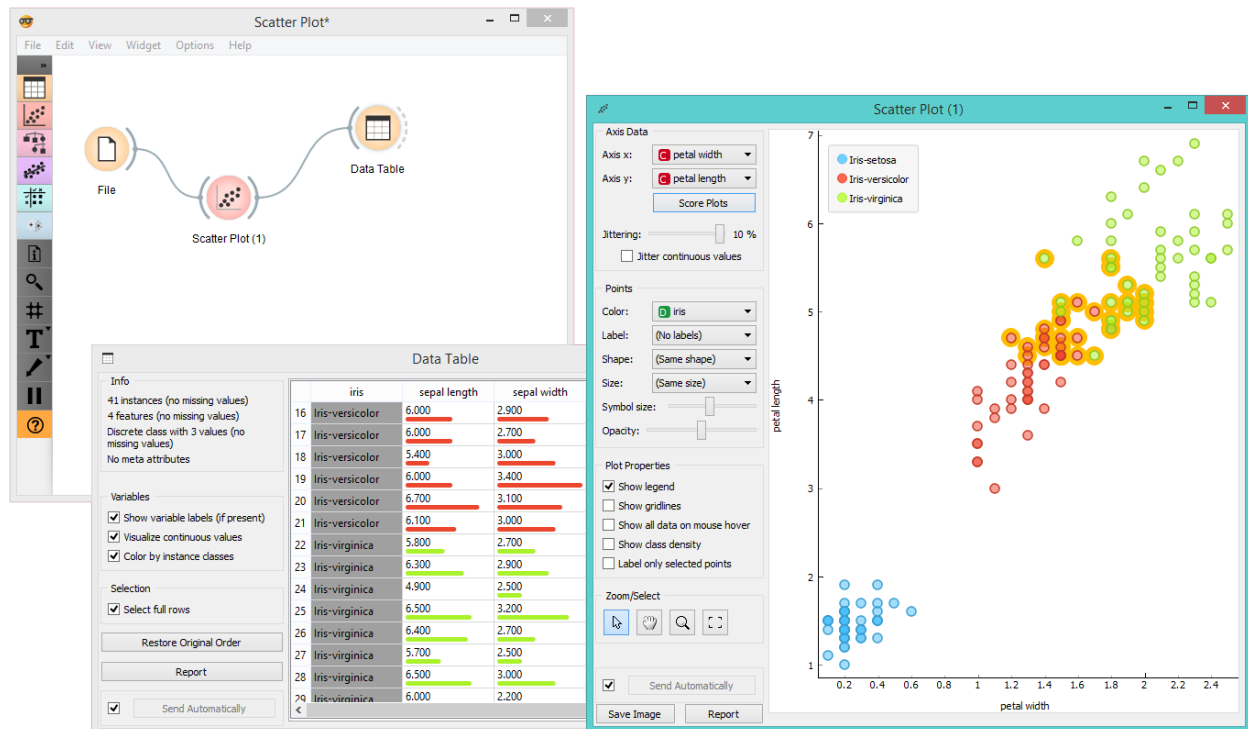
The **Scatterplot**, as the rest of Orange widgets, supports zooming-in and out of part of the plot and a manual selection of data instances. These functions are available in the lower left corner of the widget. The default tool is *Select*, which selects data instances within the chosen rectangular area. *Pan* enables you to move the scatterplot around the pane. With *Zoom* you can zoom in and out of the pane with a mouse scroll, while *Reset zoom* resets the visualization to its optimal size. An example of a simple schema, where we selected data instances from a rectangular region and sent them to the *Data Table* widget, is shown below. Notice that the scatterplot doesn't show all 52 data instances, because some data instances overlap (they have the same values for both attributes used).

## Example

The **Scatterplot** can be combined with any widget that outputs a list of selected data instances. In the example below, we combine *Classification Tree* and **Scatterplot** to display instances taken from a chosen classification tree node (clicking on any node of the classification tree will send a set of selected data instances to the scatterplot and mark selected instances with filled symbols).

## Venn Diagram

Plots a *Venn diagram* for two or more data subsets.



## Signals

### Inputs:

- **Data**

An input data set

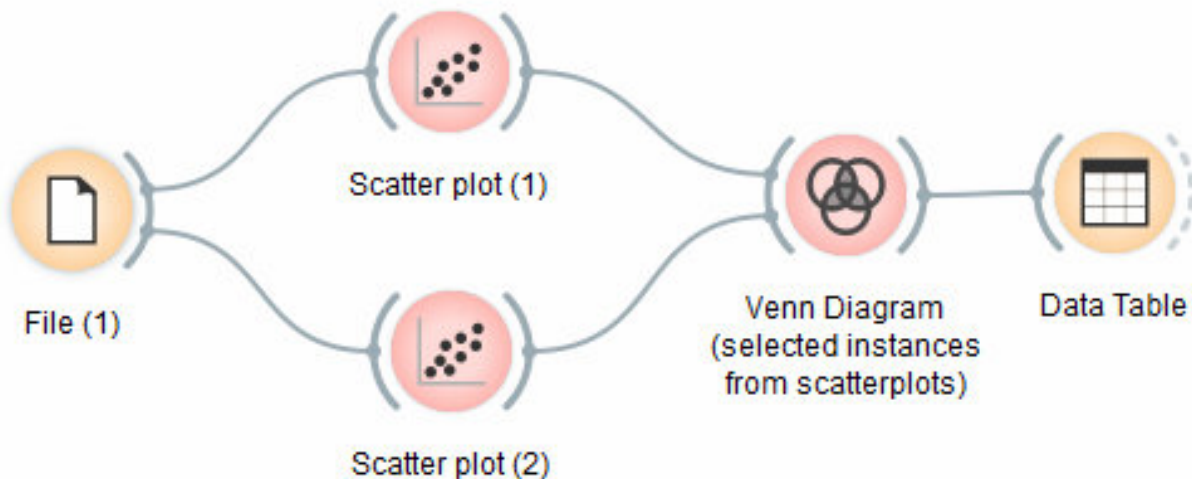
### Outputs:

- **Selected Data**

A subset of instances that the user has manually selected from the diagram.

## Description

The **Venn Diagram** widget displays logical relations between data sets. This projection shows two or more data sets represented by circles of different colors. The intersections are subsets that belong to more than one data set. To further analyze or visualize the subset, click on the intersection.

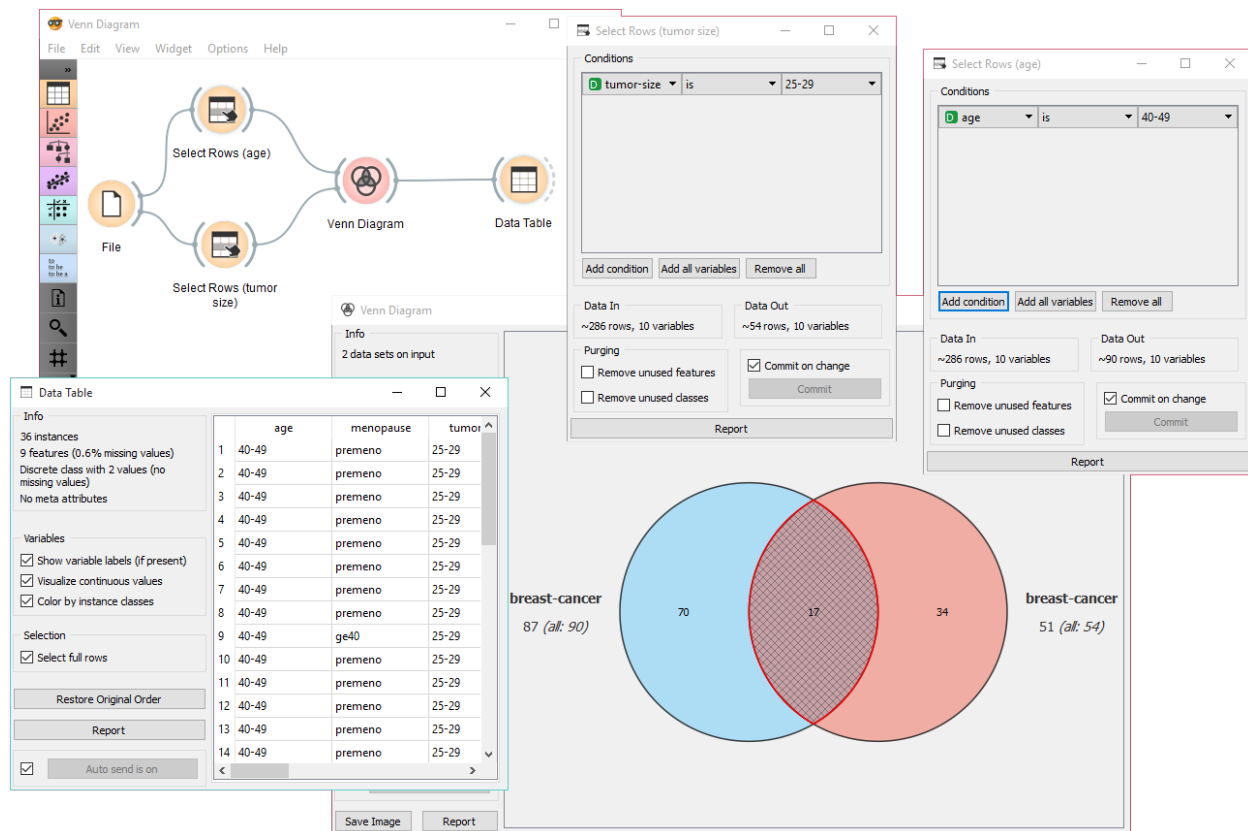


1. Information on the input data.
2. Select the identifiers by which to compare the data.
3. Tick *Output duplicates* if you wish to remove duplicates.
4. If *Auto commit* is on, changes are automatically communicated to other widgets. Alternatively, click *Commit*.
5. *Save Image* saves the created image to your computer in a .svg or .png format.
6. Produce a report.

## Examples

The easiest way to use the **Venn Diagram** is to select data subsets and find matching instances in the visualization. We use the *breast-cancer* data set to select two subsets with *Select Rows* widget - the first subset is that of breast cancer patients aged between 40 and 49 and the second is that of patients with a tumor size between 20 and 29. The **Venn Diagram** helps us find instances that correspond to both criteria, which can be found in the intersection of the two circles.





The **Venn Diagram** widget can be also used for exploring different prediction models. In the following example, we analysed 3 prediction methods, namely *Naive Bayes*, *SVM Learner* and *Random Forest Learner*, according to their misclassified instances. By selecting misclassifications in the three *Confusion Matrix* widgets and sending them to Venn diagram, we can see all the misclassification instances visualized per method used. Then we open **Venn Diagram** and select, for example, the misclassified instances that were identified by all three methods (in our case 2). This is represented as an intersection of all three circles. Click on the intersection to see this two instances marked in the *Scatterplot* widget. Try selecting different diagram sections to see how the scatterplot visualization changes.

## Linear Projection

A linear projection method with explorative data analysis.

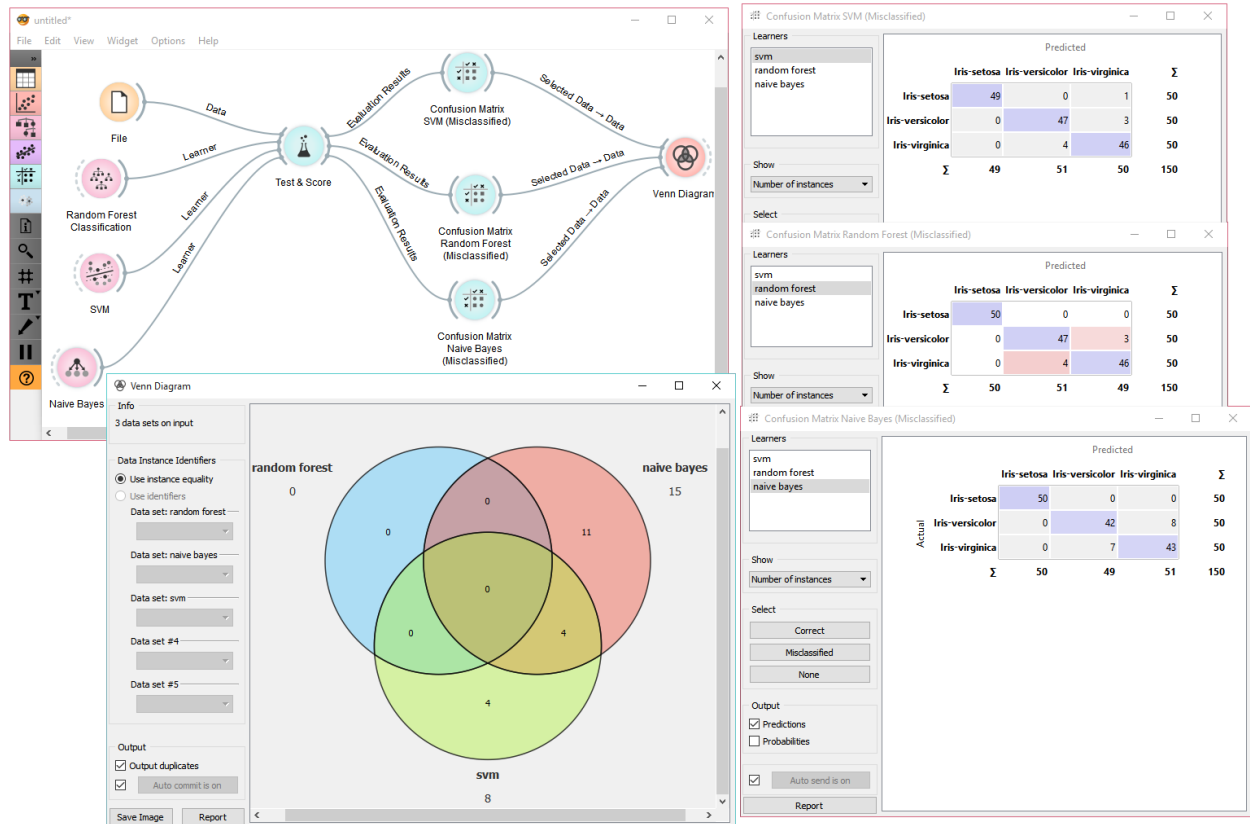
## Signals

### Inputs:

- **Data**  
An input data set
- **Data Subset**  
A subset of data instances

### Outputs:

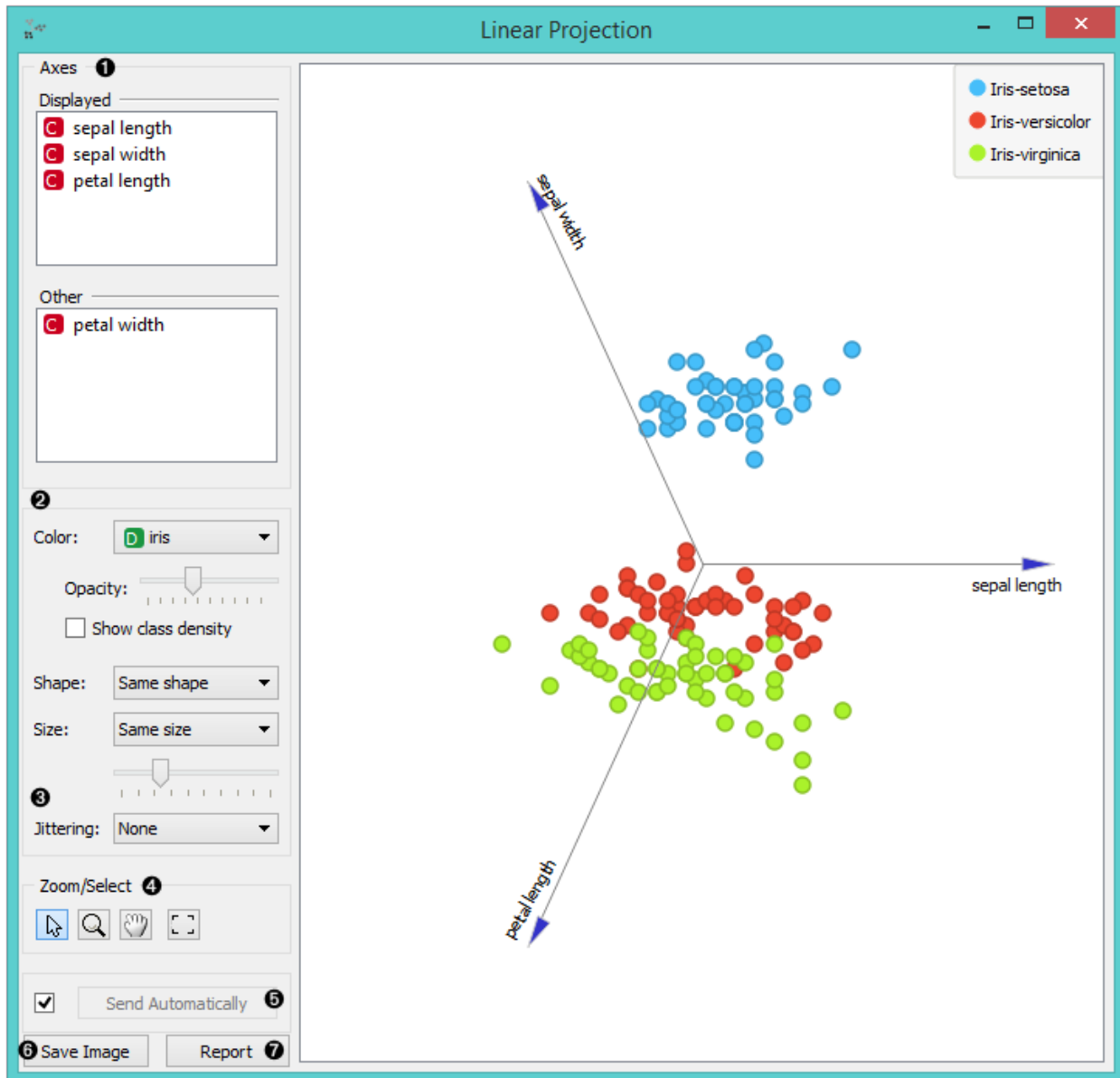
- **Selected Data**



A data subset that the user has manually selected in the projection.

## Description

This widget displays [linear projections](#) of class-labeled data. Consider, for a start, a projection of the *Iris* data set shown below. Notice that it is the sepal width and sepal length that already separate *Iris setosa* from the other two, while the petal length is the attribute best separating *Iris versicolor* from *Iris virginica*.



1. Axes in the projection that are displayed and other available axes.
2. Set the color of the displayed dots (you will get colored dots for discrete values and grey-scale dots for continuous). Set opacity, shape and size to differentiate between instances.
3. Set [jittering](#) to prevent the dots from overlapping (especially for discrete attributes).

4. *Select*, *zoom*, *pan* and *zoom to fit* options for exploring the graph. Manual selection of data instances works as a non-angular/free-hand selection tool. Double click to move the projection. Scroll in or out for zoom.
5. When the box is ticked (*Auto commit is on*), the widget will communicate the changes automatically. Alternatively, click *Commit*.
6. *Save Image* saves the created image to your computer in a .svg or .png format.
7. Produce a report.

## Example

The **Linear Projection** widget works just like other visualization widgets. Below, we connected it to the *File* widget to see the set projected on a 2-D plane. Then we selected the data for further analysis and connected it to the *Data Table* widget to see the details of the selected subset.



## References

Koren Y., Carmel L. (2003). Visualization of labeled data using linear transformations. In Proceedings of IEEE Information Visualization 2003, (InfoVis'03). Available [here](#).

Boulesteix A.-L., Strimmer K. (2006). Partial least squares: a versatile tool for the analysis of high-dimensional genomic data. Briefings in Bioinformatics, 8(1), 32-44. Abstract [here](#).

## Scatter Map

Plots a scatter map for a pair of continuous attributes.





## Signals

### Inputs:

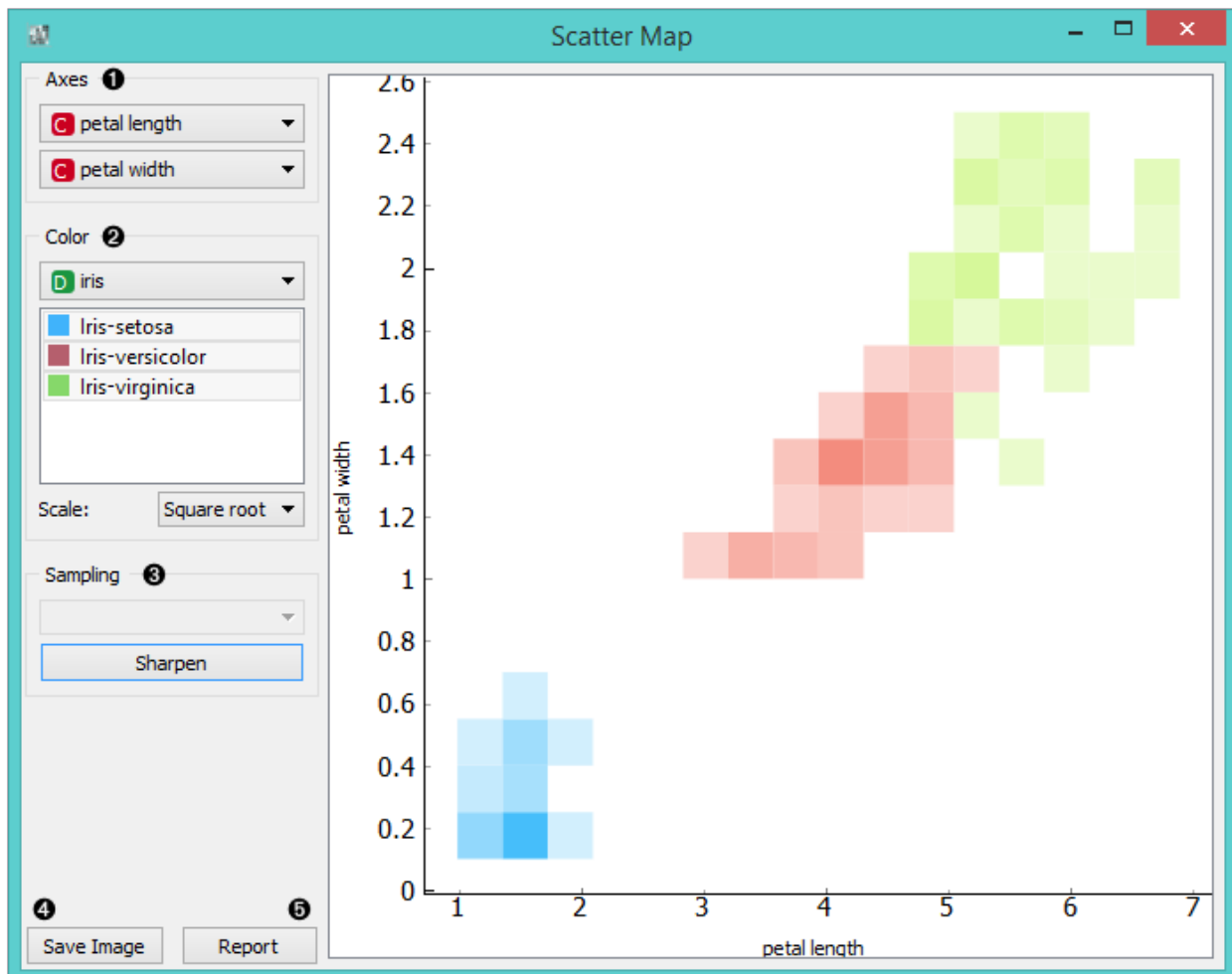
- **Data**  
An input data set

### Outputs:

- None

## Description

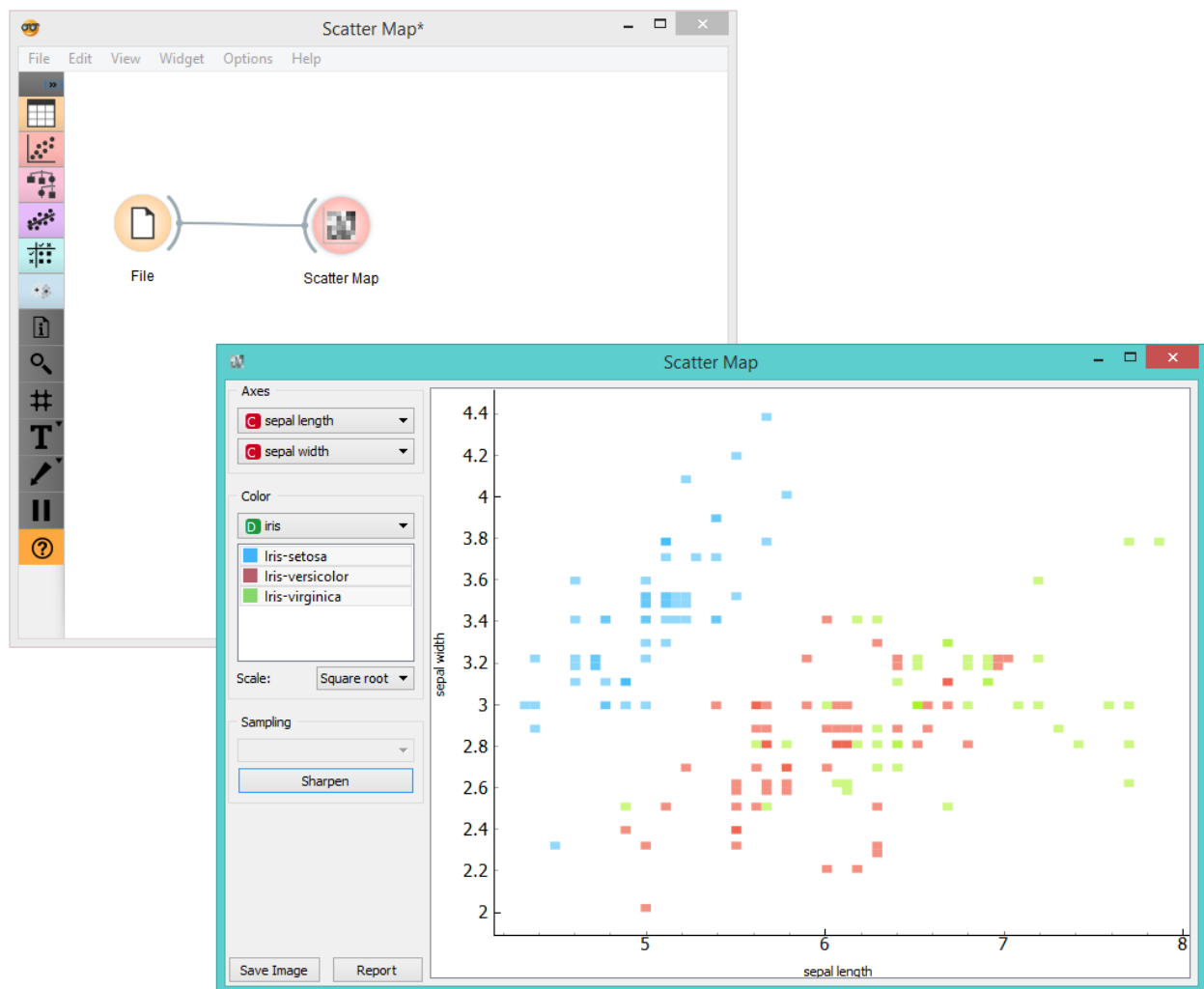
A [Scatter map](#) is a graphical method for visualizing frequencies in a two-way matrix by color. The higher the occurrence of a certain value, the darker the represented color. By combining two values on x and y axes, we see where the attribute combination is the strongest and where the weakest, thus enabling the user to find strong correlations or representative instances.



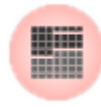
1. Select the x and y attribute to be plotted.
2. Color the plot by attribute. You can also select which attribute instances you wish to see in the visualization by clicking on them. At the bottom, you can select the color scale strength ([linear](#), [square root](#) or [logarithmic](#)).
3. *Sampling* is enabled only when the widget is connected to the *SQL Table* widget. You can set the sampling time for large data to speed up the analysis. *Sharpen* works for all data types and it will resize (sharpen) the squares in the plot.
4. *Save Image* saves the created image to your computer in a .svg or .png format.
5. Produce a report.

## Example

Below, you can see an example workflow for the **Scatter Map** widget. Notice that the widget only works with continuous data, so you need to first continuize the data attributes you want to visualize. The Scatter map below displays two attributes from the *Iris* data set, namely the petal width and petal length. Here, we can see the distribution of width and length values per *Iris* type. You can see that the variety *Iris setosa* is distinctly separated from the other two varieties by petal width and length and that the most typical values for these attributes are around 0.2 for petal width and between 1.4 and 1.7 for petal length. This shows that petal width and length are good attributes for telling *Iris setosa* apart from the other two varieties.



## Sieve Diagram



Plots a sieve diagram for a pair of attributes.

### Signals

#### Inputs:

- **Data**  
An input data set

#### Outputs:

- None

### Description

A **Sieve diagram** is a graphical method for visualizing frequencies in a two-way contingency table and comparing them to **expected frequencies** under assumption of independence. It was proposed by Riedwyl and Schüpbach in a technical report in 1983 and later called a **parquet diagram** (Riedwyl and Schüpbach, 1994). In this display, the area of each rectangle is proportional to the expected frequency, while the observed frequency is shown by the number of squares in each rectangle. The difference between observed and expected frequency (proportional to the standard Pearson residual) appears as the density of shading, using color to indicate whether the deviation from independence is positive (blue) or negative (red).

1. Select the attributes you want to display in the sieve plot.
2. Score combinations enables you to find the best possible combination of attributes.
3. *Save Image* saves the created image to your computer in a .svg or .png format.
4. Produce a report.

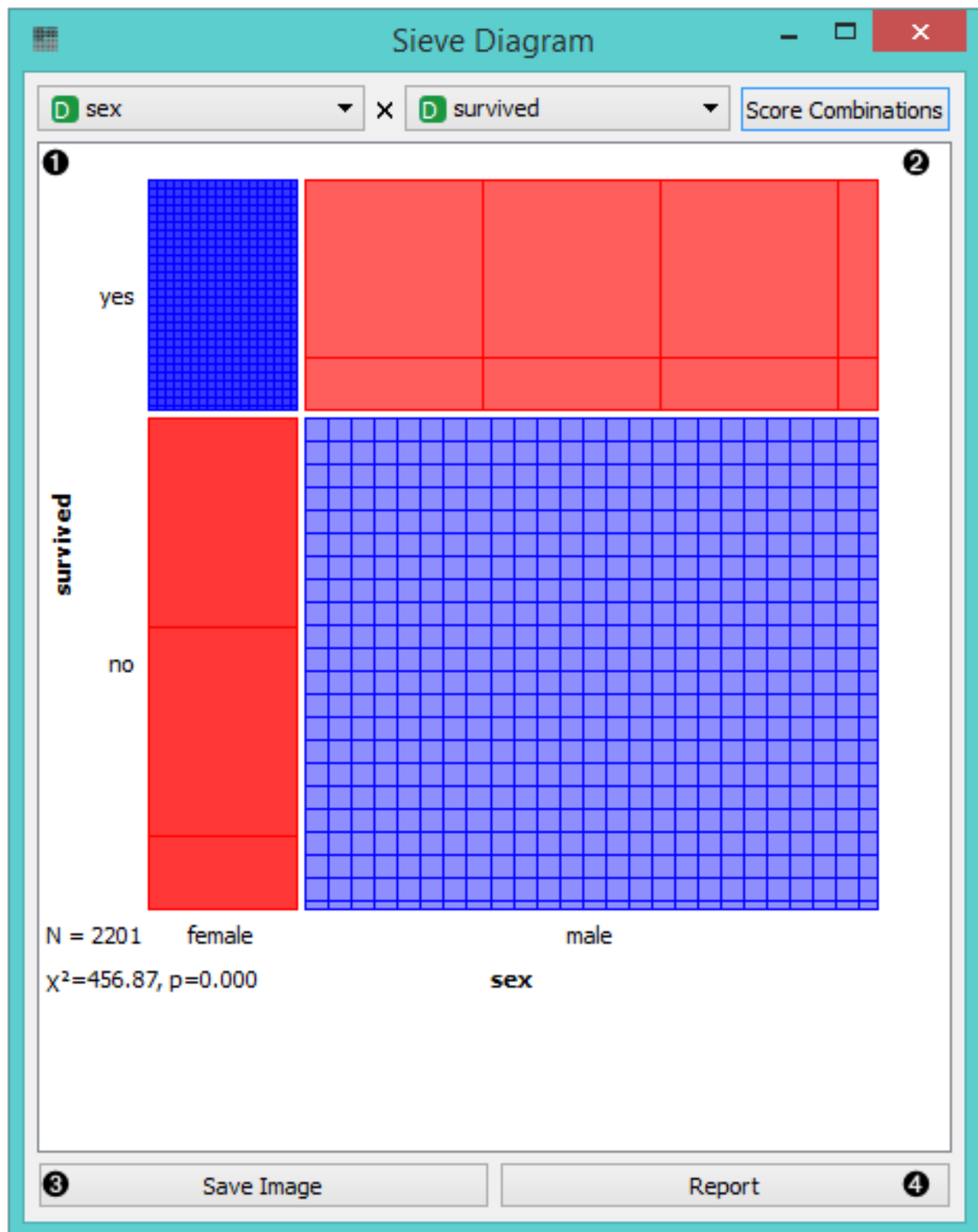
The snapshot below shows a sieve diagram for the *Titanic* data set and has the attributes *sex* and *survived* (the latter is a class attribute in this data set). The plot shows that the two variables are highly associated, as there are substantial differences between observed and expected frequencies in all of the four quadrants. For example, and as highlighted in the balloon, the chance for surviving the accident was much higher for female passengers than expected (0.06 vs. 0.15).

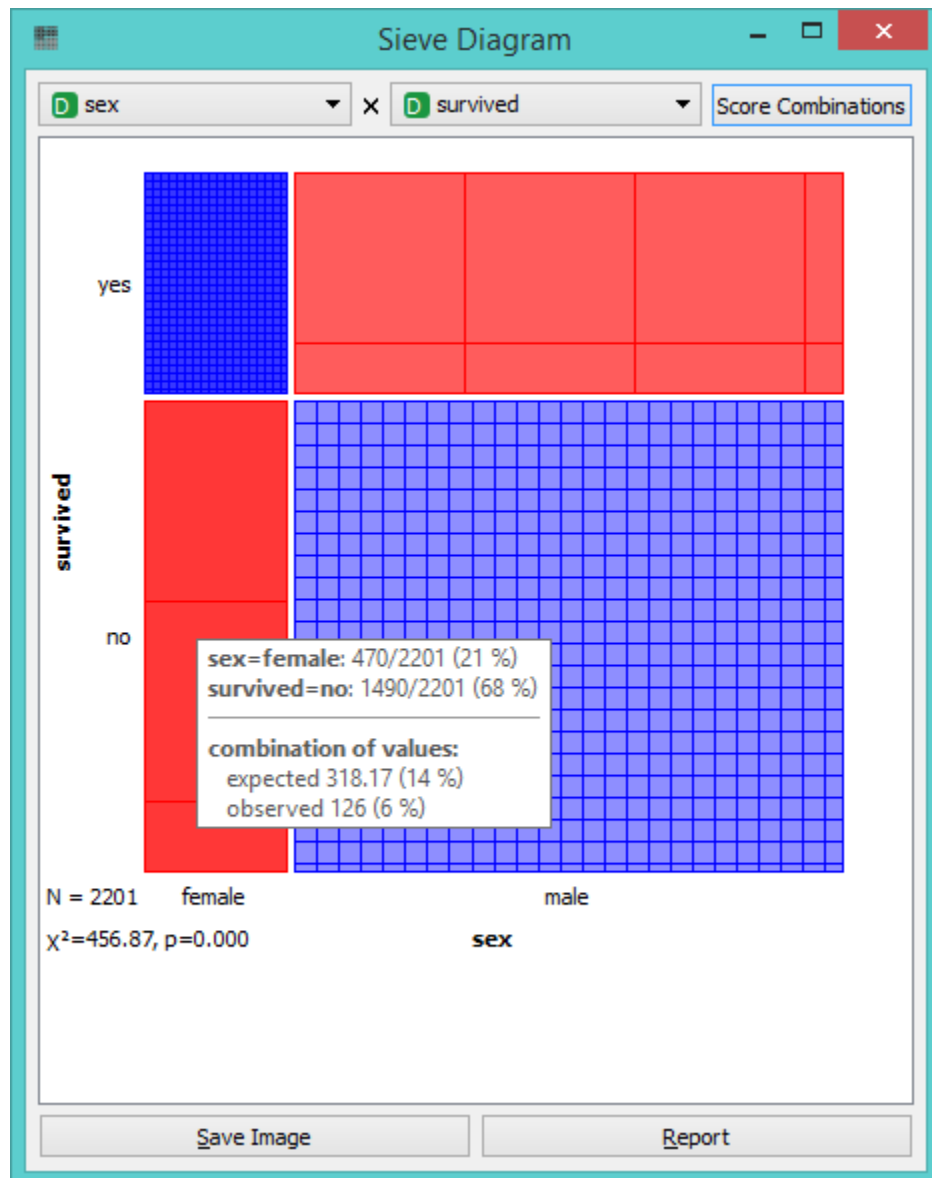
Pairs of attributes with interesting associations have a strong shading, such as the diagram shown in the above snapshot. For contrast, a sieve diagram of the least interesting pair (age vs. survival) is shown below.

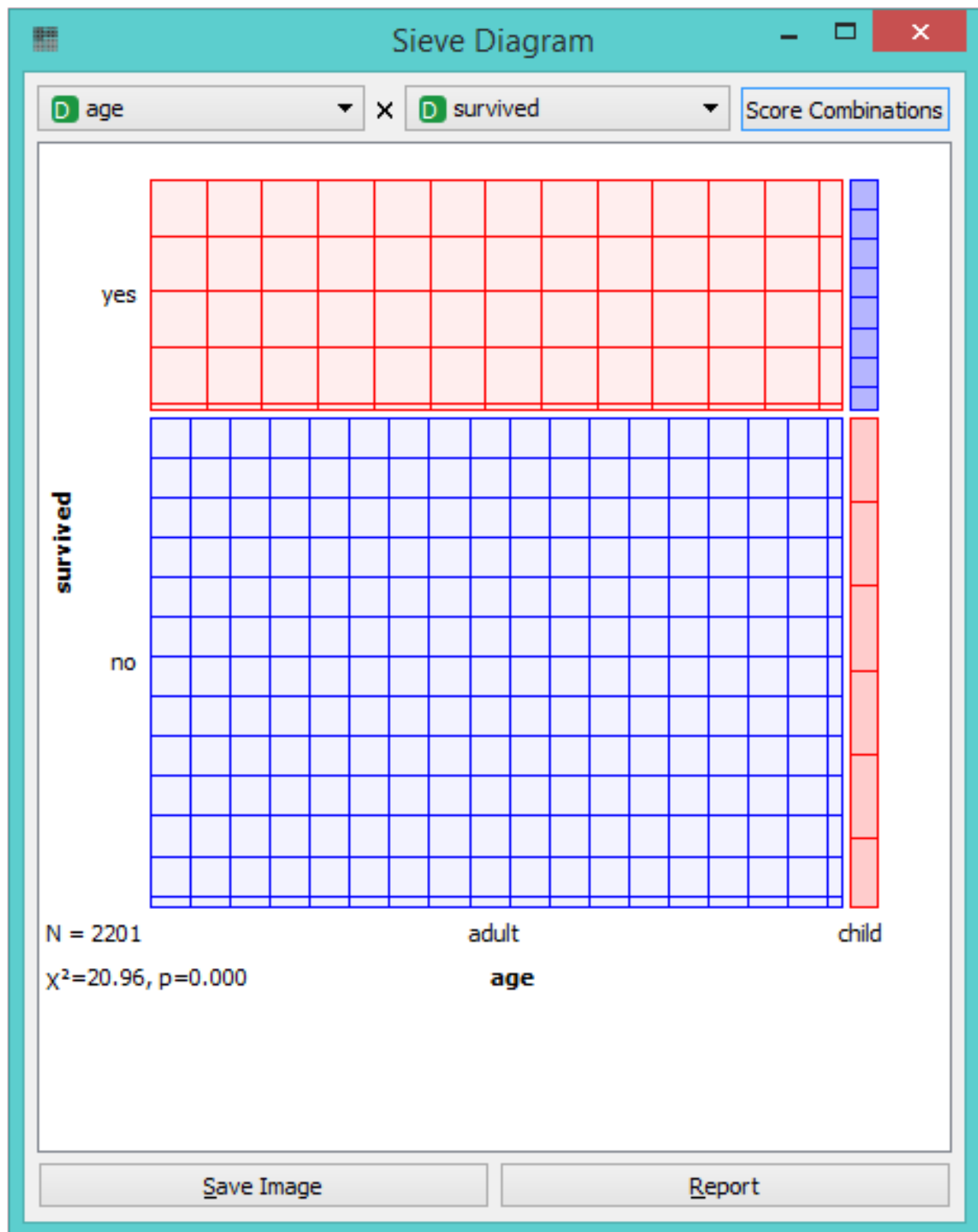
### Example

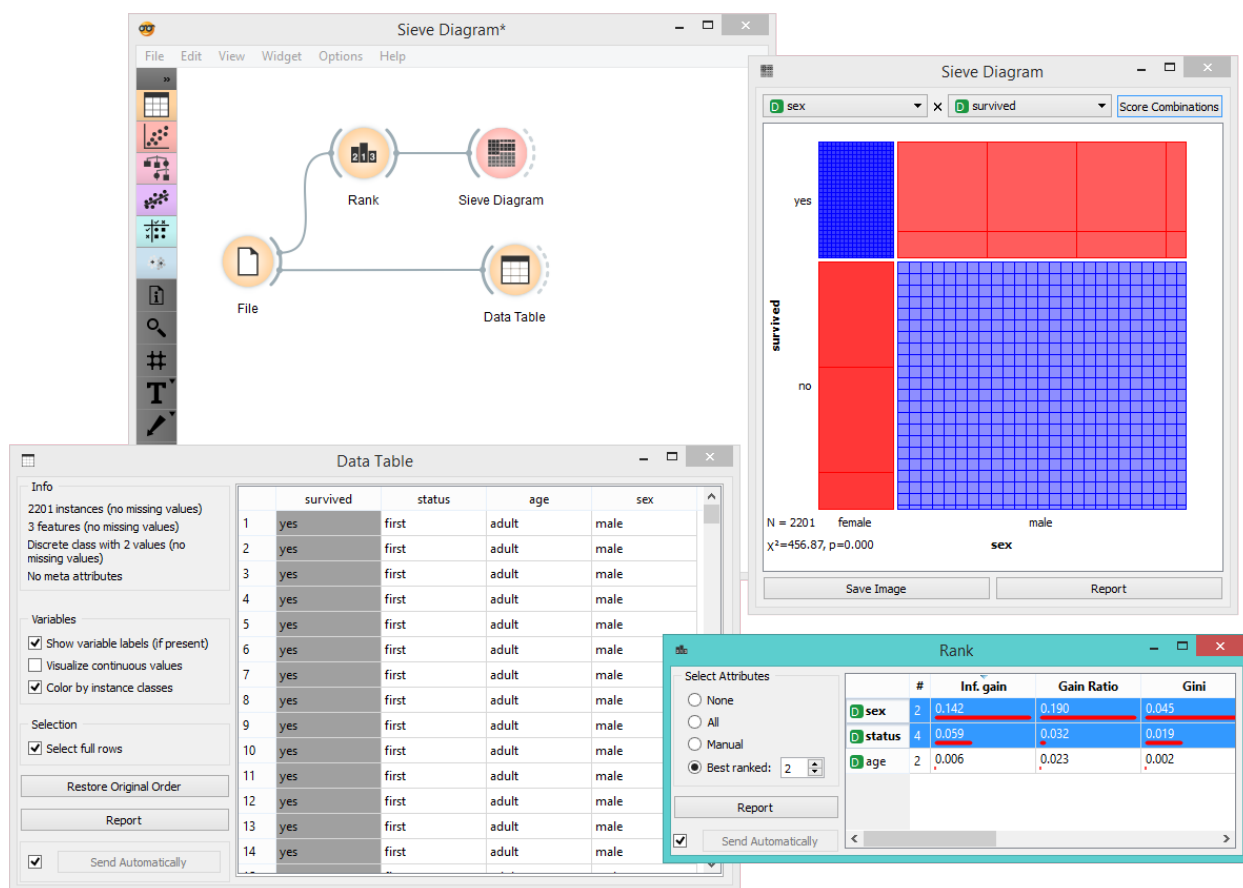
Below, we see a simple schema using the *Titanic* data set, where we use the *Rank* widget to select the best attributes (the ones with the highest information gain, gain ratio or gini index) and feed them into the **Sieve Diagram**. This displays the sieve plot for the two best attributes, which in our case are *sex* and *status*. We see that the survival rate on the *Titanic* was very high for women of the first class and very low for female crew members.

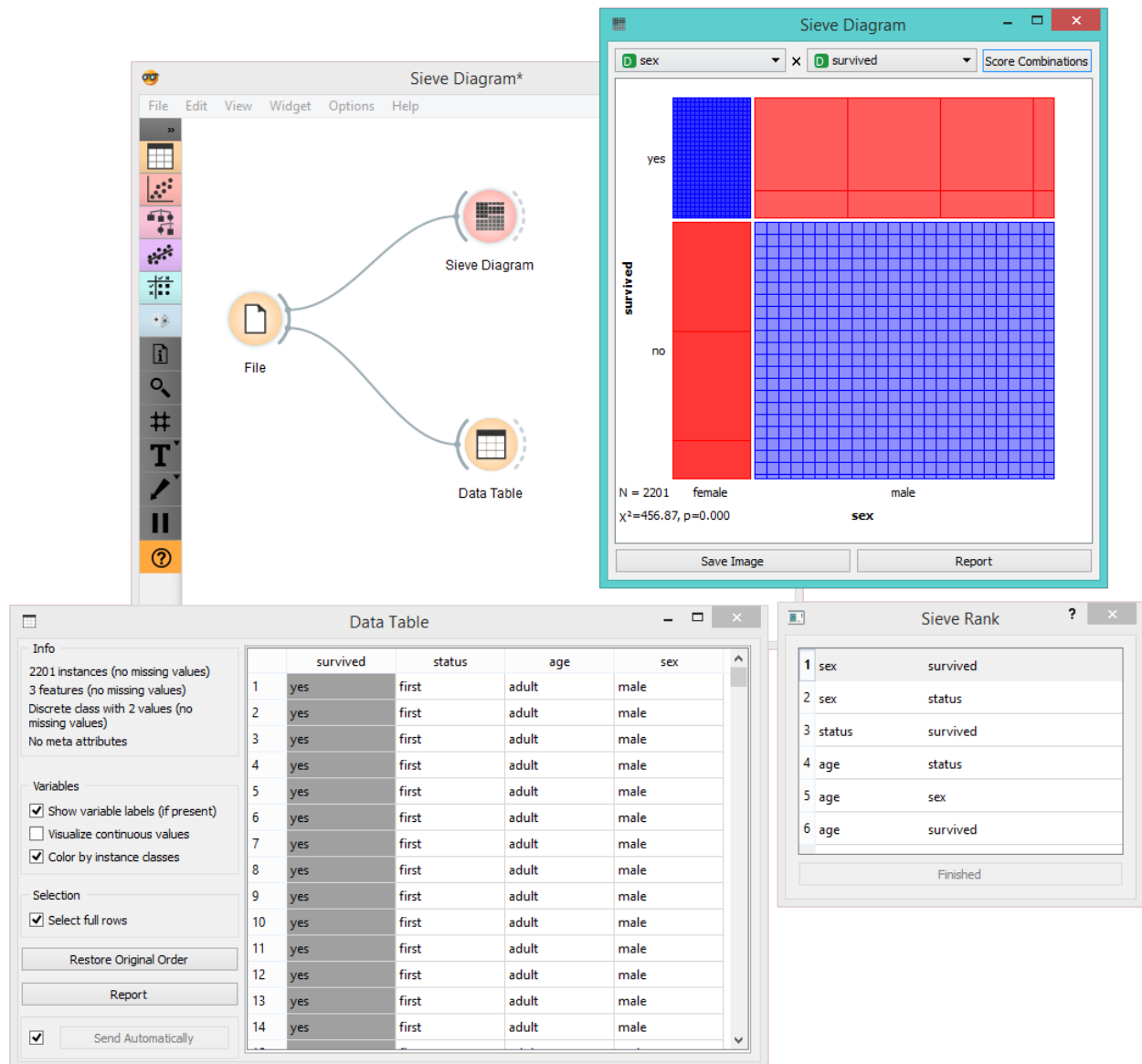
The **Sieve Diagram** also features the *Score Combinations* option, which makes the ranking of attributes even easier.













## References

Riedwyl, H., and Schüpbach, M. (1994). Parquet diagram to plot contingency tables. In Softstat '93: Advances in Statistical Software, F. Faulbaum (Ed.). New York: Gustav Fischer, 293-299.

## Pythagorean Tree



Pythagorean tree visualisation for classification or regression trees.

## Signals

### Inputs:

- **Tree**

A classification or regression tree model.

- **Selected Data**

A subset of instances that the user has manually selected from the Pythagorean tree.

## Description

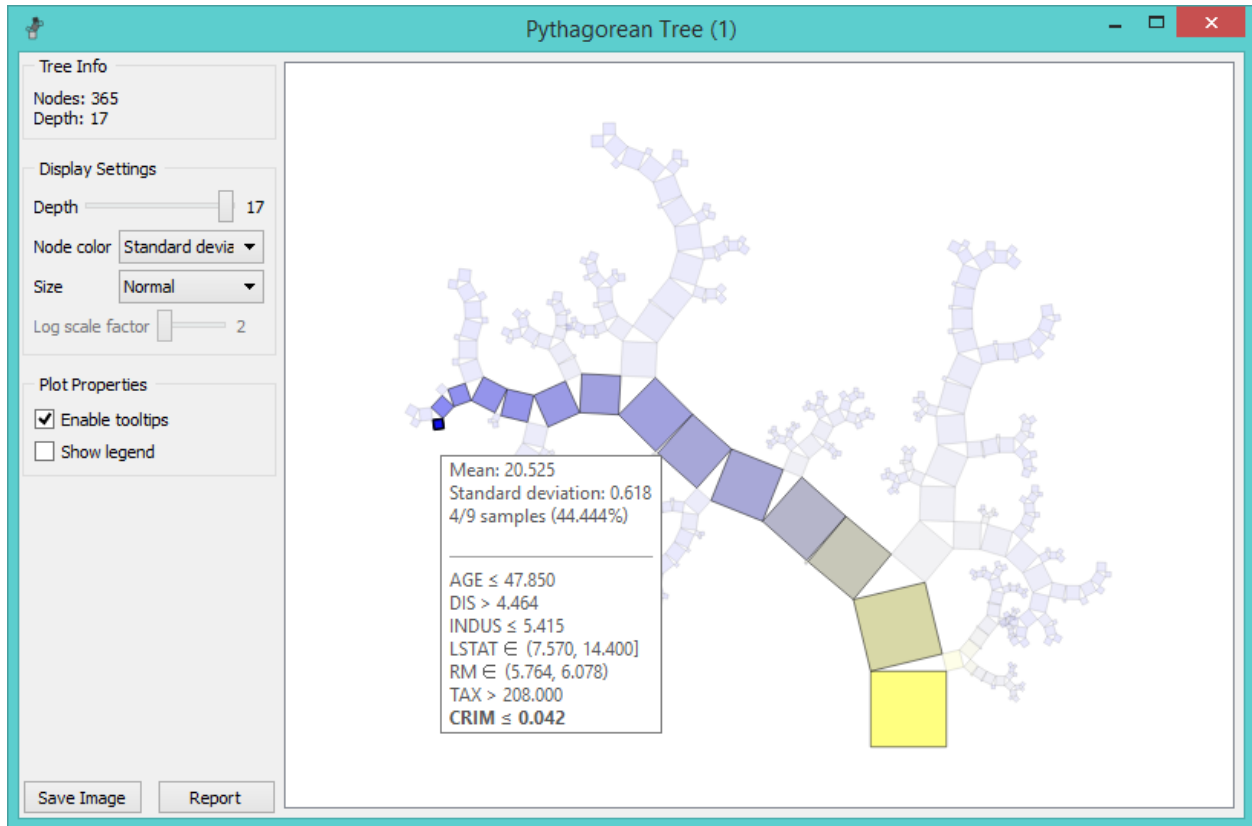
**Pythagorean Trees** are plane fractals that can be used to depict general tree hierarchies as presented in an article by [Fabian Beck and co-authors](#). In our case, they are used for visualizing and exploring tree models, such as *Classification Tree*.

1. Information on the input tree model.
2. Visualization parameters:
  - *Depth*: set the depth of displayed trees.
  - *Target class* (for classification trees): the intensity of the color for nodes of the tree will correspond to the probability of the target class. If *None* is selected, the color of the node will denote the most probable class.
  - *Node color* (for regression trees): node colors can correspond to mean or standard deviation of class value of the training data instances in the node.
  - *Size*: define a method to compute the size of the square representing the node. *Normal* will keep node sizes correspond to the size of training data subset in the node. *Square root* and *Logarithmic* are the respective transformations of the node size.
  - *Log scale factor* is only enabled when *logarithmic* transformation is selected. You can set the log factor between 1 and 10.
3. Plot properties:
  - *Enable tooltips*: display node information upon hovering.
  - *Show legend*: shows color legend for the plot.
4. **Reporting:**



- *Save Image*: save the visualization to a SVG or PNG file.
- *Report*: add visualization to the report.

Pythagorean Tree can visualize both classification and regression trees. Below is an example for regression tree. The only difference between the two is that regression tree doesn't enable coloring by class, but can color by class mean or standard deviation.



## Example

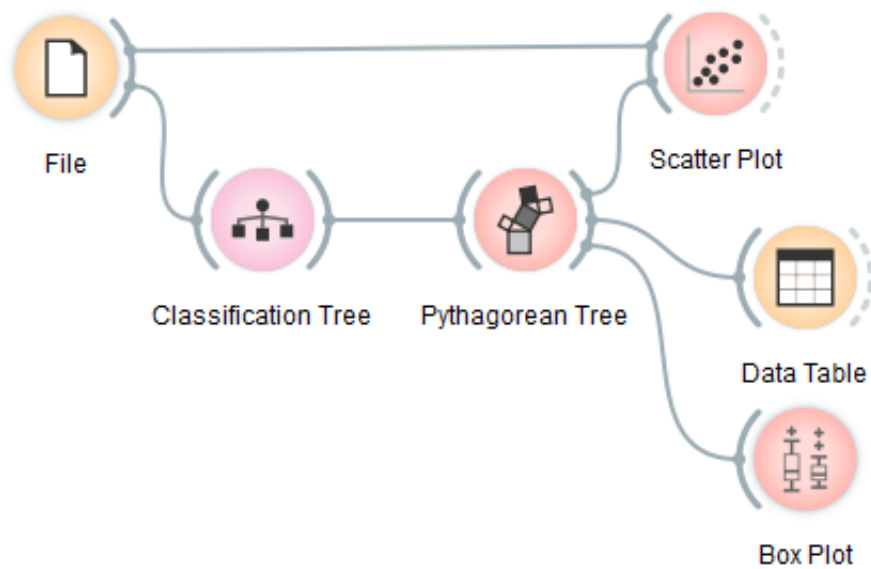
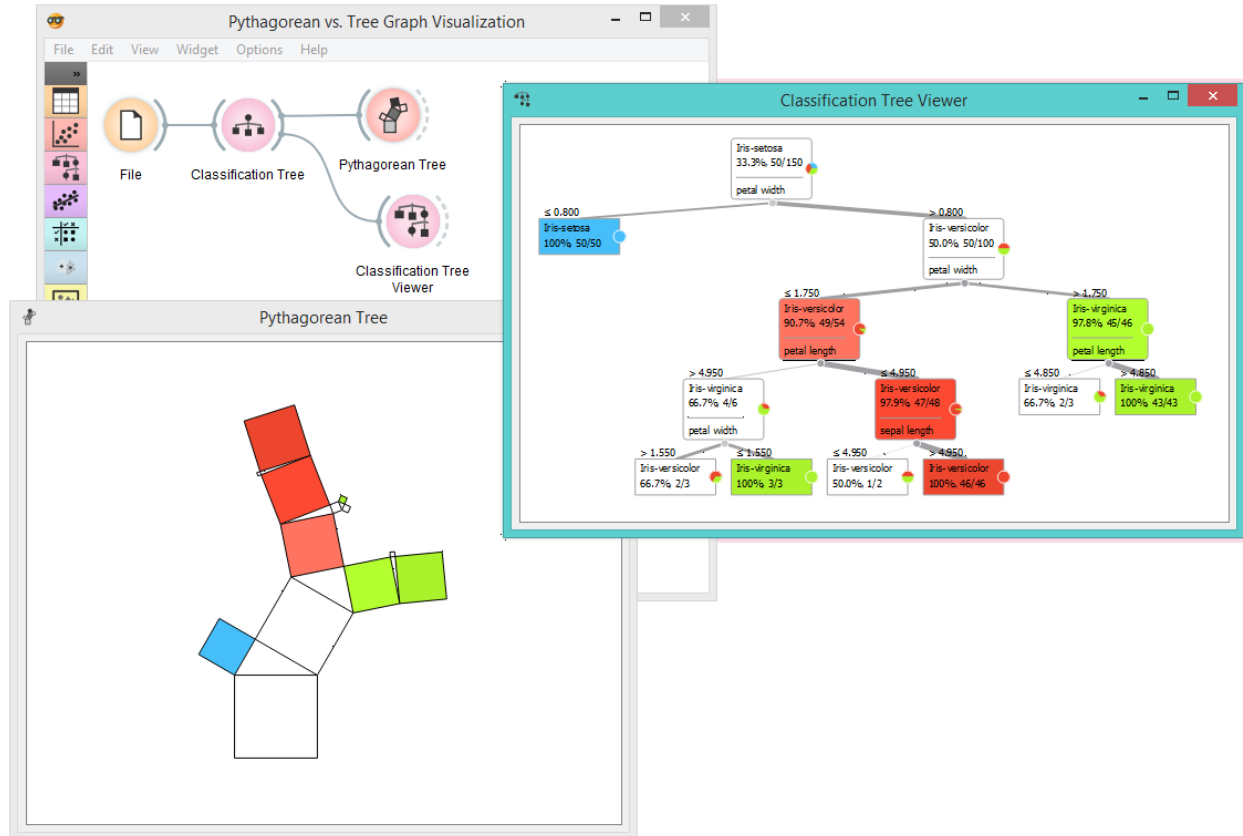
The workflow from the screenshot below demonstrates the difference between Classification Tree Viewer and Pythagorean Tree. They can both visualize *Classification Tree*, but Pythagorean visualization takes less space and is more compact, even for a small *Iris flower* data set. For both visualization widgets, we have hidden the control area on the left by clicking on the splitter between control and visualization area.

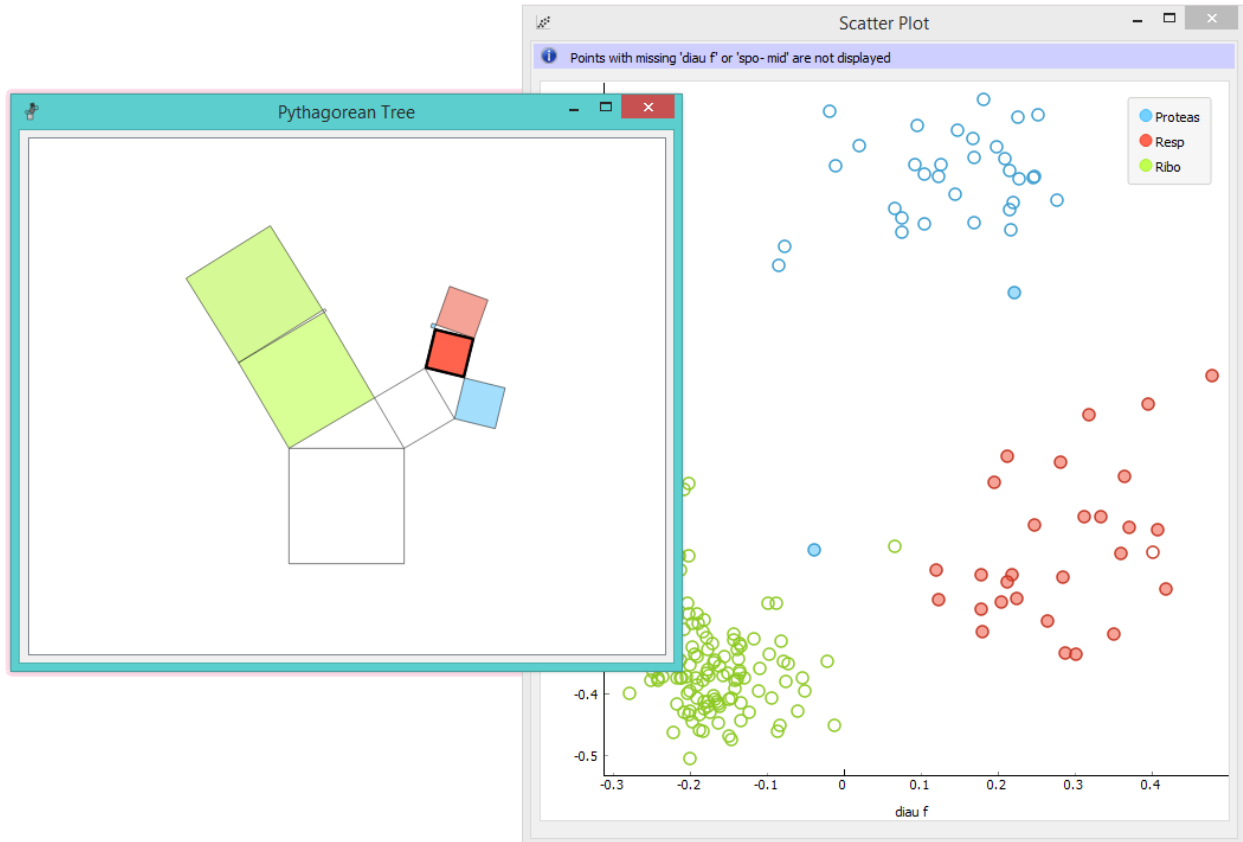
Pythagorean Tree is interactive: click on any of the nodes (squares) to select training data instances that were associated with that node. The following workflow explores these feature.

The selected data instances are shown as a subset in the *Scatter Plot*, sent to the *Data Table* and examined in the *Box Plot*. We have used brown-selected data set in this example. The tree and scatter plot are shown below; the selected node in the tree has a black outline.

## References

Beck, F., Burch, M., Munz, T., Di Silvestro, L. and Weiskopf, D. (2014). *Generalized Pythagoras Trees for Visualizing Hierarchies*. In IVAPP '14 Proceedings of the 5th International Conference on Information Visualization Theory and Applications, 17-28.





## Pythagorean Forest



Pythagorean forest for visualising random forests.

### Signals

#### Inputs:

- **Random Forest**  
Classification / regression tree models as random forest.

#### Outputs:

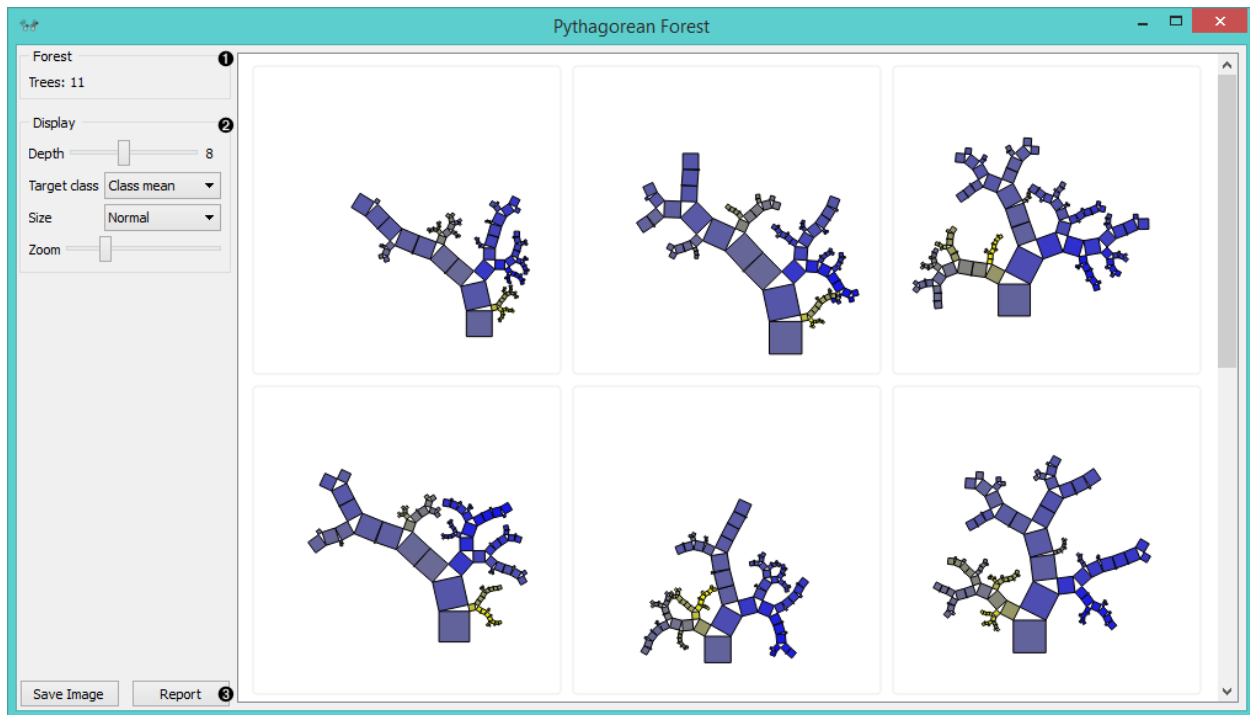
- **Tree**  
A selected classification / regression tree model.

### Description

**Pythagorean Forest** shows all learned classification tree models from *Random Forest* widget. It displays then as Pythagorean trees, each visualization pertaining to one randomly constructed tree. In the visualization, you can select

a tree and display it in **Pythagorean Tree** widget. The best tree is the one with the shortest and most strongly colored branches. This means few attributes split the branches well.

Widget displays both classification and regression results. Classification requires discrete target variable in the data set and a *Classification Tree* widget on the input, while regression requires a continuous target variable with *Regression Tree* on the input.

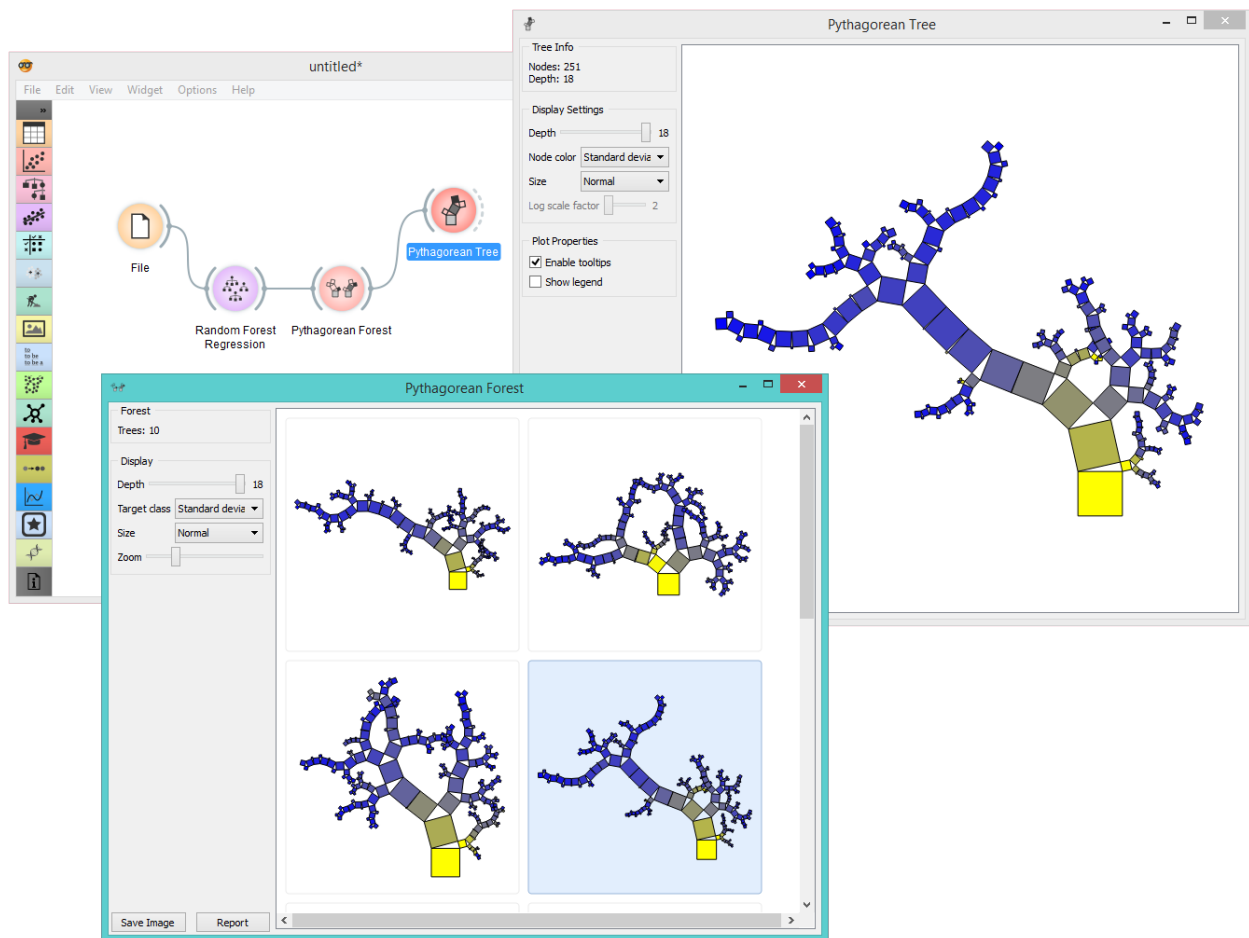


1. Information on the input random forest model.
2. Display parameters:
  - *Depth*: set the depth to which the trees are grown.
  - *Target class*: set the target class for coloring the trees. If *None* is selected, tree will be white. If the input is a classification tree, you can color nodes by their respective class. If the input is a regression tree, the options are *Class mean*, which will color tree nodes by the class mean value and *Standard deviation*, which will color then by the standard deviation value of the node.
  - *Size*: set the size of the nodes. *Normal* will keep nodes the size of the subset in the node. *Square root* and *Logarithmic* are the respective transformations of the node size.
  - *Zoom*: allows you to se the size of the tree visualizations.
3. *Save Image*: save the visualization to your computer as a .svg or .png file. *Report*: produce a report.

## Example

**Pythagorean Forest** is great for visualizing several built trees at once. In the example below, we've plotted all 10 trees we've grown with *Random Forest Regression*. When changing the parameters in Random Forest Regression, visualization in Pythagorean Forest will change as well.

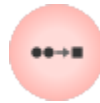
Then we've selected a tree in the visualization and inspected it further with *Pythagorean Tree* widget.



## References

Beck, F., Burch, M., Munz, T., Di Silvestro, L. and Weiskopf, D. (2014). Generalized Pythagoras Trees for Visualizing Hierarchies. In IVAPP '14 Proceedings of the 5th International Conference on Information Visualization Theory and Applications, 17-28.

## CN2 Rule Viewer



CN2 Rule Viewer

### Signals

#### Inputs:

- **Data**  
Data set to filter.
- **CN2 Rule Classifier**  
CN2 Rule Classifier, including a list of induced rules.

#### Outputs:

- **Filtered Data**  
If data is connected, upon active selection (at least one rule is selected), filtered data is emitted.  
Output are data instances covered by all selected rules.

### Description

A widget that displays **CN2 classification** rules. If data is also connected, upon rule selection, one can analyze which instances abide to the conditions.

1. Original order of induced rules can be restored.
2. When rules are many and complex, the view can appear packed. For this reason, *compact view* was implemented, which allows a flat presentation and a cleaner inspection of rules.
3. Click *Report* to bring up a detailed description of the rule induction algorithm and its parameters, the data domain, and induced rules.

Additionally, upon selection, rules can be copied to clipboard by pressing the default system shortcut (ctrl+C, cmd+C).

### Examples

In the schema below, the most common use of the widget is presented. First, the data is read and a CN2 rule classifier is trained. We are using *titanic* data set for the rule construction. The rules are then viewed using the [Rule Viewer](#). To explore different CN2 algorithms and understand how adjusting parameters influences the learning process, **Rule Viewer** should be kept open and in sight, while setting the CN2 learning algorithm (the presentation will be updated promptly).

Selecting a rule outputs filtered data instances. These can be viewed in a [Data Table](#).



Rule Viewer						
	IF conditions	THEN class	Distribution	Probabilities	Quality	Length
0	sex=female AND status=first AND age≠ad...	→ survived=yes	[0, 1]	0.33 : 0.67	-0.00	3
1	sex=female AND status≠third AND age≠a...	→ survived=yes	[0, 13]	0.07 : 0.93	-0.00	3
2	sex≠female AND status=second AND age...	→ survived=yes	[0, 11]	0.08 : 0.92	-0.00	3
3	sex≠female AND status=second	→ survived=no	[154, 14]	0.91 : 0.09	-0.414	2
4	status=crew AND sex=female	→ survived=yes	[3, 20]	0.16 : 0.84	-0.559	2
5	status=second	→ survived=yes	[13, 80]	0.15 : 0.85	-0.584	1
6	sex≠female AND status=third AND age=a...	→ survived=no	[387, 75]	0.84 : 0.16	-0.640	3
7	sex=female AND status=first	→ survived=yes	[4, 140]	0.03 : 0.97	-0.183	2
8	status≠third AND age≠adult	→ survived=yes	[0, 5]	0.14 : 0.86	-0.00	2
9	status=crew	→ survived=no	[670, 192]	0.78 : 0.22	-0.765	1
10	sex≠female AND status≠first	→ survived=no	[35, 13]	0.72 : 0.28	-0.843	2
11	status=first	→ survived=no	[118, 57]	0.67 : 0.33	-0.910	1
12	age≠adult	→ survived=no	[17, 14]	0.55 : 0.46	-0.993	1
13	TRUE	→ survived=no	[89, 76]	0.54 : 0.46	-0.996	0

Restore original order ❶ ☒ Compact view ❷ Report ❸

untitled\*

File Edit View Widget Options Help

File CN2 Rule Viewer

CN2 Rule Induction

CN2 Rule Induction ? X

Name: CN2 rule inducer

Rule ordering: ☒ Ordered ☐ Unordered

Covering algorithm: ☒ Exclusive ☐ Weighted V: 0.70

Rule search

Evaluation measure: Entropy

Beam width: 5

Rule filtering

Minimum rule coverage: 1

Maximum rule length: 5

☐ Statistical significance (default α): 1.00

☐ Relative significance (parent α): 1.00

Report ☒ Apply Automatically

CN2 Rule Viewer

	IF conditions	THEN class	Distribution	Probabilities [%]	Quality	Length
0	sex=female AND status=first AND age≠adult	→ survived=yes	[0, 1]	33 : 67	-0.00	3
1	sex=female AND status≠third AND age≠adult	→ survived=yes	[0, 13]	7 : 93	-0.00	3
2	sex≠female AND status=second AND age≠adult	→ survived=yes	[0, 11]	8 : 92	-0.00	3
3	sex≠female AND status=second	→ survived=no	[154, 14]	91 : 9	-0.414	2
4	status=crew AND sex=female	→ survived=yes	[3, 20]	16 : 84	-0.559	2
5	status=second	→ survived=yes	[13, 80]	15 : 85	-0.584	1
6	sex≠female AND status=third AND age=adult	→ survived=no	[387, 75]	84 : 16	-0.640	3
7	sex=female AND status=first	→ survived=yes	[4, 140]	3 : 97	-0.183	2
8	status≠third AND age≠adult	→ survived=yes	[0, 5]	14 : 86	-0.00	2
9	status=crew	→ survived=no	[670, 192]	78 : 22	-0.765	1
10	sex≠female AND status≠first	→ survived=no	[35, 13]	72 : 28	-0.843	2
11	status=first	→ survived=no	[118, 57]	67 : 33	-0.910	1
12	age≠adult	→ survived=no	[17, 14]	55 : 45	-0.993	1
13	TRUE	→ survived=no	[89, 76]	54 : 46	-0.996	0

Restore original order ☒ Compact view Report

## Mosaic Display



Display data in a mosaic plot.

### Signals

#### Inputs:

- **Data**  
An input data set.
- **Data subset**  
An input data subset.

#### Outputs:

- **Selected data**  
A subset of instances that the user has manually selected from the plot.

### Description

The **Mosaic plot** is a graphical representation of a two-way frequency table or a contingency table. It is used for visualizing data from two or more qualitative variables and was introduced in 1981 by Hartigan and Kleiner and expanded and refined by Friendly in 1994. It provides the user with the means to more efficiently recognize relationships between different variables. If you wish to read up on the history of Mosaic Display, additional reading is available [here](#).

1. Select the variables you wish to see plotted.
2. Select interior coloring. You can color the interior according to class or you can use the *Pearson residual*, which is the difference between observed and fitted values, divided by an estimate of the standard deviation of the observed value. If *Compare to total* is clicked, a comparison is made to all instances.
3. *Save image* saves the created image to your computer in a .svg or .png format.
4. Produce a report.

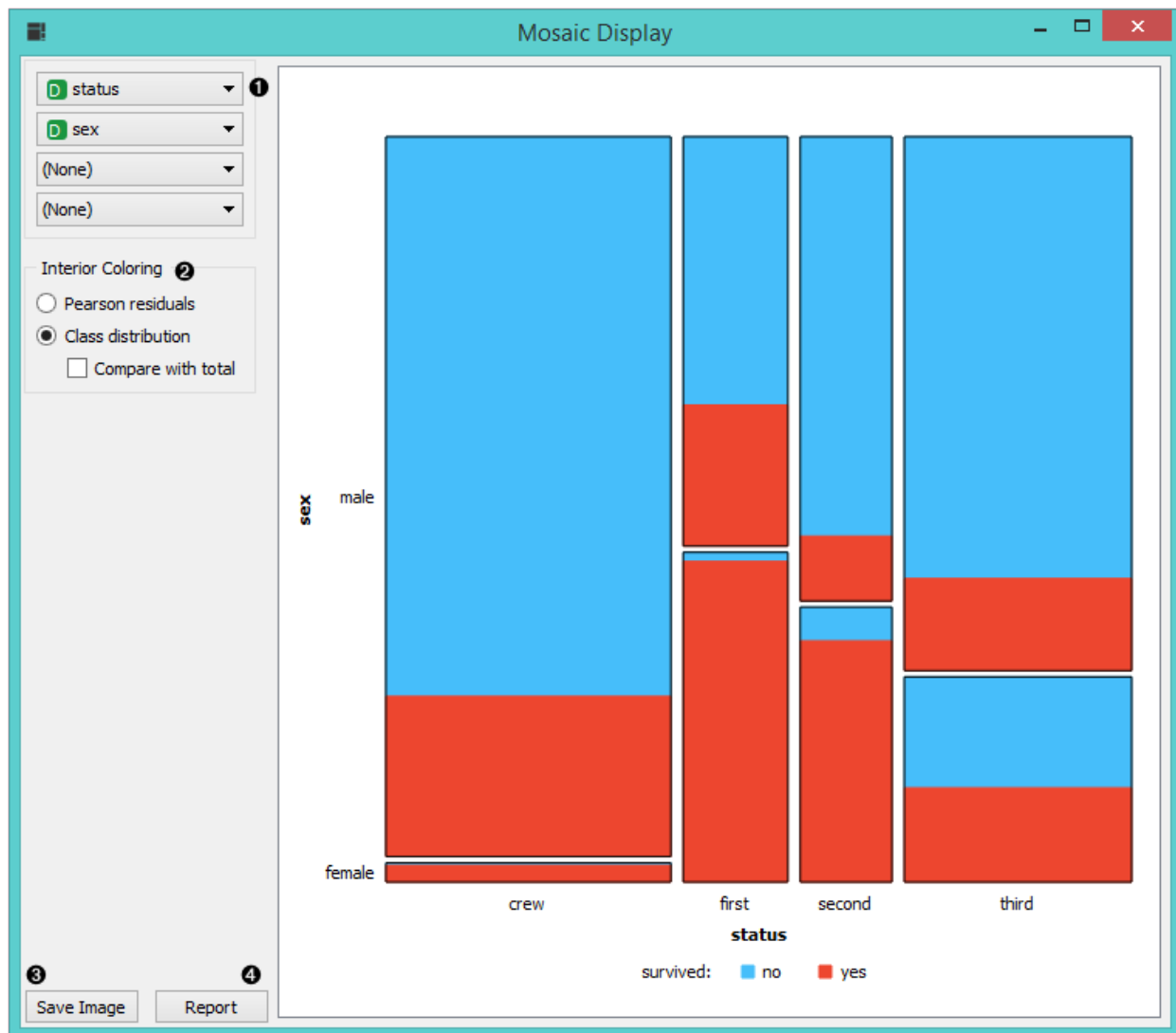
### Example

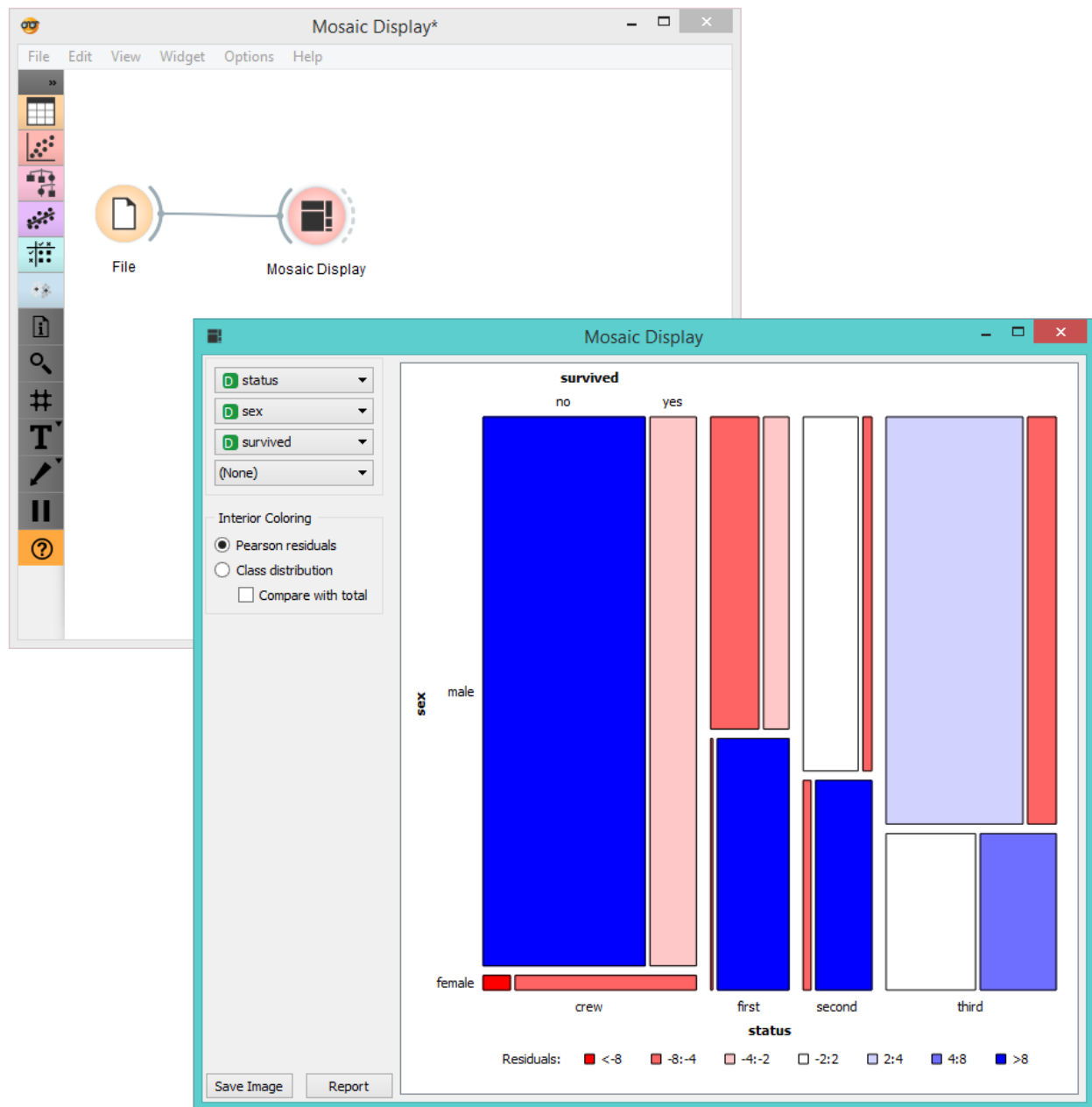
We loaded the *titanic* data set and connected it to the **Mosaic Display** widget. We decided to focus on two variables, namely status, sex and survival. We colored the interiors according to Pearson residuals in order to demonstrate the difference between observed and fitted values.

We can see that the survival rates for men and women clearly deviate from the fitted value.

### Silhouette Plot

A graphical representation of consistency within clusters of data.





## Signals

### Inputs

- **Data**

A data set.

### Outputs

- **Selected Data**

A subset of instances that the user has manually selected from the plot.

- **Other Data**

Remaining data.

## Description

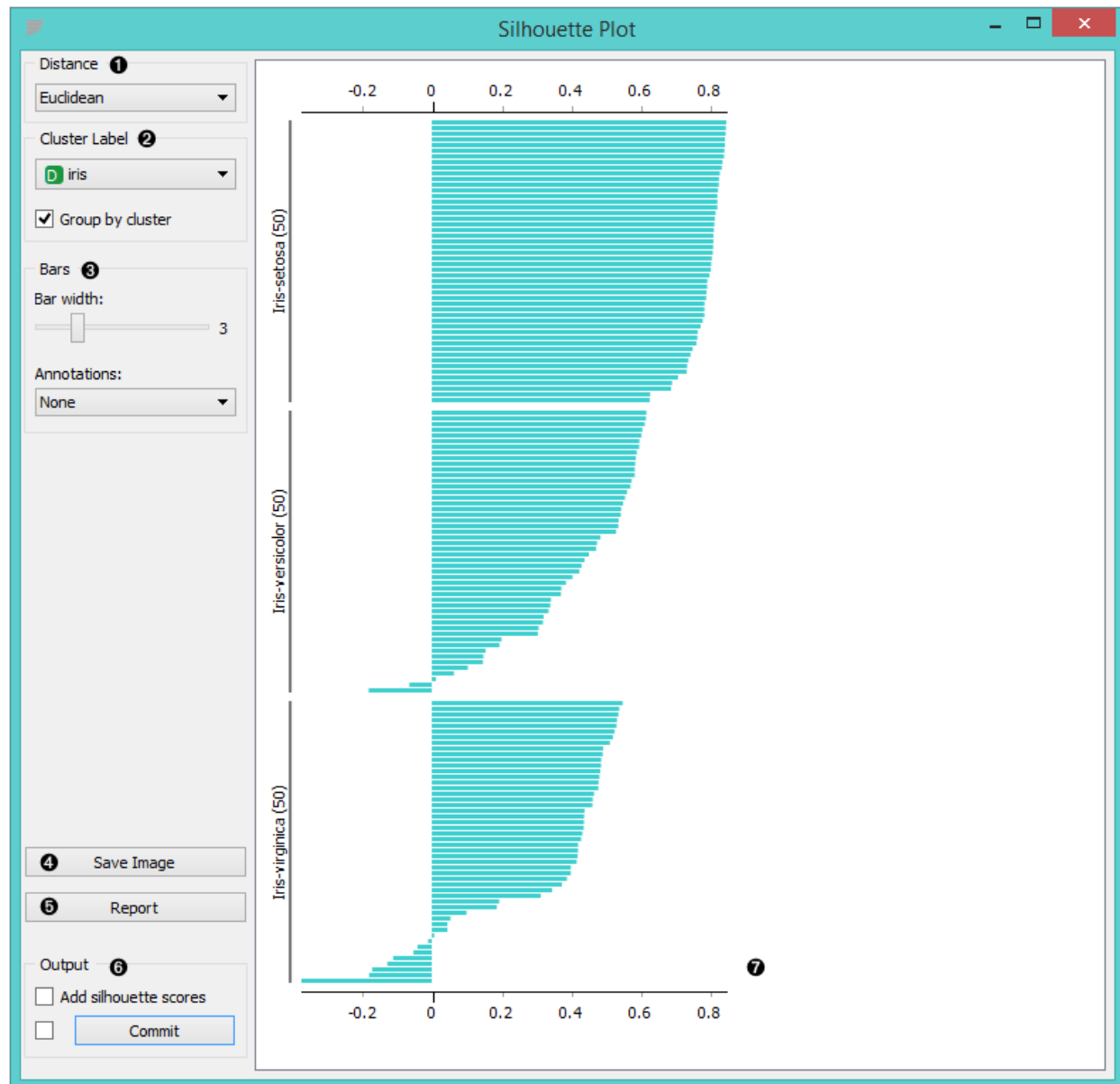
The **Silhouette Plot** widget offers a graphical representation of consistency within clusters of data and provides the user with the means to visually assess cluster quality. The silhouette score is a measure of how similar an object is to its own cluster in comparison to other clusters and is crucial in the creation of a silhouette plot. The silhouette score close to 1 indicates that the data instance is close to the center of the cluster and instances possessing the silhouette scores close to 0 are on the border between two clusters.

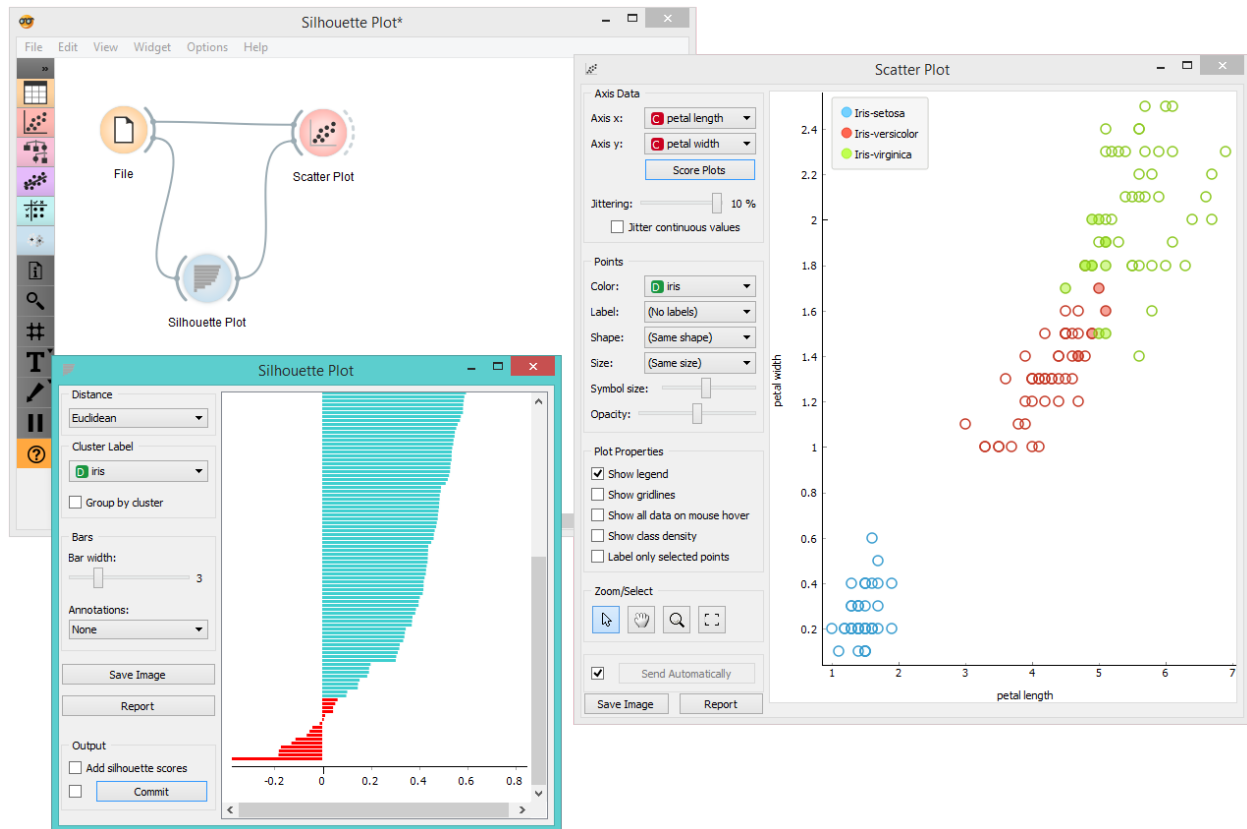
1. Choose the distance metric. You can choose between:
  - **Euclidean** (“straight line”, distance between two points)
  - **Manhattan** (the sum of absolute differences for all attributes)
2. Select the cluster label. You can decide whether to group the instances by cluster or not.
3. Display options:
  - *Choose bar width.*
  - *Annotations:* annotate the silhouette plot.
4. *Save Image* saves the created silhouette plot to your computer in a *.png* or *.svg* format.
5. Produce a report.
6. Output:
  - *Add silhouette scores* (good clusters have higher silhouette scores)
  - By clicking *Commit*, changes are communicated to the output of the widget. Alternatively, tick the box on the left and changes will be communicated automatically.
7. The created silhouette plot.

## Example

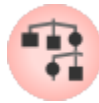
In the snapshot below, we have decided to use the **Silhouette Plot** on the *iris* data set. We selected data instances with low silhouette scores and passed them on as a subset to the *Scatter Plot* widget. This visualization only confirms the accuracy of the **Silhouette Plot** widget, as you can clearly see that the subset lies in the border between two clusters.

If you are interested in other uses of the **Silhouette Plot** widget, feel free to explore our [blog post](#).





## Tree Viewer



A visualization of classification and regression trees.

## Signals

### Inputs:

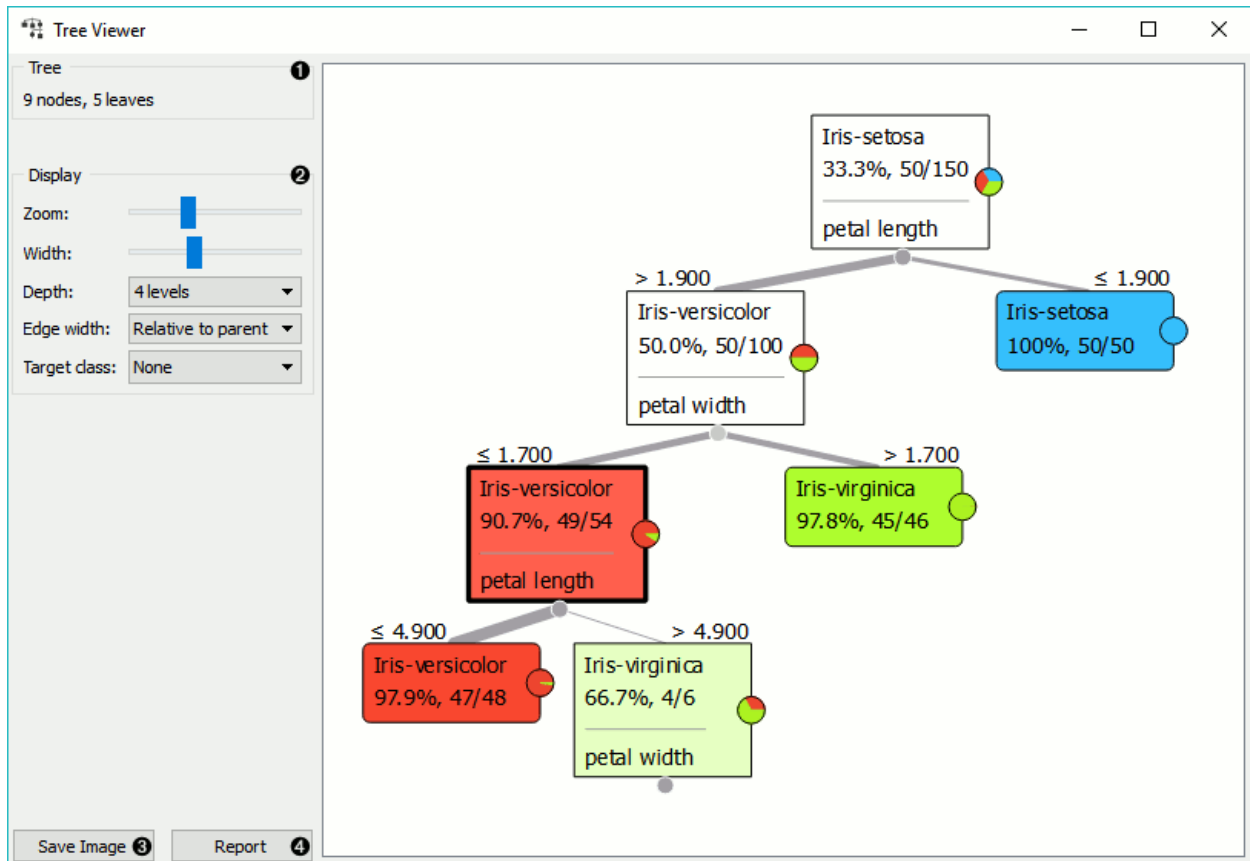
- **Tree**  
Classification or regression tree.

### Outputs:

- **Selected Data**  
Data from a selected tree node.
- **Data**  
Data set with an additional attribute for selection labels.

## Description

This is a versatile widget with 2-D visualization of a [classification tree](#). The user can select a node, instructing the widget to output the data associated with the node, thus enabling explorative data analysis.



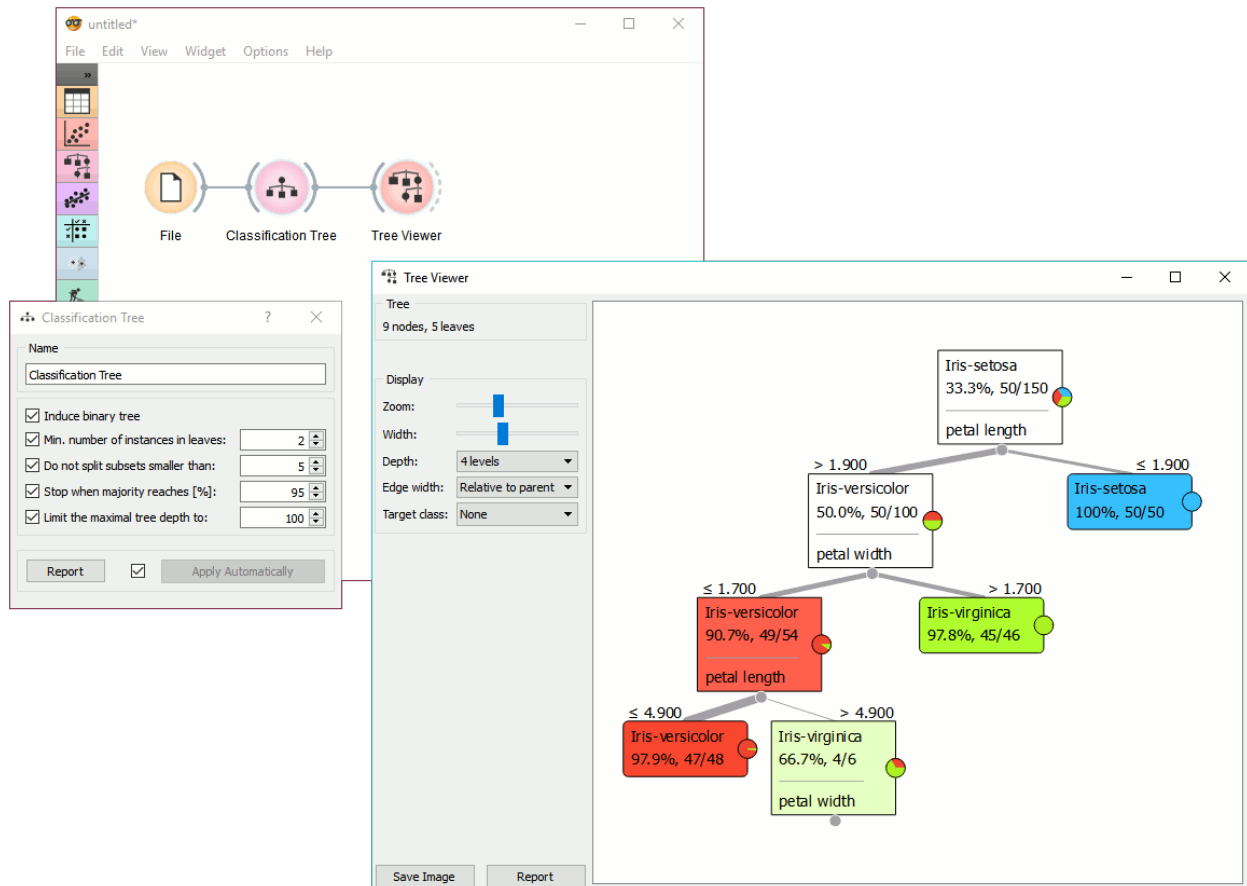
1. Information on the input.
2. Display options:
  - Zoom in and zoom out
  - Select the tree width. The nodes display information bubbles when hovering over them.
  - Select the depth of your tree.
  - **Select edge width. The edges between the nodes in the tree graph are drawn based on the selected edge width.**
    - All the edges will be of equal width if *Fixed* is chosen.
    - When *Relative to root* is selected, the width of the edge will correspond to the proportion of instances in the corresponding node with respect to all the instances in the training data. Under this selection, the edge will get thinner and thinner when traversing toward the bottom of the tree.
    - *Relative to parent* makes the edge width correspond to the proportion of instances in the nodes with respect to the instances in their parent node.
  - Define the target class, which you can change based on classes in the data.
3. Press *Save image* to save the created classification tree graph to your computer as a .svg or .png file.



4. Produce a report.

## Examples

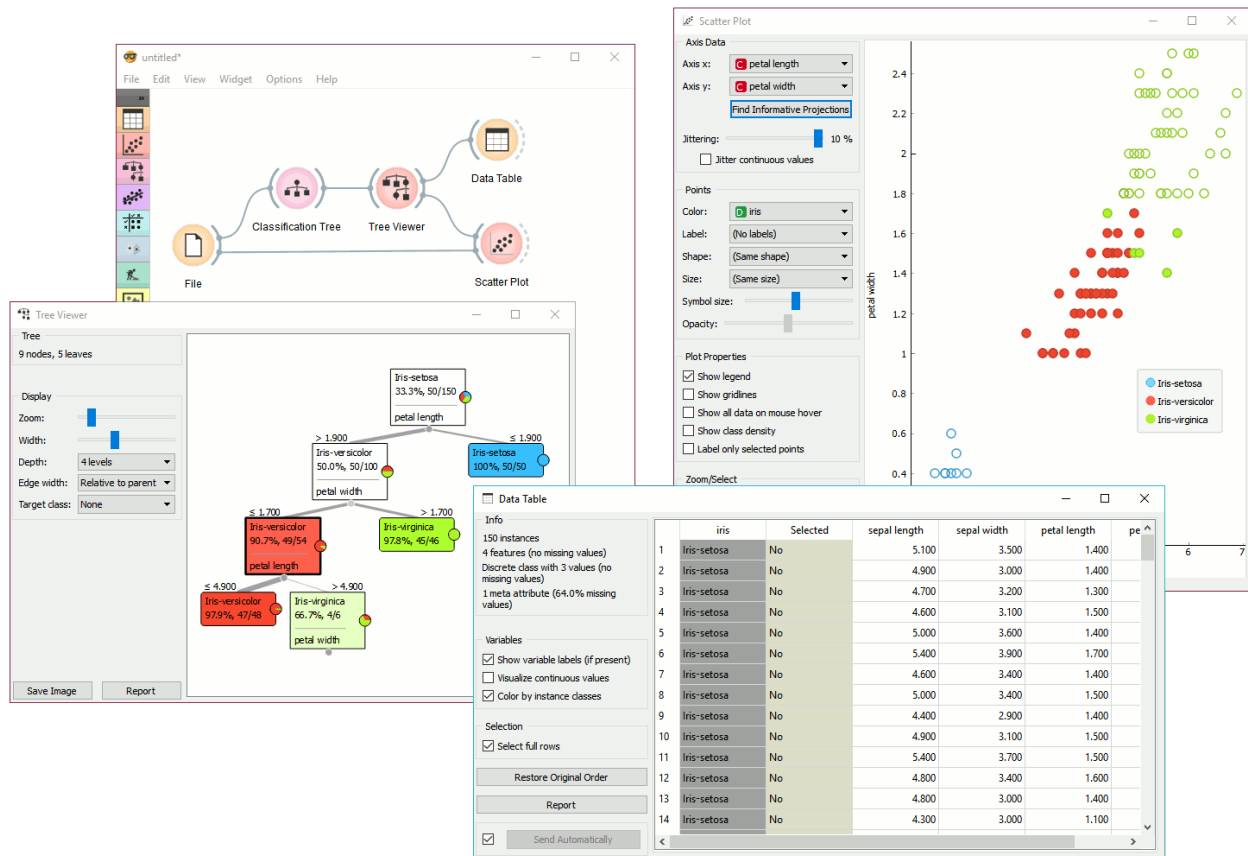
Below, is a simple schema, where we have read the data, constructed the classification tree and viewed it in our tree viewer. If both the viewer and *Classification Tree* are open, any re-run of the tree induction algorithm will immediately affect the visualization. You can thus use this combination to explore how the parameters of the induction algorithm influence the structure of the resulting tree.



Clicking on any node will output the related data instances. This is explored in the schema below that shows the subset in the data table and in the *Scatterplot*. Make sure that the tree data is passed as a data subset; this can be done by connecting the **Scatterplot** to the *File* widget first, and connecting it to the **Classification Tree Viewer** widget next. Selected data will be displayed as bold dots.

**Tree Viewer** can also export labelled data. Connect *Data Table* to **Tree Viewer** and set the link between widgets to *Data* instead of *Selected Data*. This will send the entire data to **Data Table** with an additional meta column labelling selected data instances (*Yes* for selected and *No* for the remaining).

Finally, **Tree Viewer** can be used also for visualizing regression trees. Connect *Regression Tree* to *File* widget using *housing.tab* data set. Then connect **Tree Viewer** to **Regression Tree**. The widget will display the constructed tree. For visualizing larger trees, especially for regression, *Pythagorean Tree* could be a better option.



## Geo Map

Show data points on a world map.

### Signals

#### Inputs:

- **Data**  
An input data set.
- **Data Subset**  
A subset of instances from the input data set.
- **Learner**  
A learning algorithm (classification or regression).

#### Outputs:

- **Selected Data**  
A subset of instances that the user has manually selected from the map.
- **Data**  
Data set with an appended meta attribute specifying selected and unselected data.

### Description

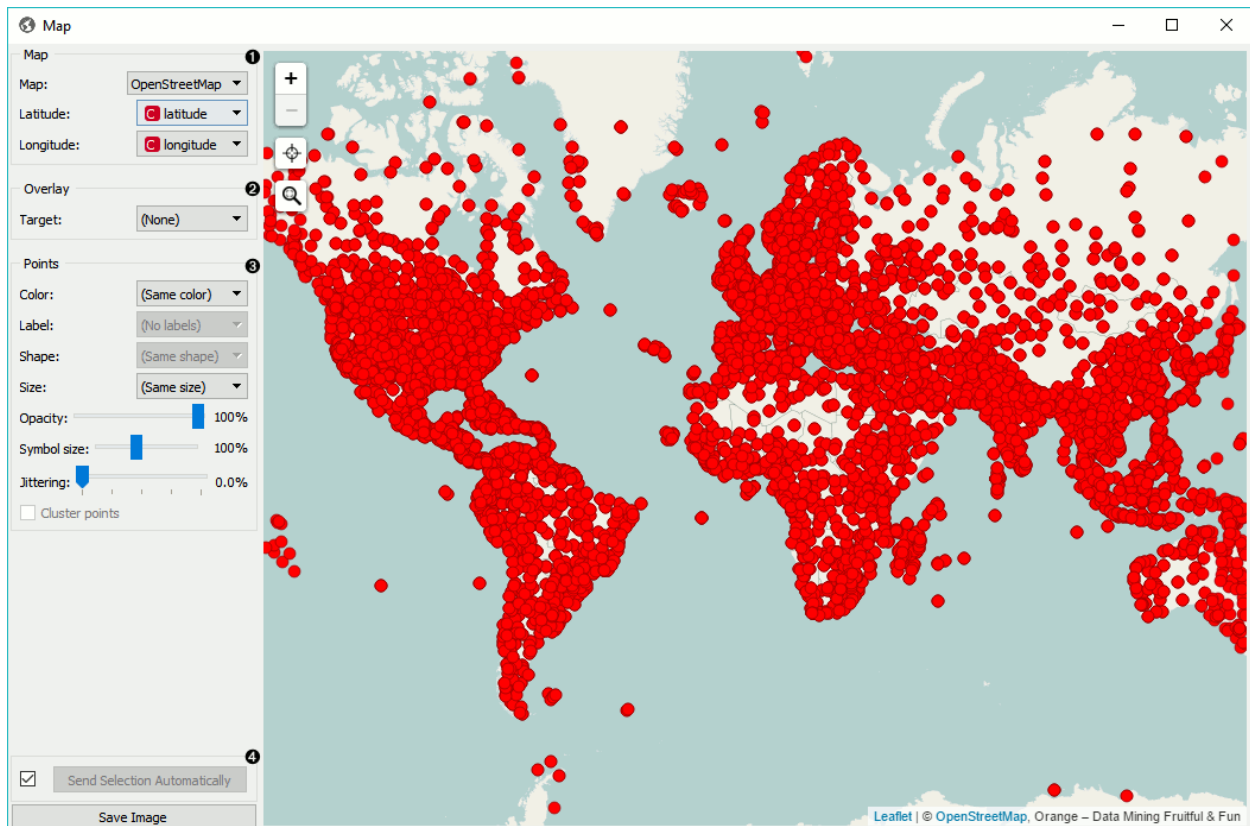
**Geo Map** widget maps geo-spatial data on a world map. It only works on data sets containing latitude and longitude variables. It also enables class predictions when a learner is provided on the input.

1. Define map properties: - Set the type of *map*: Black and White, [OpenStreetMap](#), Topographic, Satellite, Print, Light, Dark, Railyways and Watercolor. - Set latitude and longitude attributes, if the widget didn't recognize them automatically. Latitude values should be between -90(S) and 90(N) and longitude values between -180(W) and 180(E).
2. Overlay: - Set the target (class) for predictive mapping. A learner has to be provided on the input. The classifier is trained on latitude and longitude pairs only (i.e. it maps lat/lon pairs to the selected attribute).
3. Set point parameters: - *Color*: color of data points by attribute values - *Label*: label data points with an attribute (available when zoomed in) - *Shape*: shape of data points by attribute (available when zoomed in) - *Size*: size of data points by attribute - *Opacity*: set transparency of data points - *Symbol size*: size of data points (small to large) - *Jittering*: disperse overlaid data points - *Cluster points*: cluster neighboring points with [naive greedy clustering](#) (available when less than 600 points are in view)
4. If *Send Selection Automatically* is ticked, changes are communicated automatically. Alternatively, click *Send Selection*. *Save image* saves the image to your computer in a .svg or .png format.

---

**Note:** To select a subset of points from the map, hold Shift and draw a rectangle around the point you want to output.

---



## Examples

In the first example we will model class predictions on a map. We will use *philadelphia-crime* data set, load it with *File* widget and connect it to **Map**. We can already observe the mapped points in Map. Now, we connect *Classification Tree* to Map and set target variable to Type. This will display the predicted type of crime for a specific region of Philadelphia city (each region will be colored with a corresponding color code, explained in a legend on the right).

The second example uses *global-airports.csv* data. Say we somehow want to predict the altitude of the area based solely on the latitude and longitude. We again load the data with *File* widget and connect it to Map. Then we use a regressor, say, *KNN* and connect it to Map as well. Now we set target to altitude and use Black and White map type. The model guessed the Himalaya, but made some errors elsewhere.

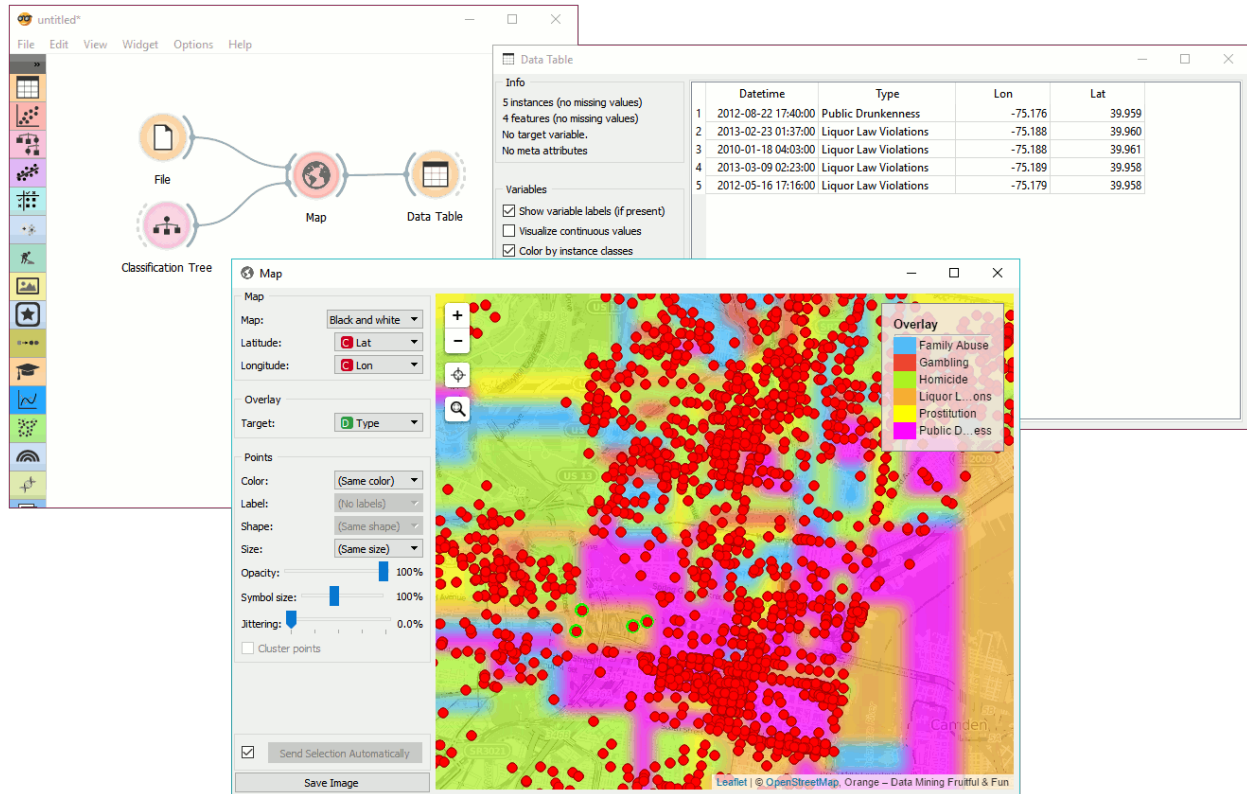
## Nomogram

Nomograms for visualization of Naive Bayes and Logistic Regression classifiers.

## Signals

### Inputs:

- **Classifier**  
A trained classifier (Naive Bayes or Logistic regression).
- **Data**  
Data instance.



## Description

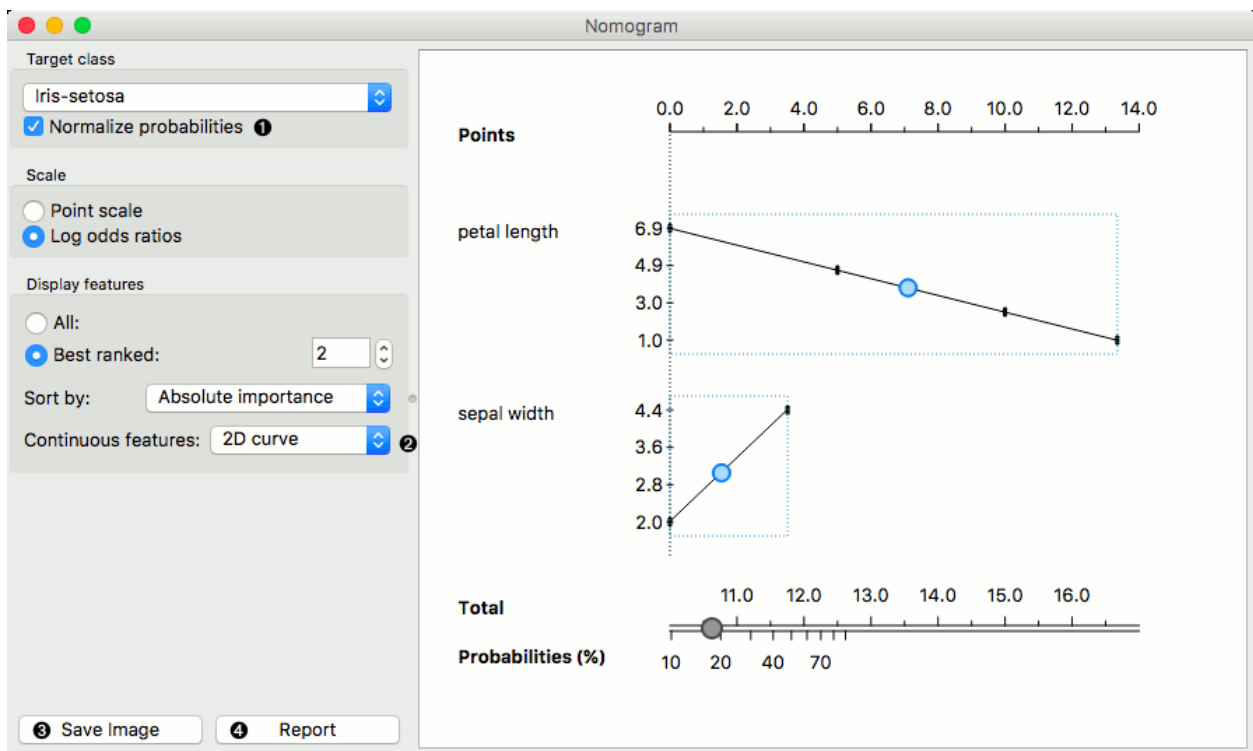
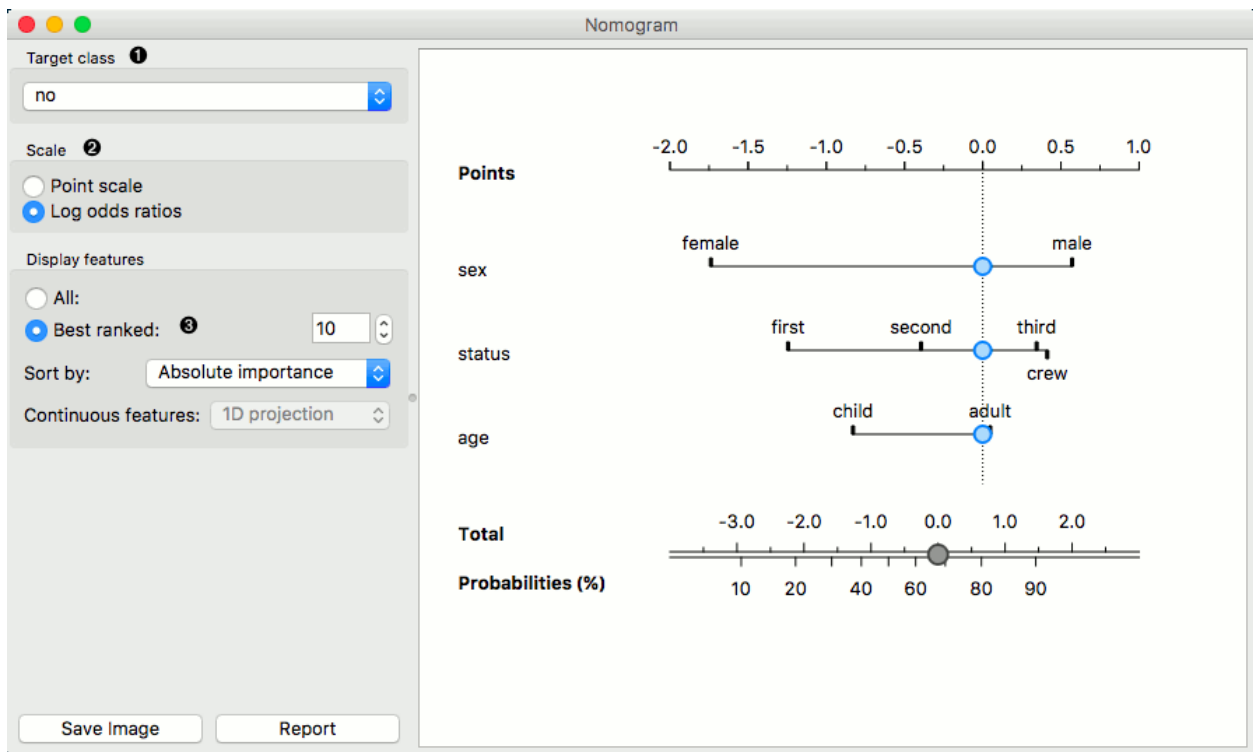
The **Nomogram** enables some classifier's (more precisely Naive Bayes classifier and Logistic Regression classifier) visual representation. It offers an insight into the structure of the training data and effects of the attributes on the class probabilities. Besides visualization of the classifier, the widget offers interactive support to prediction of class probabilities. A snapshot below shows the nomogram of the Titanic data set, that models the probability for a passenger not to survive the disaster of the Titanic.

1. Select the target class you want to model the probability for.
2. By default Scale is set to Log odds ration. For easier understanding and interpretation option Point scale can be used. The unit is obtained by re-scaling the log odds so that the maximal absolute log odds ratio in the nomogram represents 100 points.
3. When there are too many attributes in the plotted data set, you can choose to display only best ranked ones. It is possible to choose from 'No sorting', 'Name', 'Absolute importance', 'Positive influence' and 'Negative influence' for Naive Bayes representation and from 'No sorting', 'Name' and 'Absolute importance' for Logistic Regression representation.

To represent nomogram for Logistic Regression classifier Iris data set is used:

1. The probability for the chosen target class is computed by 1. vs. all principle, which should be taken in consideration when dealing with multiclass data (alternating probabilities do not sum to 1). To avoid this inconvenience, you can choose to normalize probabilities.
2. Continuous attributes can be plotted in 2D (only for Logistic Regression).
3. Save image.
4. Produce a report.

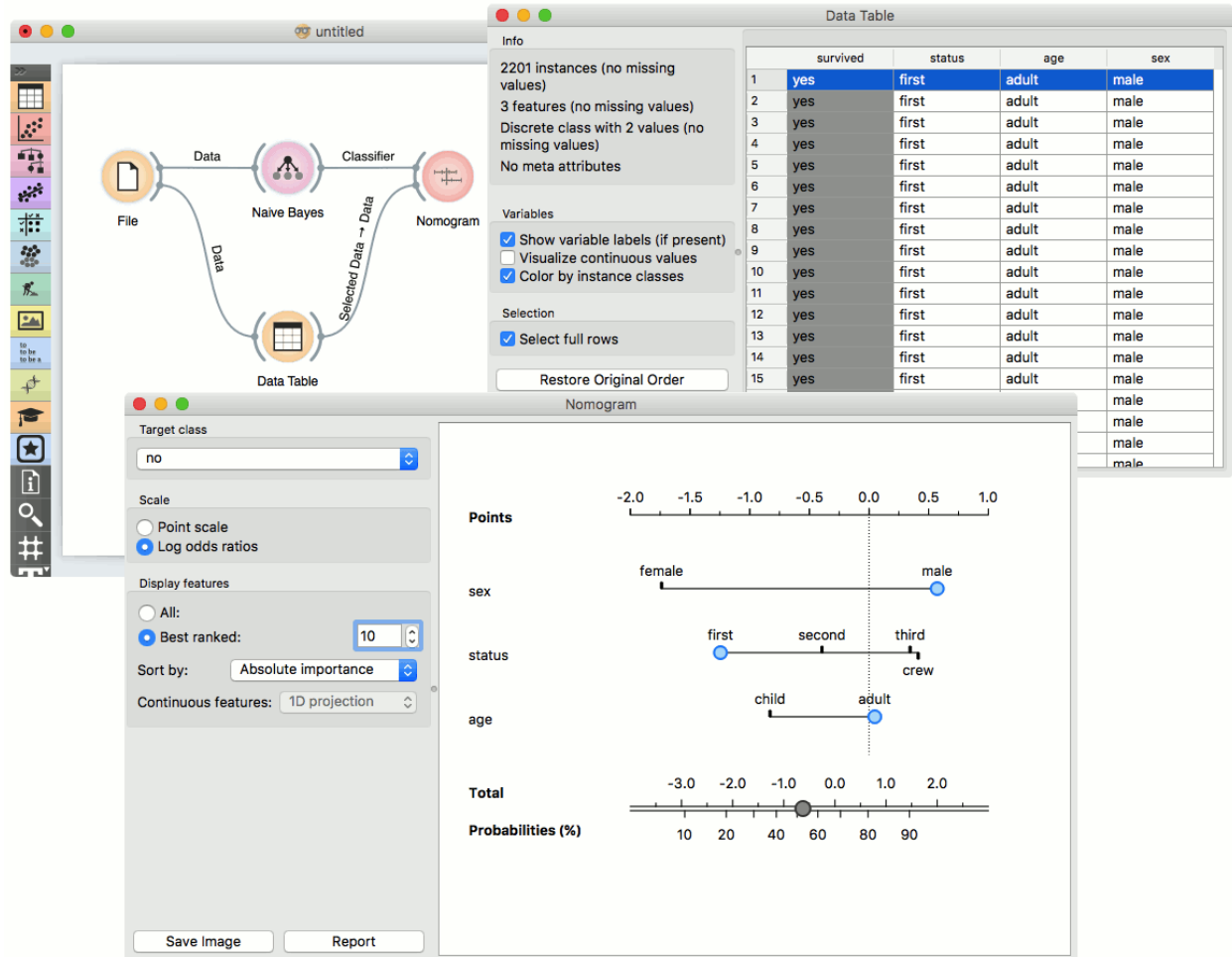






## Example

The **Nomogram** widget should be used immediately after trained classifier widget (e.g. *Naive Bayes*). It can also be passed a data instance using any widget that enables selection (e.g. *Data Table*) as shown in the workflow below.



Referring to the Titanic data set once again, 1490 (68%) of passengers on Titanic, of 2201 in total, died. To make a prediction, the contribution of each attribute is measured as a point score and the individual point scores are summed to determine the probability. When the value of the attribute is unknown, its contribution is 0 points. Therefore, not knowing anything about the passenger, the total point score is 0, and the corresponding probability equals to the unconditional prior. The nomogram in the example shows the case when we know that the passenger is a male adult from the first class. The points sum to -0.36, with a corresponding probability of not surviving of about 53%.

## Classify

### Naive Bayes





## Naive Bayesian Learner

### Signals

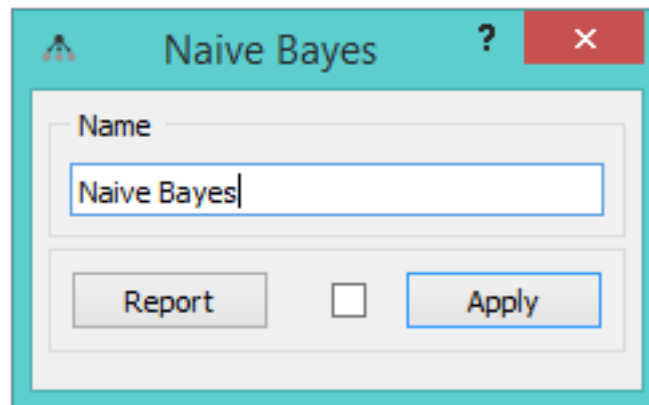
#### Inputs:

- **Data**  
A data set
- **Preprocessor**  
Preprocessed data

#### Outputs:

- **Learner**  
A [Naive Bayesian](#) learning algorithm with settings as specified in the dialog. It can be fed into widgets for testing learners.
- **Naive Bayesian Classifier**  
A trained classifier (a subtype of Classifier). The *Naive Bayesian Classifier* signal sends data only if the learning data (signal **Data**) is present.

### Description



This widget has two options: the name under which it will appear in other widgets and producing a report. The default name is *Naive Bayes*. When you change it, you need to press *Apply*.

### Examples

Here, we present two uses of this widget. First, we compare the results of the **Naive Bayesian learner** with another learner, the [Random Forest](#).

The second schema shows the quality of predictions made with **Naive Bayes**. We feed the [Test&Score](#) widget a Naive Bayes learner and then send the data to the [Confusion Matrix](#). In this widget, we select the misclassified instances and show them in [Scatterplot](#). The bold dots in the scatterplot are the misclassified instances from **Naive Bayes**.

The screenshot displays the Orange Visual Programming interface with a workflow and three configuration windows.

**Workflow:** A workflow is shown in the main window titled "Naive Bayes\*". It includes three widgets: "Random Forest Classification", "Naive Bayes", and "File". All three widgets are connected to a "Test & Score" widget.

**Random Forest Classification Configuration Window:**

- Name:** Random Forest Classification
- Basic Properties:**
  - Number of trees: 10
  - Number of attributes considered at each split: 5
  - Fixed seed for random generator: 0
- Growth Control:**
  - Limit depth of individual trees: 3
  - Do not split subsets smaller than: 5 (checked)
- Buttons:** Report, Apply Automatically (checked)

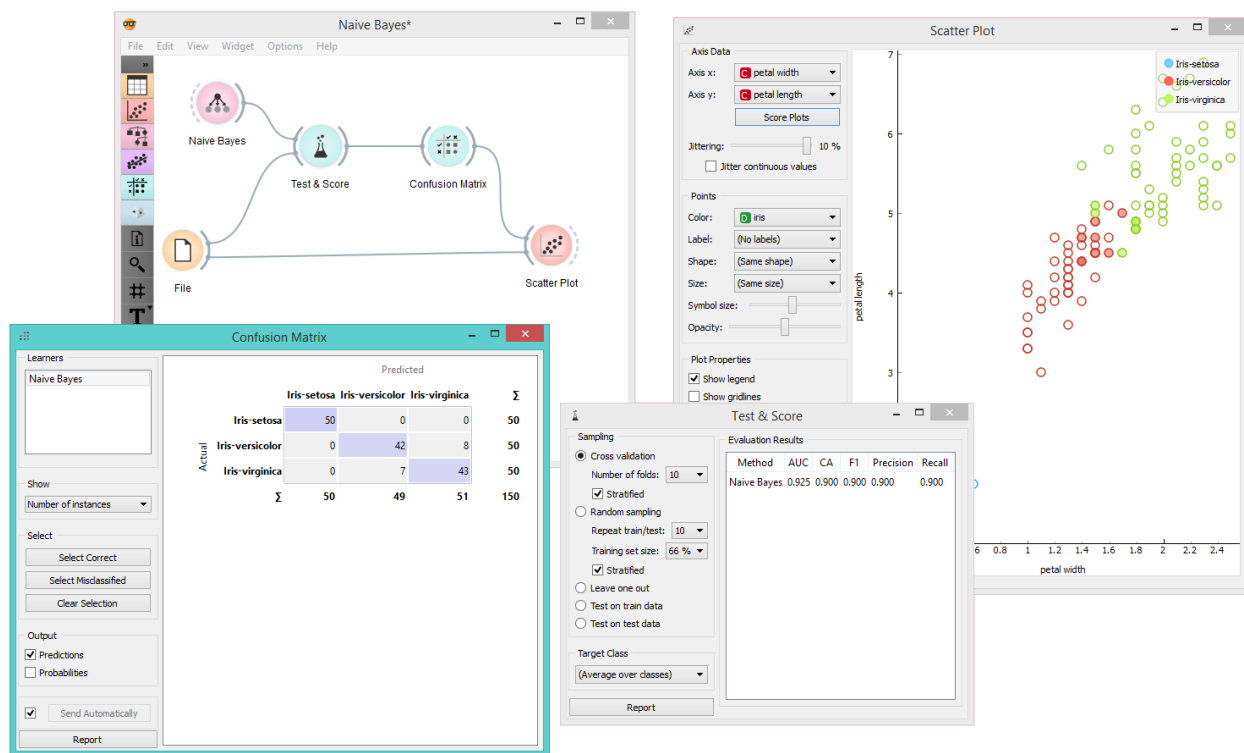
**Naive Bayes Configuration Window:**

- Name:** Naive Bayes
- Buttons:** Report, Apply Automatically (checked)

**Test & Score Configuration Window:**

- Sampling:**
  - Cross validation (selected)
    - Number of folds: 10
    - Stratified (checked)
  - Random sampling
    - Repeat train/test: 10
    - Training set size: 66 %
    - Stratified (checked)
  - Leave one out
  - Test on train data
  - Test on test data
- Target Class:** (Average over classes)
- Buttons:** Report
- Evaluation Results:**

Method	AUC	CA	F1	Precision	Recall
Naive Bayes	0.925	0.900	0.900	0.900	0.900
Random Forest Classification	0.960	0.947	0.947	0.947	0.947



## Logistic Regression

Logistic Regression Learner

### Signals

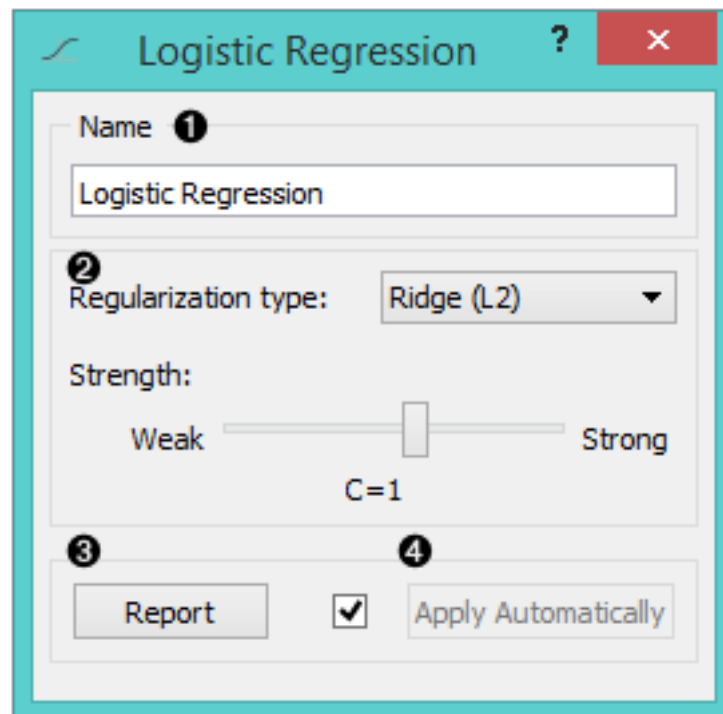
#### Inputs:

- **Data**  
A data set
- **Preprocessor**  
Preprocessed data

#### Outputs:

- **Learner**  
A logistic regression learning algorithm with settings as specified in the dialog.
- **Logistic Regression Classifier**  
A trained classifier (a subtype of Classifier). The **Logistic Regression Classifier** sends data only if data input is present.

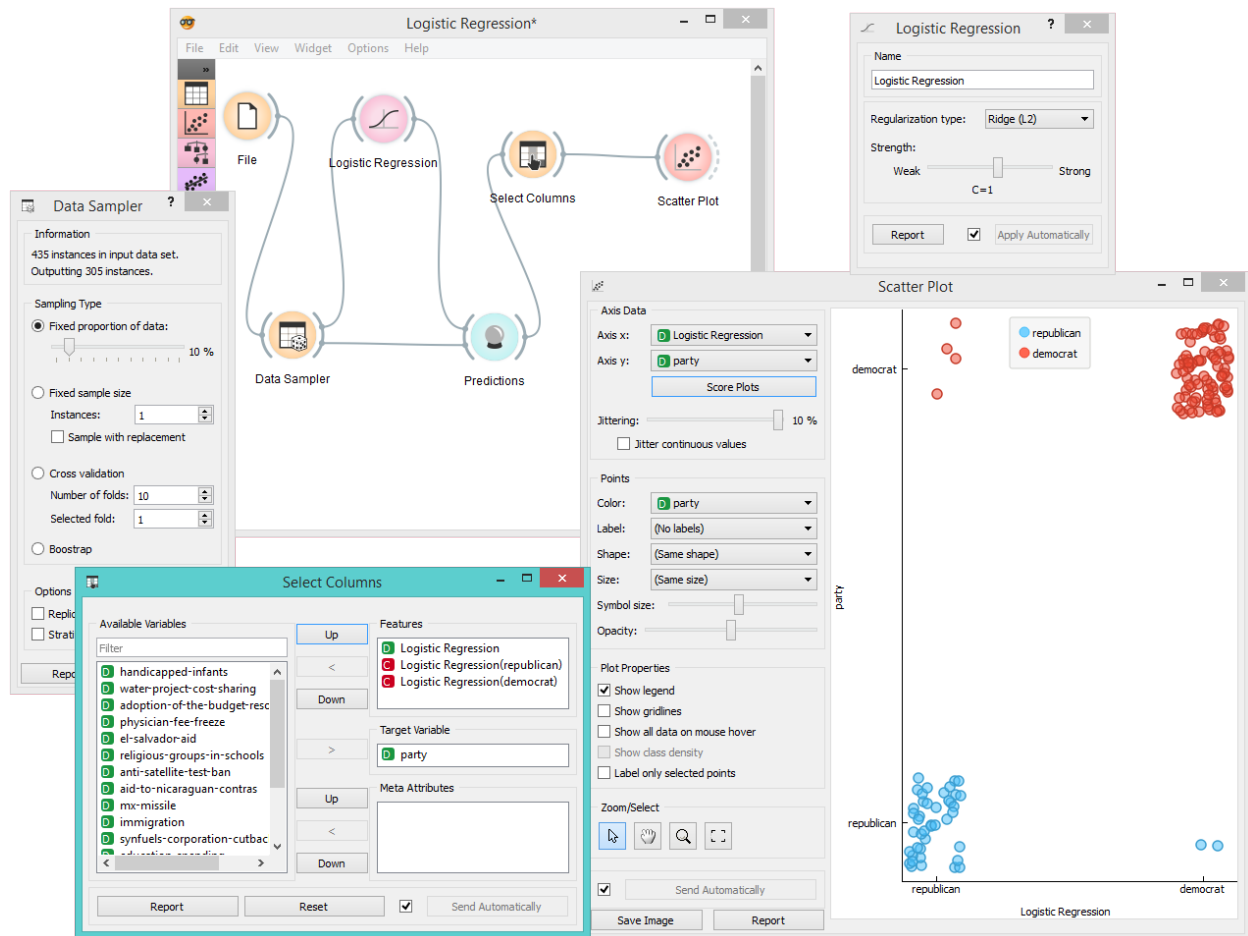
### Description



1. A name under which the learner appears in other widgets. The default name is “Logistic Regression”.
2. [Regularization type](#) (either [L1](#) or [L2](#)). Set the cost strength (default is  $C=1$ ).
3. Press *Apply* to commit changes. If *Apply Automatically* is ticked, changes will be communicated automatically.

## Example

The widget is used just as any other widget for inducing a classifier. This is an example demonstrating the prediction value of logistic regression used on the *voting.tab* data set. We first use the **Logistic Regression** learner to provide a LR classifier for the *Predictions* widget. We want to see the quality of LR prediction model for a person being a republican or a democrat, based on their voting patterns. In *Select Columns* we choose logistic regression as the feature and party as the class. Then we use the *Scatterplot* to see which instances were correctly predicted and which were false.



## Classification Tree



Classification Tree

## Signals

Inputs:

- **Data**

A data set

- **Preprocessor**

Preprocessed data.

**Outputs:**

- **Learner**

A classification tree learning algorithm with settings as specified in the dialog.

- **Classification Tree**

A trained classifier (a subtype of Classifier). The signal *Classification Tree* sends data only if the learning data (signal **Classified Data**) is present.

## Description

**Classification Tree** is a simple classification algorithm that splits the data into nodes by class purity. It is a precursor to *Random Forest*. Classification Tree in Orange is designed in-house and can handle both discrete and continuous data sets.

The screenshot shows the 'Classification Tree' settings dialog. It features a title bar with a help icon and a close button. The main area is divided into sections. The first section, labeled '1', contains a 'Name' text box with the default value 'Classification Tree'. The second section, labeled '2', contains five checked checkboxes: 'Induce binary tree', 'Min. number of instances in leaves' (set to 2), 'Do not split subsets smaller than' (set to 5), 'Stop when majority reaches [%]' (set to 95), and 'Limit the maximal tree depth to' (set to 100). The third section, labeled '3', contains a 'Report' button, a checked checkbox, and an 'Apply Automatically' button.

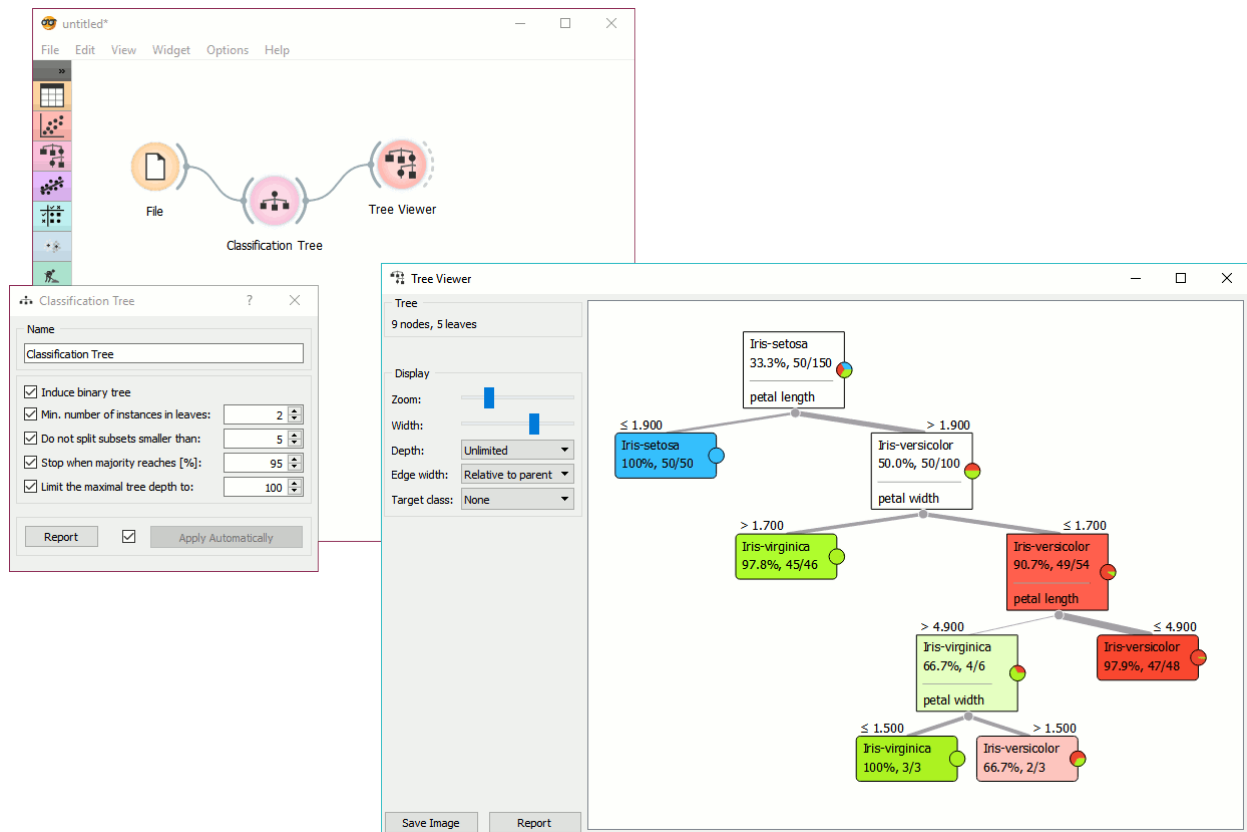
1. The learner can be given a name under which it will appear in other widgets. The default name is “Classification Tree”.
2. Tree parameters: - **Induce binary tree**: build a binary tree (split into two child nodes) - **Min. number of instances in leaves**: if checked, the algorithm will never construct a split which would put less than the specified number of training examples into any of the branches. - **Do not split subsets smaller than**: forbids the algorithm to split the nodes with less than the given number of instances. - **Stop when majority reaches [%]**: stop splitting

the nodes after a specified majority threshold is reached - **Limit the maximal tree depth**: limits the depth of the classification tree to the specified number of node levels.

3. Produce a report. After changing the settings, you need to click *Apply*, which will put the new learner in the output and, if the training examples are given, construct a new classifier and output it as well. Alternatively, tick the box on the left and changes will be communicated automatically.

## Examples

There are two typical uses for this widget. First, you may want to induce a model and check what it looks like. You do it with the schema below; to learn more about it, see the documentation on [Tree Viewer](#).



The second schema checks the nodes of the built tree.

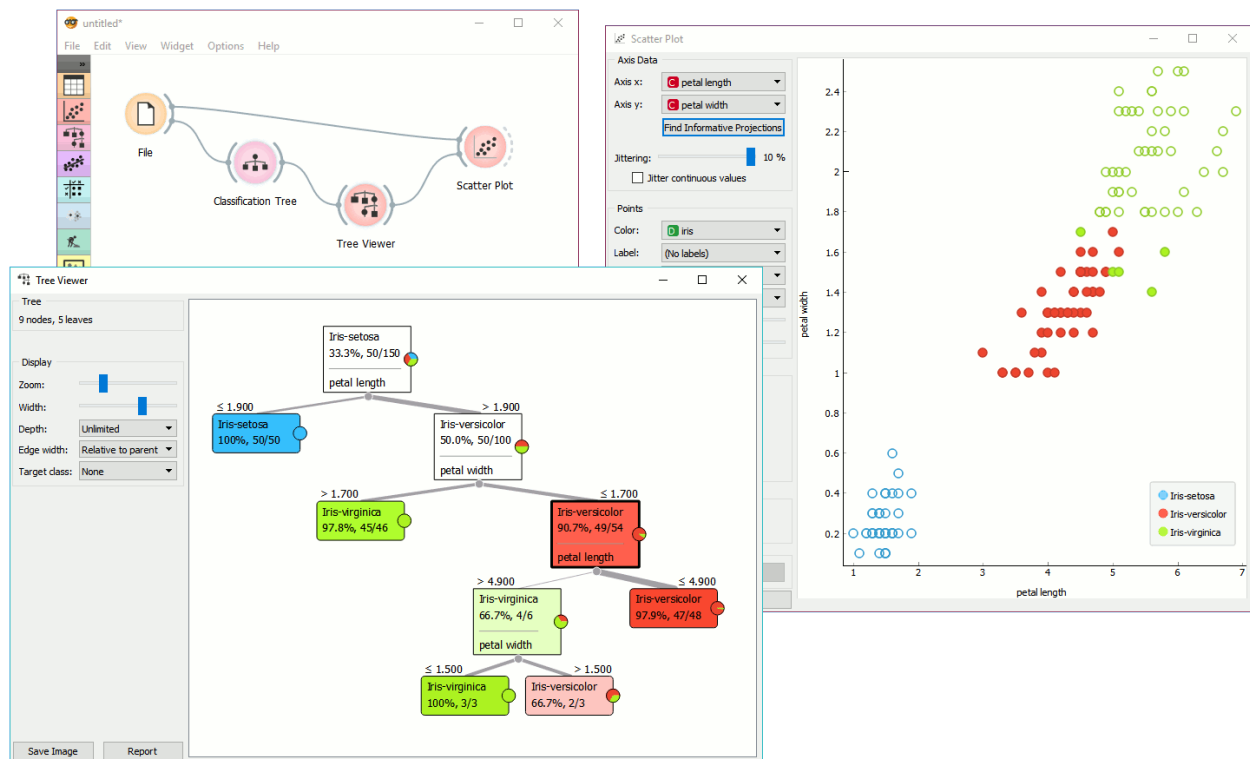
We used the *Iris* data set in both examples.

## Nearest Neighbors

k-Nearest Neighbors (kNN) learner

## Signals

Inputs:





- **Data**

A data set

- **Preprocessor**

Preprocessed data

**Outputs:**

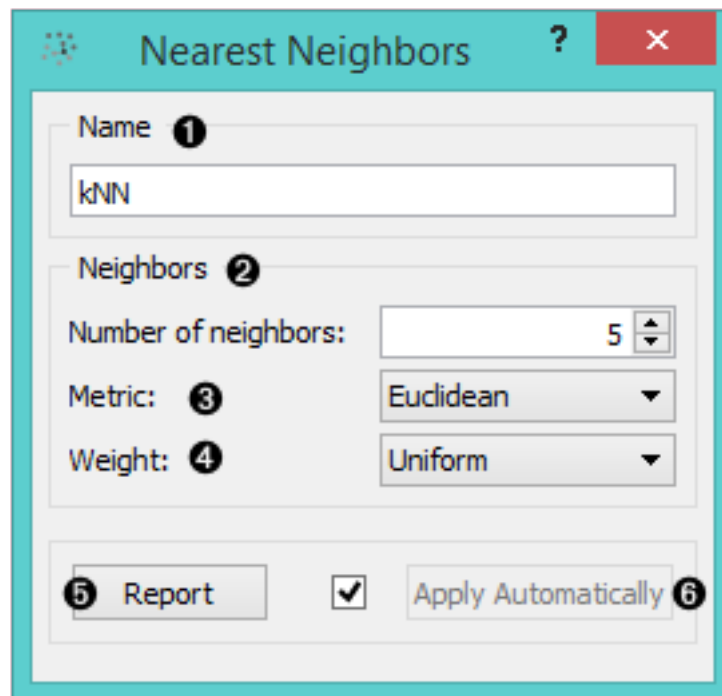
- **Learner**

A kNN learning algorithm with settings as specified in the dialog.

- **kNN Classifier**

Trained classifier (a subtype of Classifier). Signal **kNN Classifier** sends data only if the learning data (signal **Data** is present).

**Description**



1. A name under which it will appear in other widgets. The default name is “kNN”.
2. You can set the *Number of neighbors*.
3. The *Metrics* you can use are:
  - **Euclidean**
  - **Manhattan** (the sum of absolute differences for all attributes)
  - **Chebyshev** (the maximal difference between attributes)
  - **Mahalanobis** (difference between an attribute and the mean).
4. You can assign weight to the contributions of the neighbors. The *Weights* you can use are:
  - **Uniform**: all points in each neighborhood are weighted equally.

- **Distance:** closer neighbors of a query point have a greater influence than the neighbors further away.
- Produce a report.
  - When you change one or more settings, you need to click *Apply*, which will put a new learner in the output and, if the training examples are given, construct a new classifier and output it as well. Changes can also be applied automatically by clicking the box on the left side of the *Apply* button.

## Example

This schema compares the results of **k-Nearest neighbors** with the default classifier, which always predicts the majority class.

The screenshot shows the Orange3 visual programming interface. A workflow is built with three widgets: 'Nearest Neighbors', 'Majority', and 'Test & Score'. The 'Test & Score' widget is open, showing the following settings and results:

**Test & Score Settings:**

- Sampling:**
  - ☒ Cross validation
    - Number of folds: 10
    - ☒ Stratified
  - ☐ Random sampling
    - Repeat train/test: 10
    - Training set size: 66 %
    - ☒ Stratified
  - ☐ Leave one out
  - ☐ Test on train data
  - ☐ Test on test data
- Target Class:** (Average over classes)

**Evaluation Results:**

Method	AUC	CA	F1	Precision	Recall
Majority	0.500	0.333	0.167	0.111	0.333
kNN	0.980	0.973	0.973	0.974	0.973

Below the main interface, two smaller windows are shown:

- Nearest Neighbors:**
  - Name: kNN
  - Neighbors:
    - Number of neighbors: 5
    - Metric: Euclidean
    - Weight: Uniform
  - Buttons: Report, Apply Automatically (checked)
- Majority:**
  - Name: Majority
  - Buttons: Report, Apply Automatically (checked)

## Load Classifier



Loads an existing classifier

## Signals

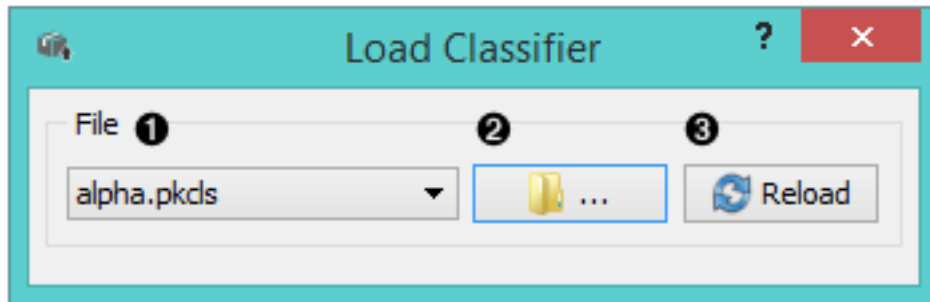
### Inputs:

- None

**Outputs:**

- **Classifier**

A classifier with selected parameters.

**Description**

1. Choose from a list of previously used classifiers.
2. Browse for saved classifiers.
3. Reload the selected classifier.

**Example**

When you want to use a custom-set classifier that you've saved before, open the **Load Classifier** widget and select the desired file with the *Browse* icon. This widget loads the existing classifier into *Predictions* widget for predicting classes.

**Majority**

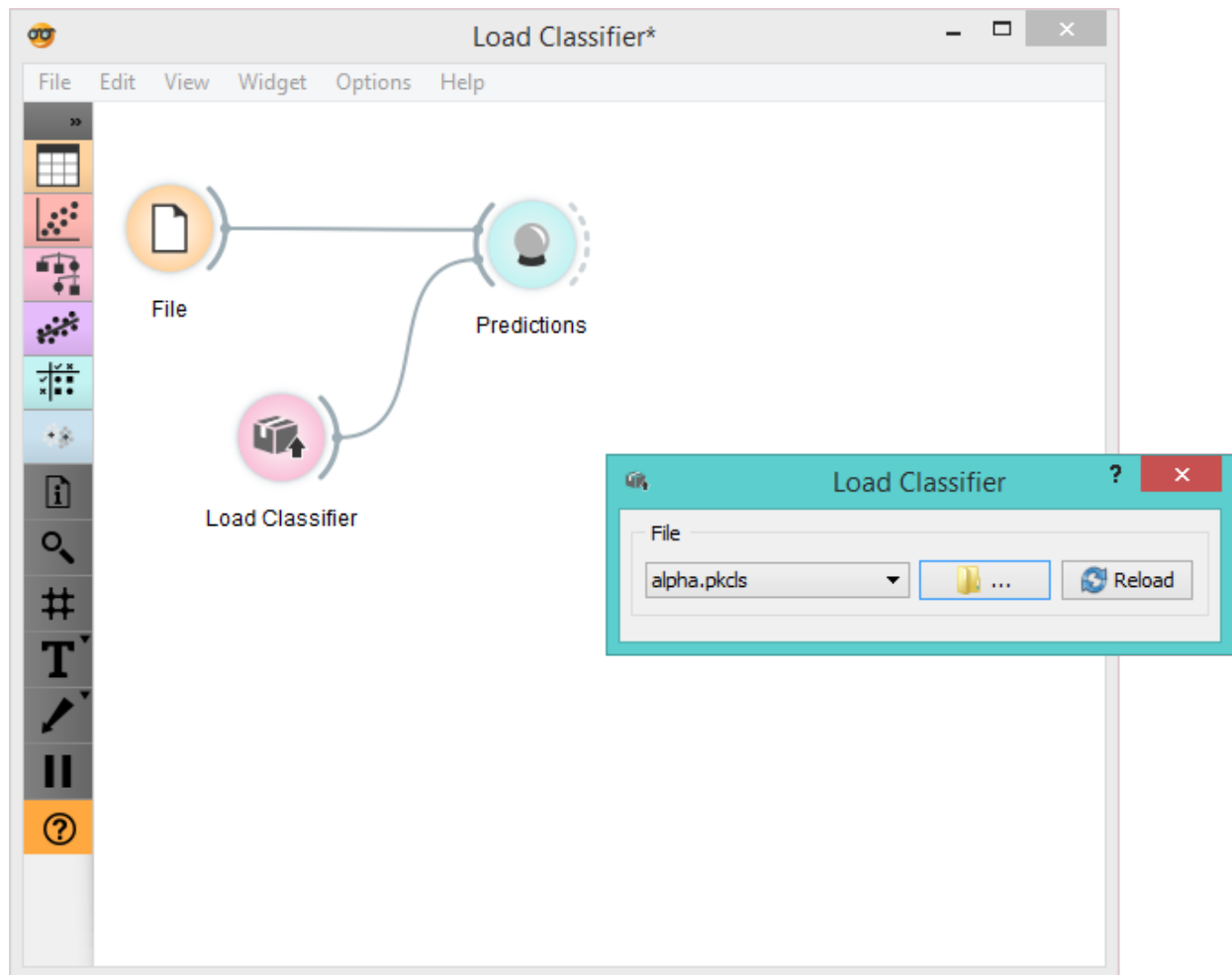
A learner that returns a majority class in a data set for all instances.

**Signals****Inputs:**

- **Data**  
A data set
- **Preprocessor**  
Preprocessed data

**Outputs:**

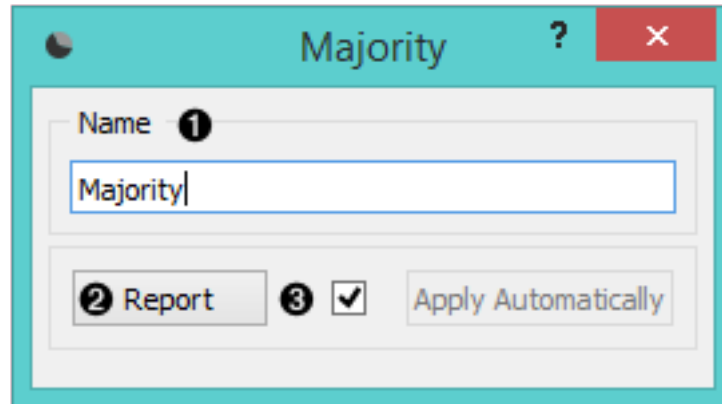
- **Learner**  
A majority learning algorithm
- **Classifier**  
A trained classifier. In the output only if the learning data (signal *Data*) is present.



## Description

This learner produces a classifier that always predicts the **majority** class. When asked for probabilities, it will return the relative frequencies of the classes in the training set. When there are two or more majority classes, the classifier chooses the predicted class randomly, but always returns the same class for a particular example.

The widget is typically used to compare other learning algorithms with the default classification accuracy.



This widget provides the user with two options:

1. The name under which it will appear in other widgets (the default name is “Majority”).
2. Producing a report.

If you change the widget’s name, you need to click *Apply*. Alternatively, tick the box on the left side and changes will be communicated automatically.

## Example

In a typical use of this widget, it would be connected to *Test&Score* to compare the scores of other learning algorithms (such as kNN) with the default scores.

## Random Forest Classification

Random forest learning algorithm.

## Signals

### Inputs:

- **Data**  
A data set
- **Preprocessor**  
Preprocessed data

### Outputs:

- **Learner**  
A random forest learning algorithm with settings as specified in the dialog.

The screenshot displays the Orange Visual Programming interface. A workflow is built with three widgets on the left: 'Nearest Neighbors', 'Majority', and 'File'. All three are connected to a 'Test & Score' widget on the right. Two dialog boxes are open. The 'Majority' dialog box has the name 'Majority' and the 'Apply Automatically' checkbox checked. The 'Test & Score' dialog box shows the 'Sampling' section with 'Cross validation' selected, 'Number of folds' set to 10, 'Stratified' checked, 'Repeat train/test' set to 10, and 'Training set size' set to 66%. The 'Evaluation Results' table is also visible.

**Majority Dialog Box**

Name: Majority

Report ☒ Apply Automatically

**Test & Score Dialog Box**

**Sampling**

- ☒ Cross validation
  - Number of folds: 10
  - ☒ Stratified
- ☐ Random sampling
  - Repeat train/test: 10
  - Training set size: 66 %
  - ☒ Stratified
- ☐ Leave one out
- ☐ Test on train data
- ☐ Test on test data

**Target Class**

(Average over classes)

Report

**Evaluation Results**

Method	AUC	CA	F1	Precision	Recall
kNN	0.980	0.973	0.973	0.974	0.973
Majority	0.500	0.333	0.167	0.111	0.333

- **Random Forest Classifier**

A trained classifier.

## Description

**Random forest** is a classification technique proposed by (Breiman, 2001). When given a set of class-labeled data, **Random Forest** builds a set of classification trees. Each tree is developed from a bootstrap sample from the training data. When developing individual trees, an arbitrary subset of attributes is drawn (hence the term “random”), from which the best attribute for the split is selected. Classification is based on the majority vote from individually developed tree classifiers in the forest.

The screenshot shows the 'Random Forest Classification' widget interface. It features a title bar with a question mark and a close button. The main area is divided into sections: 'Name' (1) with a text field containing 'Random Forest Classification'; 'Basic Properties' (2) with 'Number of trees' set to 10, 'Number of attributes considered at each split' set to 5 (unchecked), and 'Fixed seed for random generator' set to 0 (unchecked); 'Growth Control' (3) with 'Limit depth of individual trees' set to 3 (unchecked) and 'Do not split subsets smaller than' set to 5 (checked); and a bottom section (4) with a 'Report' button (checked) and an 'Apply Automatically' button (5).

1. Specify the name of the learner or classifier. The default name is “*Random Forest Classification*”.
2. Specify how many classification trees will be included in the forest (*Number of trees in the forest*), and how many attributes will be arbitrarily drawn for consideration at each node. If the latter is not specified (option *Consider a number...* left unchecked), this number is equal to the square root of the number of attributes in the data.
3. Original Brieman’s proposal is to grow the trees without any pre-pruning, but since pre-pruning often works quite well and is faster, the user can set the depth to which the trees will be grown (*Limit depth of individual trees*). Another pre-pruning option is to select the smallest subset that can be split (*Do not split subsets smaller than*)
4. Produce a report.

- Click *Apply* to communicate the changes to other widgets. Alternatively, tick the box on the left side of the *Apply* button and changes will be communicated automatically.

## Example

The example below shows a comparison schema of a random forest and a *tree learner* on a specific data set.

The screenshot displays the Orange3 visual programming interface. A workflow is set up with three widgets: 'File', 'Classification Tree', and 'Random Forest Classification', all connected to a 'Test & Score' widget. The 'Test & Score' widget's 'Evaluation Results' table is shown below:

Method	AUC	CA	F1	Precision	Recall
Classification Tree	0.955	0.940	0.940	0.941	0.940
Random Forest Classification	0.965	0.953	0.953	0.953	0.953

Below the workflow, the settings for the 'Classification Tree' and 'Random Forest Classification' widgets are shown. The 'Classification Tree' widget settings include:

- Name: Classification Tree
- Feature Selection: Entropy
- Pruning:
  - ☒ Min. instances in leaves: 2
  - ☒ Stop splitting nodes with less instances than: 5
  - ☒ Limit the depth to: 100
- Buttons: Report, ☒ Apply Automatically

The 'Random Forest Classification' widget settings include:

- Name: Random Forest Classification
- Basic Properties:
  - Number of trees: 10
  - ☐ Number of attributes considered at each split: 5
  - ☐ Fixed seed for random generator: 0
- Growth Control:
  - ☐ Limit depth of individual trees: 3
  - ☒ Do not split subsets smaller than: 5
- Buttons: Report, ☒ Apply Automatically

## References

Breiman, L. (2001). Random Forests. In *Machine Learning*, 45(1), 5-32. Available [here](#)

## Save Classifier



Saves classifier



## Signals

### Inputs:

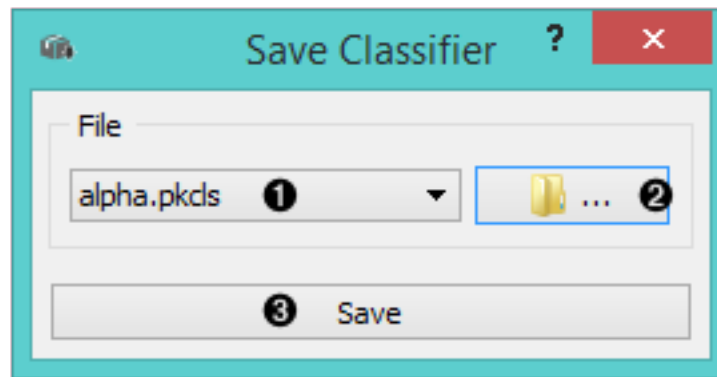
- **Classifier**

A classifier with selected parameters

### Outputs:

- None

## Description



1. Choose from previously saved classifiers.
2. Save the created classifier with the *Browse* icon. Click on the icon and enter the name of the file. The classifier will be saved to a pickled file.

File name:	alpha	▼
Save as type:	Pickle files (*.pickle *.pck)	▼

3. Save the classifier.

## Example

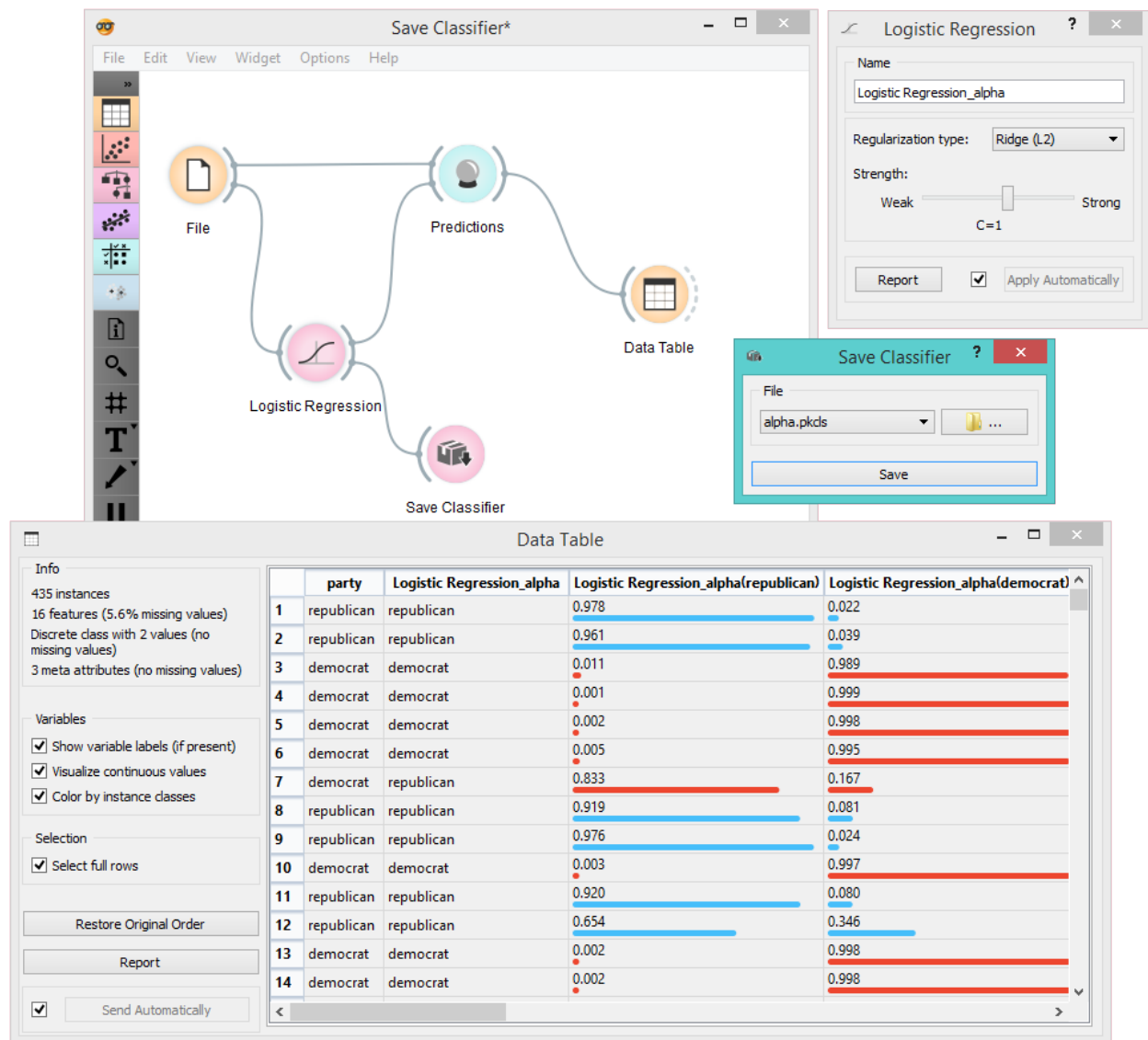
When you want to save a custom-set classifier, select the desired parameters in the classification widget and connect it to *Save Classifier*. Name the classifier; load it later into workflows with data sets containing compatible attributes.

## SVM

Support vector machine learning algorithm.

## Signals

### Inputs:



The screenshot shows the Orange3 visual programming interface. A workflow is built with the following widgets: File, Logistic Regression, Predictions, Data Table, and Save Classifier. The File widget is connected to the Logistic Regression widget, which is connected to the Predictions widget, which is connected to the Data Table widget. The Save Classifier widget is also connected to the Predictions widget.

The Logistic Regression widget's settings are shown in a separate window:

- Name: Logistic Regression\_alpha
- Regularization type: Ridge (L2)
- Strength: Weak (C=1) to Strong
- Buttons: Report, Apply Automatically (checked)

The Save Classifier widget's settings are shown in another window:

- File: alpha.pkcls
- Buttons: Save

The Data Table widget displays the results of the classification. The table has 14 rows and 4 columns: party, Logistic Regression\_alpha, Logistic Regression\_alpha(republican), and Logistic Regression\_alpha(democrat). The table shows the predicted class for each instance and the corresponding probability values.

	party	Logistic Regression_alpha	Logistic Regression_alpha(republican)	Logistic Regression_alpha(democrat)
1	republican	republican	0.978	0.022
2	republican	republican	0.961	0.039
3	democrat	democrat	0.011	0.989
4	democrat	democrat	0.001	0.999
5	democrat	democrat	0.002	0.998
6	democrat	democrat	0.005	0.995
7	democrat	republican	0.833	0.167
8	republican	republican	0.919	0.081
9	republican	republican	0.976	0.024
10	democrat	democrat	0.003	0.997
11	republican	republican	0.920	0.080
12	republican	republican	0.654	0.346
13	democrat	democrat	0.002	0.998
14	democrat	democrat	0.002	0.998

- **Data**

A data set.

- **Preprocessor**

Preprocessed data.

**Outputs:**

- **Learner**

A support vector machine learning algorithm with settings as specified in the dialog.

- **Classifier**

A trained SVM classifier

- **Support Vectors**

A subset of data instances from the training set that were used as support vectors in the trained classifier

## Description

**Support vector machine** (SVM) is a classification technique that separates the attribute space with a hyperplane, thus maximizing the margin between the instances of different classes. The technique often yields supreme predictive performance results. Orange embeds a popular implementation of SVM from the **LIBSVM** package. This widget is its graphical user interface.

1. The learner can be given a name under which it will appear in other widgets. The default name is “SVM”.
2. Classification type with test error settings. *C-SVM* and *v-SVM* are based on different minimization of the error function. On the right side, you can set test error bounds, **Cost** for C-SVM and *Complexity bound* for v-SVM.
3. The next block of options deals with kernel, a function that transforms attribute space to a new feature space to fit the maximum-margin hyperplane, thus allowing the algorithm to create non-linear classifiers with **Polynomial**, **RBF** and **Sigmoid** kernels. Functions that specify the kernel are presented besides their names, and the constants involved are:
  - **g** for the gamma constant in kernel function (the recommended value is  $1/k$ , where  $k$  is the number of the attributes, but since there may be no training set given to the widget the default is 0 and the user has to set this option manually),
  - **c** for the constant  $c_0$  in the kernel function (default 0), and
  - **d** for the degree of the kernel (default 3).
4. Set permitted deviation from the expected value in *Numerical Tolerance*. Tick the box next to *Iteration Limit* to set the maximum number of iterations permitted.
5. Produce a report.
6. Click *Apply* to commit changes. If you tick the box on the left side of the *Apply* button, changes will be communicated automatically.

## Examples

There are two typical uses for this widget, one where the widget is a classifier and the other where it constructs an object for learning. For the first one, we have split our data set into two data subsets (*Sample* and *Remaining Examples*). The sample was sent to SVM which produced a *Classifier*, which was then used in *Predictions* widget to classify the data in *Remaining Examples*. A similar schema can be used if the data is already in two separate files; in this case, two *File* widgets would be used instead of the *File* - *Data Sampler* combination.





The second schema shows how to use the **SVM** widget to construct a learner and compare it in cross-validation with *Majority* and *k-Nearest Neighbours* learners.

The following schema observes a set of support vectors in a *Scatterplot* visualization.

For the above schema to work correctly, the channel between **SVM** and *Scatterplot* widget has to be set appropriately. Set the channel between these two widgets by double-clicking on the connection between the widgets and use the settings as displayed in the dialog below.

## References

Introduction to SVM on StatSoft.

## CN2 Rule Induction

CN2 Rule Induction

## Signals

### Inputs

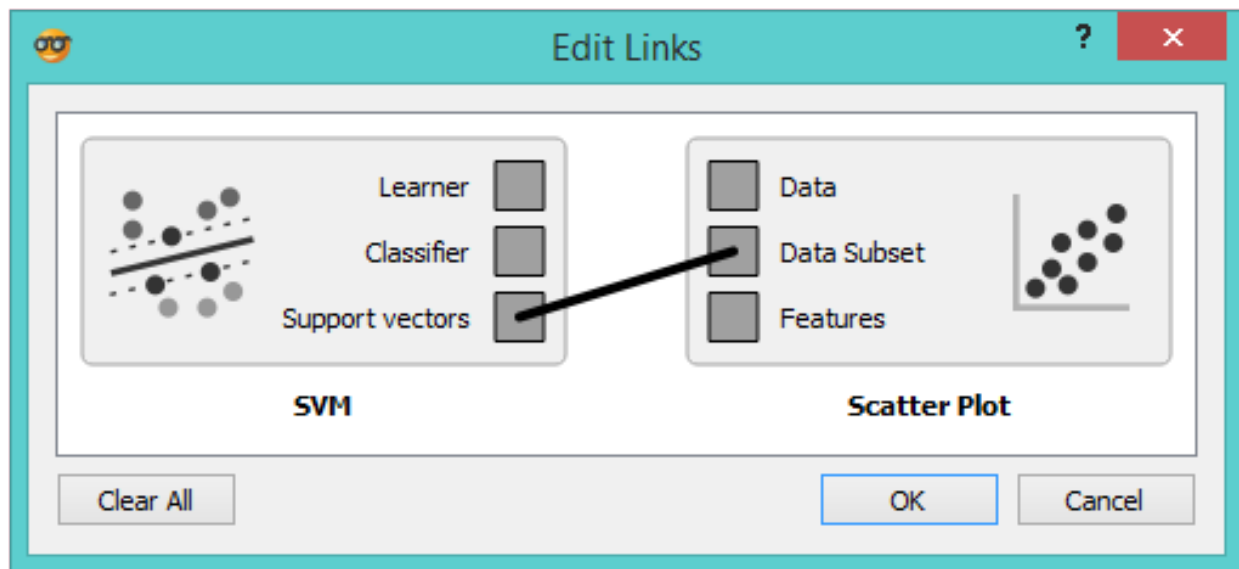
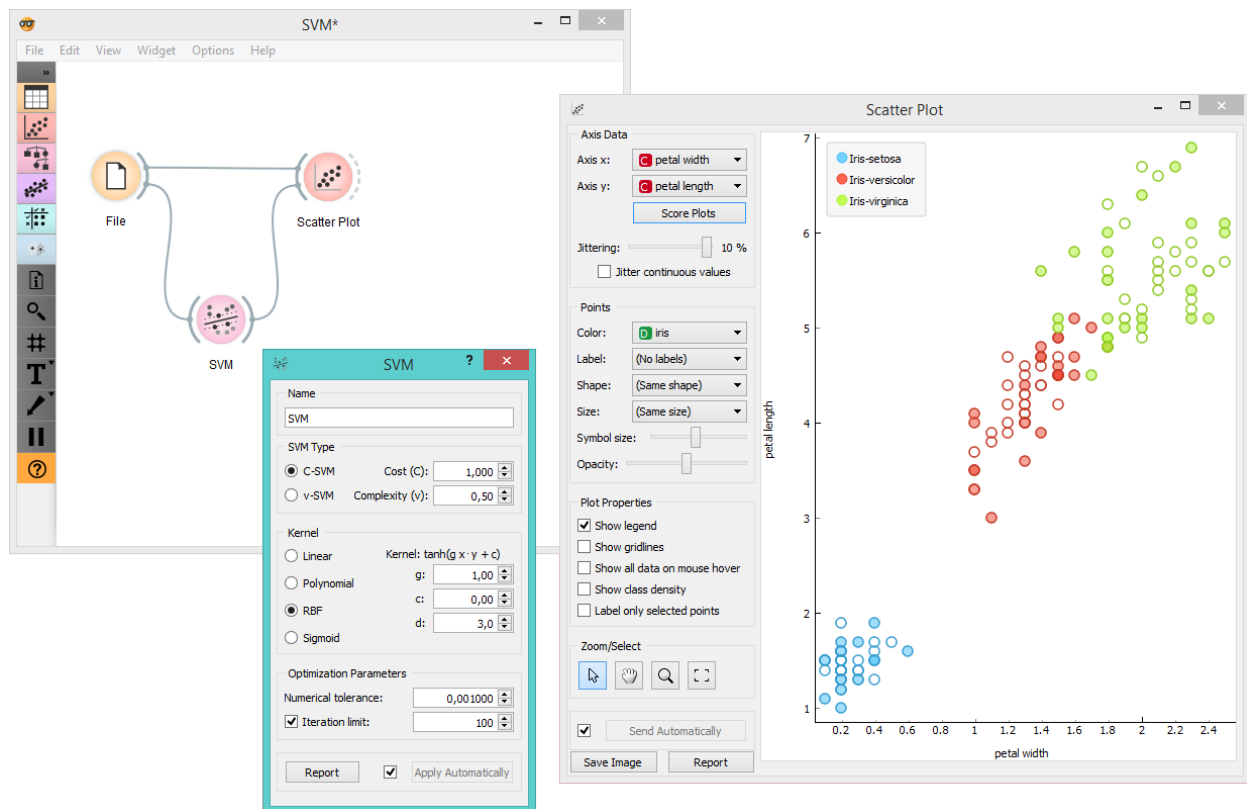
- **Data**  
Data set.
- **Preprocessor**  
Preprocessed data.

The screenshot displays the Orange Visual Programming interface. The main window, titled "SVM\*", shows a workflow with four widgets: "File", "Majority", "Nearest Neighbors", and "SVM", all connected to a "Test & Score" widget. A "Test & Score" settings dialog is open, showing the following configuration:

- Sampling:**
  - ☒ Cross validation
    - Number of folds: 10
    - ☒ Stratified
  - ☐ Random sampling
    - Repeat train/test: 10
    - Training set size: 66 %
    - ☒ Stratified
  - ☐ Leave one out
  - ☐ Test on train data
  - ☐ Test on test data
- Target Class:** (Average over classes)
- Evaluation Results:**

Method	AUC	CA	F1	Precision	Recall
Majority	0.500	0.333	0.167	0.111	0.333
kNN	0.980	0.973	0.973	0.974	0.973
SVM	0.960	0.947	0.947	0.947	0.947

The "Report" button is visible at the bottom of the dialog.



## Outputs

- **Learner**

The CN2 learning algorithm with settings as specified in the dialog.

- **CN2 Rule Classifier**

Trained classifier (a subtype of Classifier). **CN2 Rule Classifier** is only trained and forwarded if learning data is connected.

## Description

The CN2 algorithm is a classification technique designed for the efficient induction of simple, comprehensible rules of form “if *cond* then predict *class*”, even in domains where noise may be present.

**Rule Induction** ? [X]

1 **Name**

CN2 rule inducer

2 **Rule ordering**

☒ Ordered  
☐ Unordered

3 **Covering algorithm**

☒ Exclusive  
☐ Weighted Y: 0,70

4 **Rule search**

Evaluation measure: Entropy  
Beam width: 5

5 **Rule filtering**

Minimum rule coverage: 1  
Maximum rule length: 5  
☐ Statistical significance (default  $\alpha$ ): 1,00  
☐ Relative significance (parent  $\alpha$ ): 1,00

6 **Report** ☐ **Apply**



1. Name under which the learner appears in other widgets. The default name is *CN2 Rule Induction*.
2. *Rule ordering*:
  - **Ordered**; induce ordered rules (decision list). Rule conditions are found and the majority class is assigned in the rule head.
  - **Unordered**; induce unordered rules (rule set). Learn rules for each class individually, in regard to the original learning data.
3. *Covering algorithm*:
  - **Exclusive**; after covering a learning instance, remove it from further consideration.
  - **Weighted**; after covering a learning instance, decrease its weight (multiplication by *gamma*) and in-turn decrease its impact on further iterations of the algorithm.
4. *Rule search*:
  - **Evaluation measure**; select a heuristic to evaluate found hypotheses:
    - (a) **Entropy** (measure of unpredictability of content)
    - (b) Laplace Accuracy
    - (c) Weighted Relative Accuracy
  - **Beam width**; remember the best rule found thus far and monitor a fixed number of alternatives (the beam).
5. *Rule filtering*:
  - **Minimum rule coverage**; found rules must cover at least the minimum required number of covered examples. Unordered rules must cover this many target class examples.
  - **Maximum rule length**; found rules may combine at most the maximum allowed number of selectors (conditions).
  - **Default alpha**; significance testing to prune out most specialised (less frequently applicable) rules in regard to the initial distribution of classes.
  - **Parent alpha**; significance testing to prune out most specialised (less frequently applicable) rules in regard to the parent class distribution.
6. Tick ‘Apply Automatically’ to auto-communicate changes to other widgets and to immediately train the classifier if learning data is connected. Alternatively, press ‘Apply’ after configuration.

## Examples

Having trained the model, induced rules can be quickly reviewed and interpreted. Showcased in the example below is the *CN2 Rule Viewer* widget.

The second schema tests the accuracy of the algorithm, compares its performance to the *Classification Tree* on a specific data set, and presents a standard use of the widget.

## References

1. “Separate-and-Conquer Rule Learning”, Johannes Fürnkranz, Artificial Intelligence Review 13, 3-54, 1999
2. “The CN2 Induction Algorithm”, Peter Clark and Tim Niblett, Machine Learning Journal, 3 (4), pp261-283, (1989)
3. “Rule Induction with CN2: Some Recent Improvements”, Peter Clark and Robin Boswell, Machine Learning - Proceedings of the 5th European Conference (EWSL-91), pp151-163, 1991

The screenshot shows the Orange3 interface with a workflow containing a 'File' widget, a 'CN2 Rule Induction' widget, and a 'CN2 Rule Viewer' widget. The 'CN2 Rule Induction' widget is configured with the following settings:

- Name: CN2 rule inducer
- Rule ordering: Ordered
- Covering algorithm: Exclusive
- Rule search: Evaluation measure: Entropy, Beam width: 5
- Rule filtering: Minimum rule coverage: 1, Maximum rule length: 5, Statistical significance (default  $\alpha$ ): 1.00, Relative significance (parent  $\alpha$ ): 1.00

The 'CN2 Rule Viewer' widget displays the following table:

	IF conditions	THEN class	Distribution	Probabilities [%]	Quality	Length
0	feathersz=0	type=bird	[0, 20, 0, 0, 0, 0]	4: 78: 4: 4: 4: 4	-0.00	1
1	milkz=0	type=mammal	[0, 0, 0, 0, 41, 0]	2: 2: 2: 2: 88: 2	-0.00	1
2	hairz=0	type=insect	[0, 0, 0, 4, 0, 0]	9: 9: 9: 45: 9: 9	-0.00	1
3	airbornz=0	type=insect	[0, 0, 0, 2, 0, 0]	11: 11: 11: 33: 11: 11	-0.00	1
4	finsz=0	type=fish	[0, 0, 13, 0, 0, 0]	5: 5: 70: 5: 5: 5	-0.00	1
5	legsz=5	type=invertebrate	[0, 0, 0, 0, 1, 0]	12: 12: 12: 12: 25: 12	-0.00	1
6	legsz=8	type=invertebrate	[0, 0, 0, 0, 2, 0]	11: 11: 11: 11: 33: 11	-0.00	1
7	eggz=0	type=reptile	[0, 0, 0, 0, 0, 1]	12: 12: 12: 12: 12: 25	-0.00	1
8	breathesz=0	type=invertebrate	[0, 0, 0, 0, 5, 0]	8: 8: 8: 8: 50: 8	-0.00	1
9	aquaticz=0	type=amphibian	[4, 0, 0, 0, 0, 0]	45: 9: 9: 9: 9: 9	-0.00	1
10	predatorz=0	type=reptile	[0, 0, 0, 0, 0, 3]	10: 10: 10: 10: 10: 40	-0.00	1
11	backbonez=0	type=reptile	[0, 0, 0, 0, 0, 1]	12: 12: 12: 12: 12: 25	-0.00	1
12	legsz=0	type=invertebrate	[0, 0, 0, 0, 2, 0]	11: 11: 11: 11: 33: 11	-0.00	1
13	TRUE	type=insect	[0, 0, 0, 2, 0, 0]	11: 11: 11: 33: 11: 11	-0.00	0

The 'CN2 Rule Viewer' widget also has buttons for 'Restore original order', 'Compact view', and 'Report'.

The screenshot shows the Orange3 interface with a workflow containing a 'File' widget, a 'CN2 Rule Induction' widget, a 'Test & Score' widget, and a 'Classification Tree' widget. The 'CN2 Rule Induction' widget is configured with the following settings:

- Name: CN2 rule inducer
- Rule ordering: Ordered
- Covering algorithm: Exclusive
- Rule search: Evaluation measure: Entropy, Beam width: 5
- Rule filtering: Minimum rule coverage: 1, Maximum rule length: 5, Statistical significance (default  $\alpha$ ): 1.00, Relative significance (parent  $\alpha$ ): 1.00

The 'Test & Score' widget is configured with the following settings:

- Sampling: Cross validation, Number of folds: 10, Stratified: checked, Repeat train/test: 10, Training set size: 66 %, Stratified: checked
- Target Class: (Average over classes)

The 'Test & Score' widget displays the following table:

Method	AUC	CA	F1	Precision	Recall
CN2 rule inducer	0.982	0.970	0.969	0.969	0.970
Classification Tree	0.936	0.901	0.898	0.902	0.901

The 'Classification Tree' widget is configured with the following settings:

- Name: Classification Tree
- Feature Selection: Entropy
- Pruning: Min. instances in leaves: 2, Stop splitting nodes with less instances than: 5, Limit the depth to: 100

4. “Subgroup Discovery with CN2-SD”, Nada Lavrač et al., Journal of Machine Learning Research 5 (2004), 153-188, 2004

## AdaBoost



An ensemble meta-algorithm that combines multiple weak learners to build to build more accurate prediction models.

### Signals

#### Inputs:

- **Data**  
A data set.
- **Preprocessor**  
Preprocessed data.
- **Learner**  
A learning algorithm.

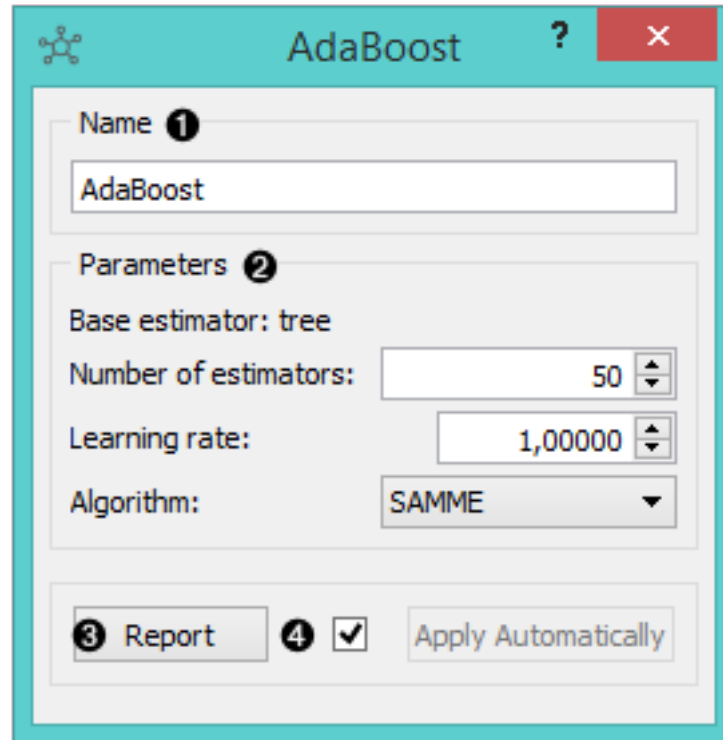
#### Outputs:

- **Learner**  
[AdaBoost](#) learning algorithm with settings as specified in the dialog.
- **Classifier**  
Trained classifier (a subtype of Classifier). The *AdaBoost classifier* signal sends data only if the learning data (signal Data) is present.

### Description

The **AdaBoost** (short for “Adaptive boosting”) widget is a machine-learning algorithm, formulated by [Yoav Freund](#) and [Robert Schapire](#). It can be used with other learning algorithms to boost their performance. It does so by tweaking the weak learners.

1. The learner can be given a name under which it will appear in other widgets. The default name is “AdaBoost”.
2. Set the parameters. The base estimator is a tree and you can set:
  - the **Number of estimators**
  - the **Learning rate**: it determines to what extent the newly acquired information will override the old information (0 = the agent will not learn anything, 1 = the agent considers only the most recent information)
  - the **Algorithm**: SAMME (updates base estimator’s weights with classification results) or SAMME.R. (updates base estimator’s weight with probability estimates)
3. Produce a report.



4. Click *Apply* after changing the settings. That will put the new learner in the output and, if the training examples are given, construct a new classifier and output it as well. To communicate changes automatically tick *Apply Automatically*.

## Examples

For our first example, we loaded the *Iris* data set and compared the results of two different classification algorithms against the *AdaBoost* widget.

For our second example, we loaded the *Iris* data set, sent the data instances to several different classifiers (**AdaBoost**, *Classification Tree*, *Logistic Regression*) and output them in the *Predictions* widget.

## Regression

### Linear Regression

Learns a linear function of its input data.

### Signals

#### Inputs:

- **Data**  
A data set

**Test & Score Configuration**

**Sampling**

- ☒ Cross validation
  - Number of folds: 10
  - ☒ Stratified
- ☐ Random sampling
  - Repeat train/test: 10
  - Training set size: 66 %
  - ☒ Stratified
- ☐ Leave one out
- ☐ Test on train data
- ☐ Test on test data

**Target Class**

(Average over classes)

**Evaluation Results**

Method	AUC	CA	F1	Precision	Recall
AdaBoost	0.950	0.933	0.933	0.934	0.933
Classification Tree	0.955	0.940	0.940	0.940	0.940
Logistic Regression	0.970	0.960	0.960	0.962	0.960

Report

**Predictions Configuration**

**Info**

Data: 150 instances.  
Predictors: 3  
Task: Classification

[Restore Original Order](#)

**Options (classification)**

- ☒ Show predicted class
- ☒ Show predicted probabilities

Iris-setosa  
Iris-versicolor  
Iris-virginica

- ☒ Draw distribution bars

**Data View**

- ☒ Show full data set

**Output**

- ☒ Original data
- ☒ Predictions
- ☒ Probabilities

Report

**Predictions Data Table**

	AdaBoost	Classification Tree	Logistic Regression	iris	sepal length
1	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.88 : 0.12 : 0.00 → Iris-setosa	Iris-setosa	5.100
2	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.80 : 0.20 : 0.00 → Iris-setosa	Iris-setosa	4.900
3	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.85 : 0.15 : 0.00 → Iris-setosa	Iris-setosa	4.700
4	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.83 : 0.17 : 0.00 → Iris-setosa	Iris-setosa	4.600
5	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.90 : 0.10 : 0.00 → Iris-setosa	Iris-setosa	5.000
6	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.93 : 0.07 : 0.00 → Iris-setosa	Iris-setosa	5.400
7	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.90 : 0.10 : 0.00 → Iris-setosa	Iris-setosa	4.600
8	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.86 : 0.14 : 0.00 → Iris-setosa	Iris-setosa	5.000
9	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.80 : 0.20 : 0.00 → Iris-setosa	Iris-setosa	4.400
10	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.80 : 0.20 : 0.00 → Iris-setosa	Iris-setosa	4.900
11	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.89 : 0.11 : 0.00 → Iris-setosa	Iris-setosa	5.400
12	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.86 : 0.14 : 0.00 → Iris-setosa	Iris-setosa	4.800
13	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.79 : 0.21 : 0.00 → Iris-setosa	Iris-setosa	4.800
14	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.84 : 0.16 : 0.00 → Iris-setosa	Iris-setosa	4.300
15	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.93 : 0.07 : 0.00 → Iris-setosa	Iris-setosa	5.800
16	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.96 : 0.04 : 0.00 → Iris-setosa	Iris-setosa	5.700
17	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.94 : 0.06 : 0.00 → Iris-setosa	Iris-setosa	5.400
18	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.89 : 0.11 : 0.00 → Iris-setosa	Iris-setosa	5.100
19	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.90 : 0.10 : 0.00 → Iris-setosa	Iris-setosa	5.700
20	0.45 : 0.27 : 0.27 → Iris-setosa	1.00 : 0.00 : 0.00 → Iris-setosa	0.92 : 0.08 : 0.00 → Iris-setosa	Iris-setosa	5.100



- **Preprocessor**

A preprocessed data set.

**Outputs:**

- **Learner**

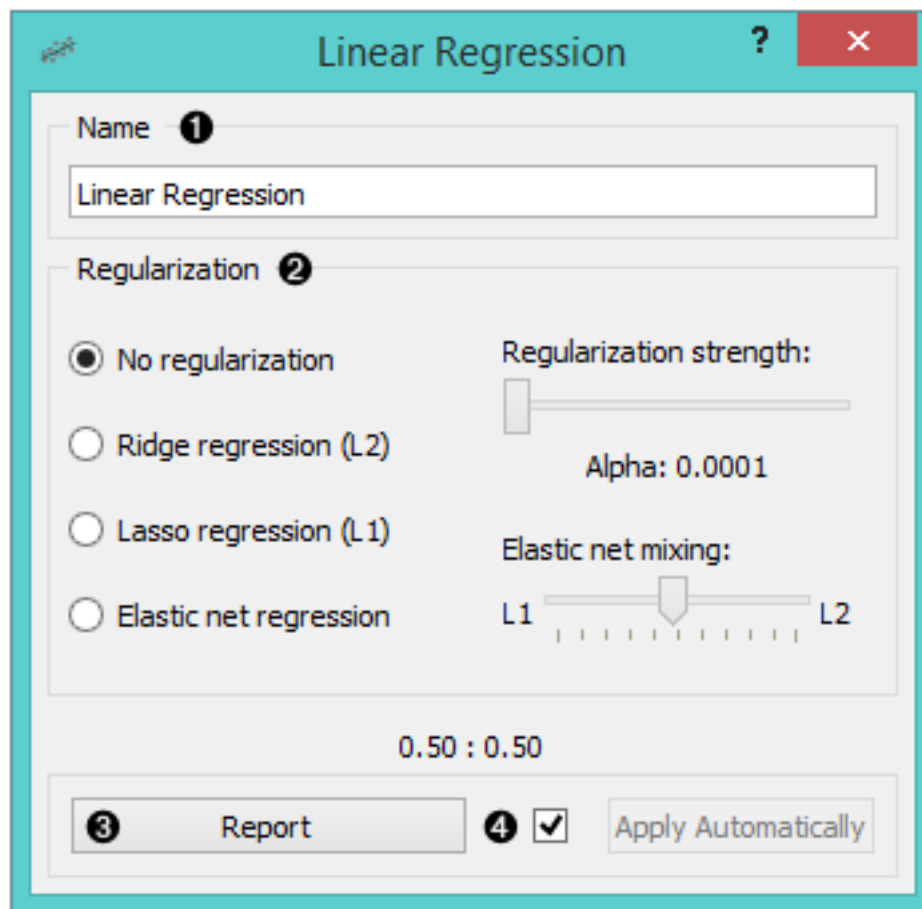
A learning algorithm with the supplied parameters

- **Predictor**

A trained regressor. Signal *Predictor* sends the output signal only if signal *Data* is present.

## Description

The **Linear Regression** widget constructs a learner/predictor that learns a [linear function](#) from its input data. The model can identify the relationship between a predictor  $x_i$  and the response variable  $y$ . Additionally, [Lasso](#) and [Ridge](#) regularization parameters can be specified. Lasso regression minimizes a penalized version of the least squares loss function with L1-norm penalty and Ridge regularization with L2-norm penalty.



1. The learner/predictor name
2. Choose a model to train:
  - no regularization
  - a [Ridge](#) regularization (L2-norm penalty)

- a **Lasso** bound (L1-norm penalty)
  - an **Elastic net** regularization
3. Produce a report.
  4. Press *Apply* to commit changes. If *Apply Automatically* is ticked, changes are committed automatically.

## Example

Below, is a simple workflow showing how to use both the *Predictor* and the *Learner* output. We used the *Housing* data set. For the *Predictor*, we input the prediction model into the *Predictions* widget and view the results in the *Data Table*. For the *Learner*, we can compare different learners in the *Test&Score* widget.

The screenshot shows an Orange3 workflow and several widget settings windows.

**Workflow:** A 'File' widget connects to a 'Linear Regression' widget. The 'Linear Regression' widget connects to both a 'Predictions' widget and a 'Test & Score' widget. The 'Predictions' widget connects to a 'Data Table' widget.

**Linear Regression\* Widget Settings:**

- Info:** 506 instances (no missing values), 13 features (no missing values), Continuous target variable (no missing values), 1 meta attribute (no missing values).
- Variables:** ☒ Show variable labels (if present), ☐ Visualize continuous values, ☐ Color by instance classes.
- Selection:** ☒ Select full rows.
- Buttons:** Restore Original Order, Report, ☒ Send Automatically.

**Data Table Widget:**

	MEDV	Linear Regression	CRIM	ZN
1	24.000	30.004	0.006	18.000
2	21.600	25.026	0.027	0.000
3	34.700	30.568	0.027	0.000
4	33.400	28.607	0.032	0.000
5	36.200	27.944	0.069	0.000
6	28.700	25.256	0.030	0.000
7	22.900	23.002	0.088	12.500
8	27.100	19.536	0.145	12.500
9	16.500	11.524	0.211	12.500
10	18.900	18.920	0.170	12.500
11	15.000	18.999	0.225	12.500
12	18.900	21.587	0.117	12.500
13	21.700	20.907	0.094	12.500
14	20.400	19.553	0.630	0.000

**SVM Regression Widget Settings:**

- Name:** SVM Regression
- SVR Type:** ☒  $\epsilon$ -SVR, ☐  $\nu$ -SVR.
- Cost (C):** 1.00
- Loss epsilon ( $\epsilon$ ):** 0.10
- Complexity bound ( $\nu$ ):** 0.50
- Kernel:** ☐ Linear, ☒ Polynomial, ☐ RBF, ☐ Sigmoid.
- Kernel:**  $(g \cdot x \cdot y + c)^d$
- Optimization Parameters:** Numerical tolerance: 0.001000
- Buttons:** Report, ☐ Apply

**Linear Regression Widget Settings:**

- Name:** Linear Regression
- Regularization:** ☒ No regularization, ☐ Ridge regression (L2), ☐ Lasso regression (L1), ☐ Elastic net regression.
- Regularization strength:** Alpha: 0.0001
- Elastic net mixing:** L1: 0.50, L2: 0.50
- Buttons:** Report, ☒ Apply Automatically

**Predictions Widget:**

- Info:** Data: 506 instances, Predictors: 1, Task: Regression.
- Buttons:** Restore Original Order
- Data View:** ☐ Show full data set
- Output:** ☒ Original data, ☒ Predictions, ☒ Probabilities.
- Buttons:** Report

**Test & Score Widget:**

- Sampling:** ☒ Cross validation, ☐ Random sampling.
- Number of folds:** 10
- ☒ Stratified
- Repeat train/test:** 10
- Training set size:** 66 %
- ☒ Stratified
- ☐ Leave one out
- ☐ Test on train data
- ☐ Test on test data
- Buttons:** Report

**Evaluation Results:**

Method	MSE	RMSE	MAE	R2
Linear Regression	23.370	4.834	3.376	0.723
SVM Regression	31.708	5.631	3.547	0.624

## Mean Learner



Learns the mean of its input data.

## Signals

### Inputs:

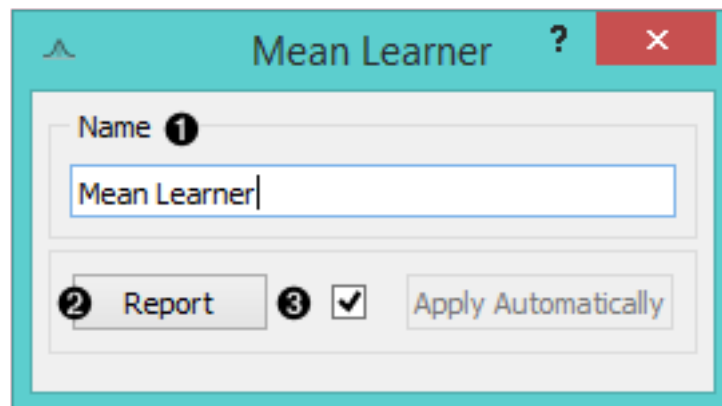
- **Data**  
A data set.
- **Preprocessor**  
Preprocessed data.

### Outputs:

- **Learner**  
A mean learning algorithm.
- **Predictor**  
A trained regressor. Signal *Predictor* sends the regressor only if signal *Data* is present.

## Description

This is the simplest learner widget for regression problems. It *learns* the mean of the class variable and returns a predictor with the same *mean value*. Due to its accuracy, this widget can serve as a baseline for other regression models.



1. Learner/predictor name
2. Produce a report.
3. The *Apply* button commits changes to the output. Alternatively, tick the box on the left side of the button to apply changes automatically.

## Examples

In the first example, we use **Mean Learner** to construct a predictor and input it into the *Data Table*. We used the *housing* data set. In the table, you can see an extra column *Mean Learner* with one (mean) value for all instances.

Another way to use **Mean Learner** is to compare it to other learners in the *Test&Score* widget.



The screenshot displays the Orange Visual Programming interface with a workflow consisting of three widgets: File, Predictions, and Data Table, connected in a sequence. A Mean Learner widget is also present, connected to the Predictions widget. The Predictions widget's output is connected to the Data Table widget. The Mean Learner widget's output is connected to the Predictions widget. The Predictions widget's output is connected to the Data Table widget. The Data Table widget's output is connected to the Mean Learner widget.

The **Mean Learner** widget's configuration window is open, showing the following settings:

- Name: Mean Learner
- Report: ☒ Apply Automatically

The **Predictions** widget's configuration window is open, showing the following settings:

- Info: Data: 506 instances, Predictors: 1, Task: Regression
- Data View: ☐ Show full data set
- Output: ☒ Original data, ☒ Predictions, ☒ Probabilities

The **Data Table** widget's configuration window is open, showing the following settings:

- Info: 506 instances (no missing values), 13 features (no missing values), Continuous target variable (no missing values), 1 meta attribute (no missing values)
- Variables: ☒ Show variable labels (if present), ☐ Visualize continuous values, ☒ Color by instance classes
- Selection: ☒ Select full rows
- Restore Original Order: ☐ Report: ☐ Send Automatically: ☒

The **Predictions** widget's output table is shown below:

	Mean Learner	MEDV
1	22.533	24.000
2	22.533	21.600
3	22.533	34.700
4	22.533	33.400
5	22.533	36.200
6	22.533	28.700
7	22.533	22.900
8	22.533	27.100
9	22.533	16.500
10	22.533	18.900
11	22.533	15.000

The **Data Table** widget's output table is shown below:

	MEDV	Mean Learner	CRIM	ZN
1	24.000	22.533	0.006	18.000
2	21.600	22.533	0.027	0.000
3	34.700	22.533	0.027	0.000
4	33.400	22.533	0.032	0.000
5	36.200	22.533	0.069	0.000
6	28.700	22.533	0.030	0.000
7	22.900	22.533	0.088	12.500
8	27.100	22.533	0.145	12.500
9	16.500	22.533	0.211	12.500
10	18.900	22.533	0.170	12.500
11	15.000	22.533	0.225	12.500
12	18.900	22.533	0.117	12.500
13	21.700	22.533	0.094	12.500
14	20.400	22.533	0.630	0.000

The screenshot displays the Orange3 visual programming environment. A workflow is set up with three input widgets (File, Mean Learner, and Linear Regression) connected to a single 'Test & Score' widget. The 'Test & Score' widget's 'Evaluation Results' table provides a comparison of the two models:

Method	MSE	RMSE	MAE	R2
Mean Learner	84.644	9.200	6.662	-0.003
Linear Regression	23.370	4.834	3.376	0.723

Three configuration windows are overlaid on the main interface:

- Mean Learner Configuration:** Shows the widget name as 'Mean Learner' with 'Report' and 'Apply Automatically' options.
- Linear Regression Configuration:** Shows regularization options (No regularization, Ridge regression (L2), Lasso regression (L1), Elastic net regression) and a regularization strength slider set to Alpha: 70. It also displays an R2 score of 0.67 : 0.33.
- Linear Regression Icon:** A circular icon representing the Linear Regression widget.

## Nearest Neighbors

Predicts according to the nearest training instances.

### Signals

#### Inputs:

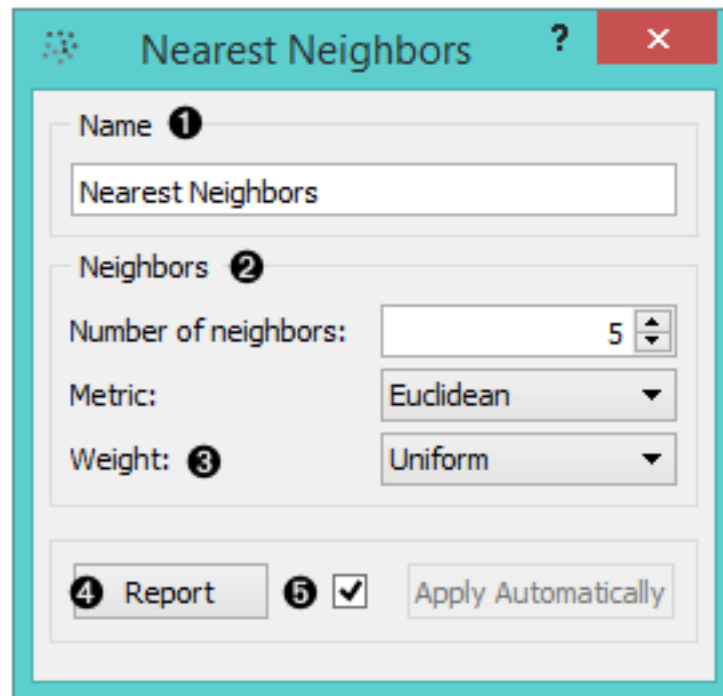
- **Data**  
A data set
- **Preprocessor**  
Preprocessed data

#### Outputs:

- **Learner**  
A learning algorithm with supplied parameters
- **Predictor**  
A trained regressor. Signal *Predictor* sends the output signal only if input *Data* is present.

### Description

The **Nearest Neighbors** widget uses the [kNN algorithm](#) that searches for k closest training examples in feature space and uses their average as prediction.



1. Learner/predictor name
2. Set the number of nearest neighbors and the distance parameter (metric) as regression criteria. Metric can be:

- **Euclidean** (“straight line”, distance between two points)
  - **Manhattan** (sum of absolute differences of all attributes)
  - **Maximal** (greatest of absolute differences between attributes)
  - **Mahalanobis** (distance between point and distribution).
- You can assign weight to the contributions of the neighbors. The *Weights* you can use are:
    - **Uniform**: all points in each neighborhood are weighted equally.
    - **Distance**: closer neighbors of a query point have a greater influence than the neighbors further away.
  - Produce a report.
  - Press *Apply* to commit changes.

## Example

Below, is a workflow showing how to use both the *Predictor* and the *Learner* output. For the purpose of this example, we used the *Housing* data set. For the *Predictor*, we input the prediction model into the *Predictions* widget and view the results in the *Data Table*. For *Learner*, we can compare different learners in the *Test & Score* widget.

The screenshot shows an Orange3 workflow and the outputs of the **Nearest Neighbors** widget. The workflow consists of a **File** widget connected to a **Nearest Neighbors** widget, which is then connected to **Predictions**, **Data Table**, **Mean Learner**, and **Test & Score** widgets.

The **Nearest Neighbors** widget settings are shown in the bottom-left panel:

- Name: Nearest Neighbors
- Neighbors: 5
- Metric: Euclidean
- Weight: Uniform

The **Data Table** widget shows the following data:

	MEDV	Nearest Neighbors	CRIM	ZN
1	24.000	21.780	0.006	18.00
2	21.600	22.900	0.027	0.00
3	34.700	25.360	0.027	0.00
4	33.400	26.060	0.032	0.00
5	36.200	27.100	0.069	0.00
6	28.700	27.100	0.030	0.00
7	22.900	20.880	0.088	12.50
8	27.100	19.100	0.145	12.50
9	16.500	18.400	0.211	12.50
10	18.900	19.480	0.170	12.50
11	15.000	19.280	0.225	12.50
12	18.900	22.000	0.117	12.50
13	21.700	24.340	0.094	12.50
14	20.400	20.520	0.630	0.00

The **Test & Score** widget shows the following evaluation results:

Method	MSE	RMSE	MAE	R2
Mean Learner	84.420	9.188	6.647	0.000
Nearest Neighbors	23.967	4.896	3.375	0.716

## Stochastic Gradient Descent



Learns a linear classifier by minimizing a chosen loss function.

## Signals

### Inputs:

- **Data**  
A data set.
- **Preprocessor**  
Preprocessed data.

### Outputs:

- **Learner**  
A learning algorithm with supplied parameters
- **Predictor**  
A trained regressor. Signal *Predictor* sends the output signal only if input *Data* is present.

## Description

The **Stochastic Gradient Descent** widget uses [stochastic gradient descent](#) that minimizes a chosen loss function with a linear function. The algorithm approximates a true gradient by considering one sample at a time, and simultaneously updates the model based on the gradient of the loss function. This regression returns predictors as minimizers of the sum, i.e. M-estimators, and is especially useful for large-scale and sparse data sets.

1. Learner/predictor name
2. Loss functions:
  - [Squared Loss](#) (fitted to ordinary least-squares)
  - [Huber](#) (switches to linear loss beyond  $\epsilon$ )
  - [Epsilon insensitive](#) (ignores errors within  $\epsilon$ , linear beyond it)
  - *Squared epsilon insensitive* (loss is squared beyond  $\epsilon$ -region).
3. Regularization norms to prevent overfitting:
  - [Absolute norm](#) (L1, leading to sparse solutions)
  - [Euclidean norm](#) (L2, standard regularizer)
  - [Elastic net](#) (mixing both penalty norms).
4. Use the default [Inverse scaling](#) for the learning rate to be inversely related to the number of iterations or select *Constant* for LR to stay the same through all epochs (passes).
5. Set the constants for the [regression algorithm](#):
  - **Eta0**: initial learning rate
  - **Power t**: exponent for inverse scaling learning rate; t is time step 6, which relates to the number of passes through the training data.
7. Produce a report.
8. Press *Apply* to commit changes. Alternatively, tick the box on the left side of the *Apply* button and changes will be communicated automatically.

**Stochastic Gradient D...** ? ✕

**Name 1**

SGD Regression

**Loss Function 2**

☒ Squared loss

☐ Huber

☐ Epsilon insensitive

☐ Squared epsilon insensitive

$\epsilon$ : 0,10

**Penalty 3**

☐ Absolute norm (L1)

☒ Euclidean norm (L2)

☐ Elastic net (L1 and L2)

$\alpha$ : 0,0001

L1 ratio: 0,15

**Learning Rate 4**

☐ Constant

☒ Inverse scaling

**5**  $\eta_0$ : 0,01

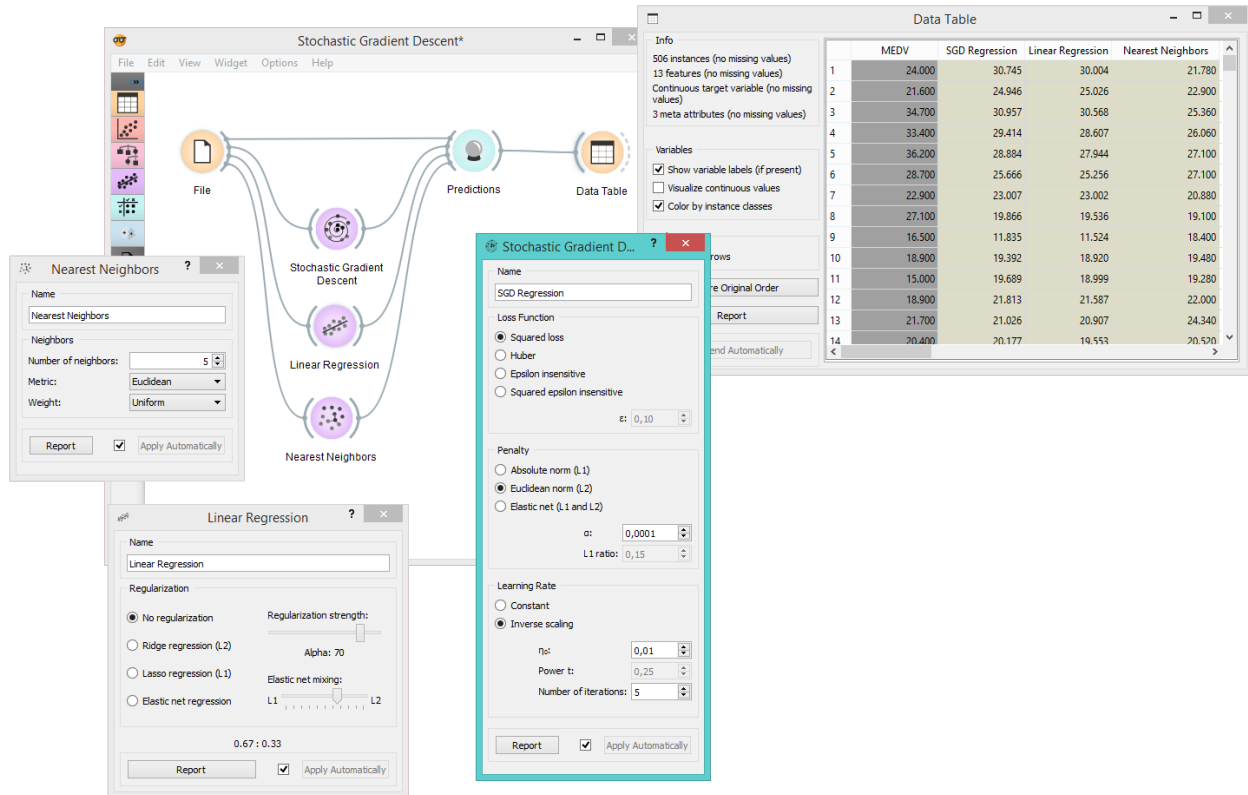
Power t: 0,25

**6** Number of iterations: 5

**7** Report **8** ☒ Apply Automatically

## Example

Below, we compare three different regression models to see which predict what kind of results. For the purpose of this example, the *Housing* data set is used. We connect the *File* widget to *Stochastic Gradient Descent*, *Linear Regression* and *Nearest Neighbors* widget and all four to the *Predictions* widget. Then we use the *Data Table* to see what predictions each model makes.



## SVM Regression



Learns a SVM regression of its input data.

### Signals

#### Inputs:

- **Data**  
A data set.
- **Preprocessor**  
Preprocessed data.

**Outputs:**

- **Learner**

A SVM learning algorithm with supplied parameters.

- **Predictor**

A trained regressor. Signal *Predictor* sends the regressor only if signal *Data* is present.

**Description**

**SVM Regression** performs linear regression in a high dimension feature space using an  $\epsilon$ -intensive loss. Its estimation accuracy depends on a good setting of  $C$ ,  $\epsilon$  and kernel parameters. The widget outputs class predictions based on a [SVM learning algorithm](#).

1. Learner/predictor name
2. Train an  $\epsilon$ -SVR or  $\nu$ -SVR model and set test error bounds.
3. Set kernel, a function that transforms attribute space to a new feature space to fit the maximum-margin hyper-plane, thus allowing the algorithm to create non-linear regressors. The first kernel in the list, however, is a [Linear](#) kernel that does not require this trick, but all the others ([Polynomial](#), [RBF](#) and [Sigmoid](#)) do. Functions that specify the kernel are presented beside their names, and the constants involved are:
  - **g** for the gamma constant in kernel function (the recommended value is  $1/k$ , where  $k$  is the number of the attributes, but since there may be no training set given to the widget the default is 0 and the user has to set this option manually),
  - **c** for the constant  $c_0$  in the kernel function (default 0), and
  - **d** for the degree of the kernel (default 3).
4. Set permitted deviation from the expected value.
5. Produce a report.
6. Press *Apply* to commit changes. Alternatively, tick the box on the left side of the *Apply* button to communicate changes automatically.

**Example**

Below, we use two regression models to compare predictions in the [Predictions](#) widget. We used the *Housing* data set.

**Regression Tree**

Regression Tree

**Signals****Inputs:**




- **Data**

A data set

- **Preprocessor**

Preprocessed data.



 SVM Regression  

**Name 1**

**SVR Type 2**  

☒  $\epsilon$ -SVR  
Cost (C):   
Loss epsilon ( $\epsilon$ ):

☐  $\nu$ -SVR  
Cost (C):   
Complexity bound ( $\nu$ ):

**Kernel 3**  

☐ Linear  
Kernel:  $\tanh(g \cdot x \cdot y + c)$

☒ Polynomial  
g:   
c:   
d:

☐ RBF

☐ Sigmoid

**Optimization Parameters 4**  
Numerical tolerance:

**5**  **6** ☒

The screenshot shows the Orange3 visual programming interface. A workflow is built with the following widgets: **File** (data source), **SVM Regression** (model), **Mean Learner** (model), and **Predictions** (output). The **Mean Learner** widget configuration window is open, showing its name and a checkbox for "Apply Automatically". The **SVM Regression** widget configuration window is also open, showing settings for SVR Type (ε-SVR), Cost (C), Loss epsilon (ε), Complexity bound (ν), Kernel (Polynomial), and Optimization Parameters (Numerical tolerance). The **Predictions** widget displays a table of results for 11 instances, comparing the predictions of the SVM Regression and Mean Learner models against the actual values of the target variable (MEDV).

	SVM Regression	Mean Learner	MEDV	CRIM	ZN	INDUS	CHAS	
1	24.865	22.533	24.000	0.006	18.000	2.310	0.000	0.538
2	22.011	22.533	21.600	0.027	0.000	7.070	0.000	0.469
3	23.349	22.533	34.700	0.027	0.000	7.070	0.000	0.469
4	25.092	22.533	33.400	0.032	0.000	2.180	0.000	0.458
5	24.352	22.533	36.200	0.069	0.000	2.180	0.000	0.458
6	23.442	22.533	28.700	0.030	0.000	2.180	0.000	0.458
7	21.459	22.533	22.900	0.088	12.500	7.870	0.000	0.524
8	20.471	22.533	27.100	0.145	12.500	7.870	0.000	0.524
9	18.098	22.533	16.500	0.211	12.500	7.870	0.000	0.524
10	20.422	22.533	18.900	0.170	12.500	7.870	0.000	0.524
11	20.126	22.533	15.000	0.225	12.500	7.870	0.000	0.524

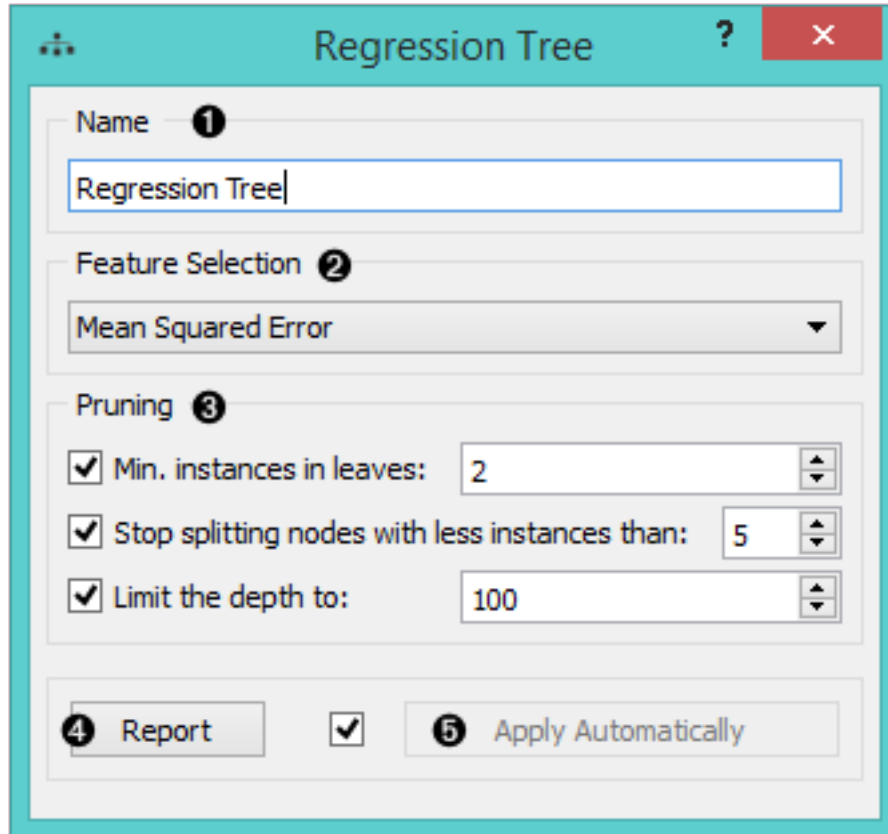
**Outputs:**

- **Learner**

A regression tree learning algorithm with settings as specified in the dialog.

- **Predictor**

Trained regressor.

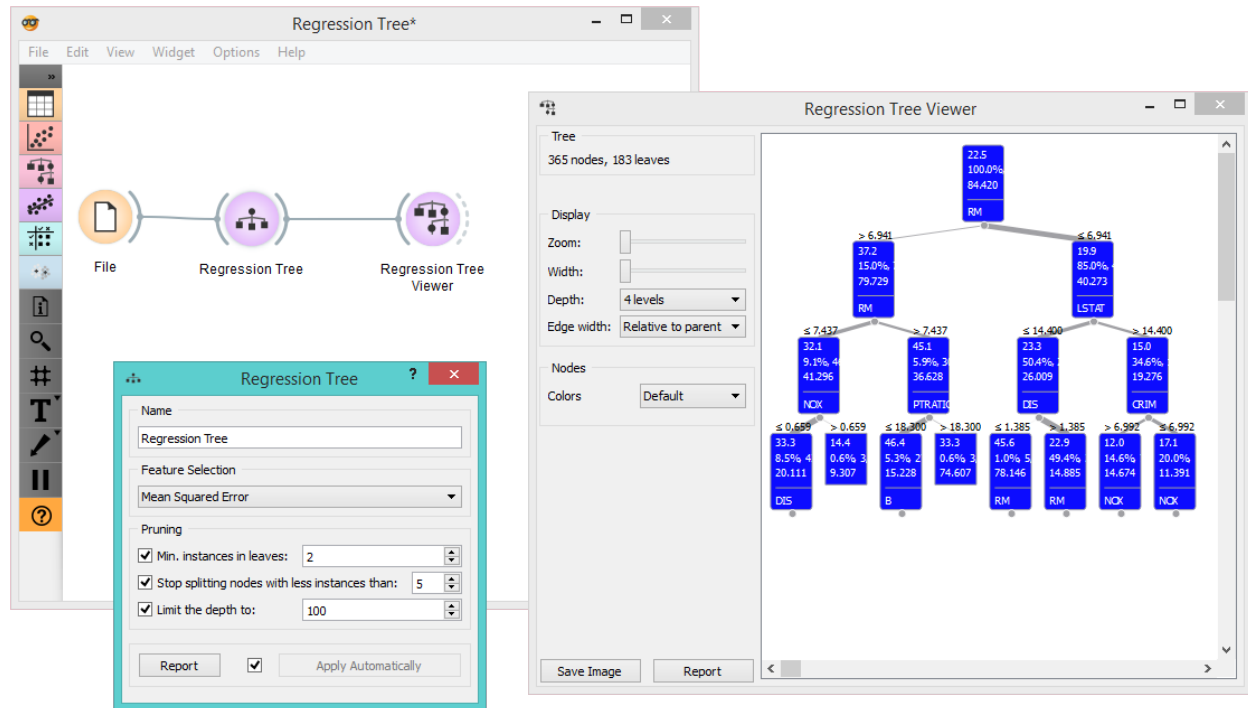
**Description**

1. The learner can be given a name under which it will appear in other widgets. The default name is “Regression Tree”.
2. In *Feature selection*, there is just one option, namely **Mean Squared Error**, which measures the average of the squares of the errors or deviations (the difference between the estimator and what is estimated).
3. *Pruning* criteria:
  - **Minimal instances in leaves**; if checked, the algorithm will never construct a split which would put less than the specified number of training examples into any of the branches.
  - **Stop splitting nodes with less instances than** forbids the algorithm to split the nodes with less than the given number of instances.
  - **Limit the depth** of the regression tree.
3. Produce a report.

- After changing the settings, you need to click *Apply*, which will put the new learner in the output and, if the training examples are given, construct a new regressor and output it as well.

## Examples

There are two typical uses for this widget. First, you may want to induce a model and check what it looks like. You do it with the workflow below. To learn more about it, see the documentation on [Regression Tree Viewer](#).



The second schema checks the accuracy of the algorithm. The selected tree node is presented in the [Scatter Plot](#) and we can see that the selected examples exhibit the same features.

## Regression Tree Viewer

Visualization of a regression tree.

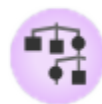
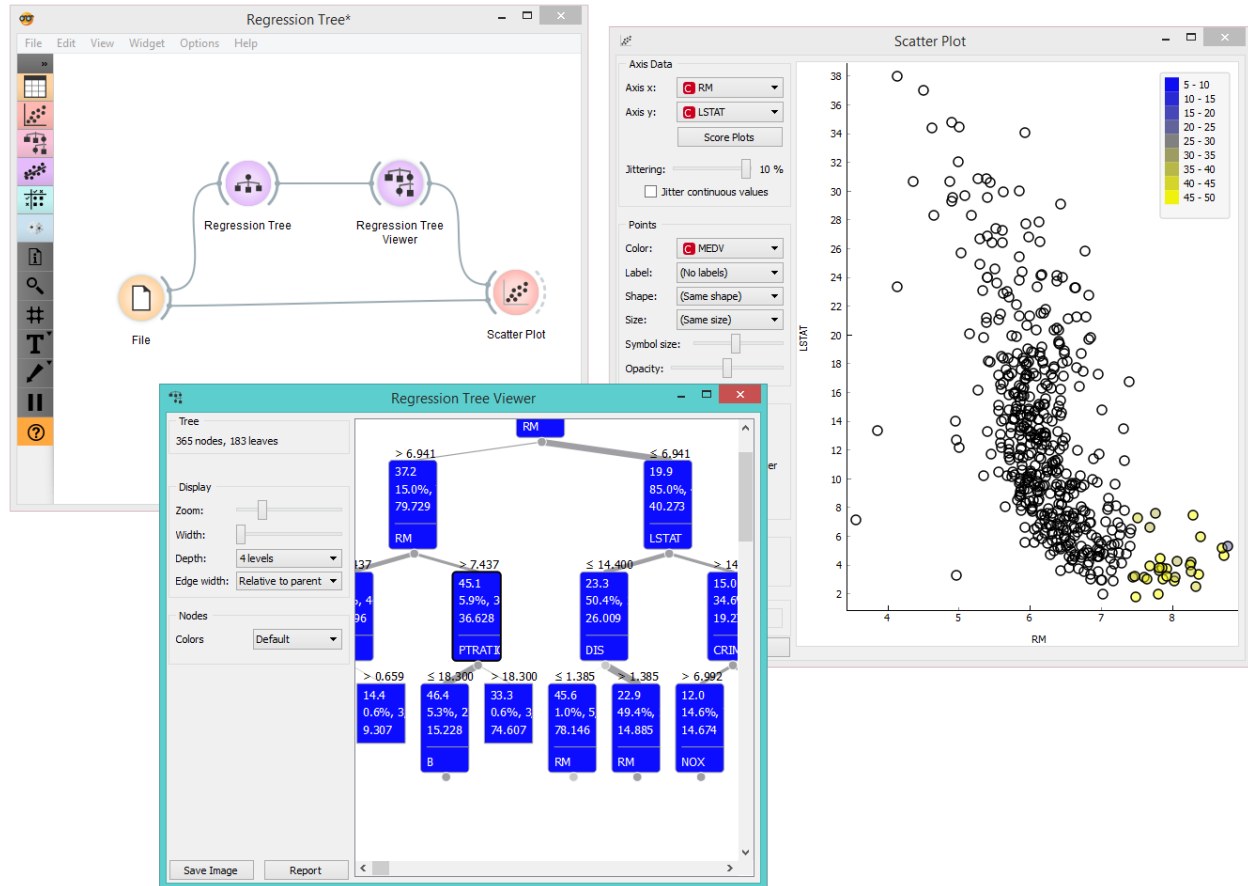
### Signals

#### Inputs:

- **Regression Tree**  
Regression tree

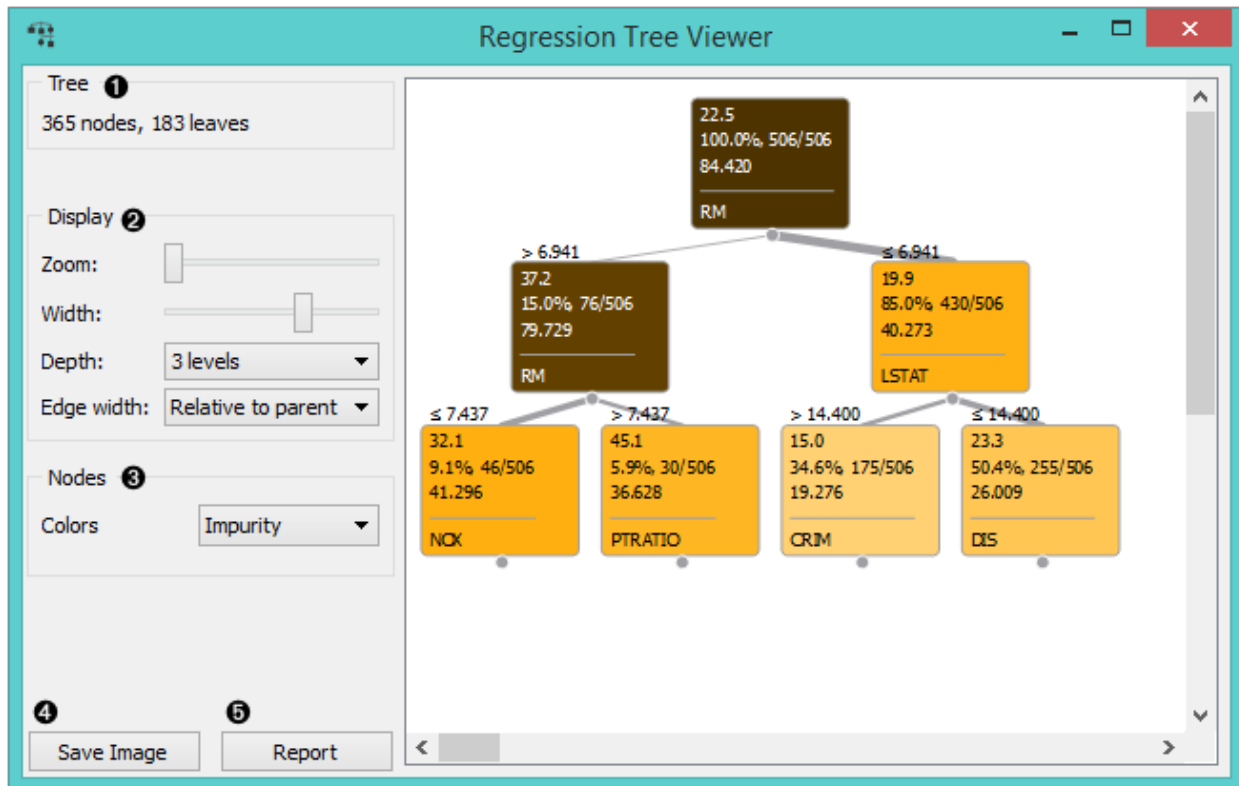
#### Outputs:

- **Data**  
Data from a selected tree node



## Description

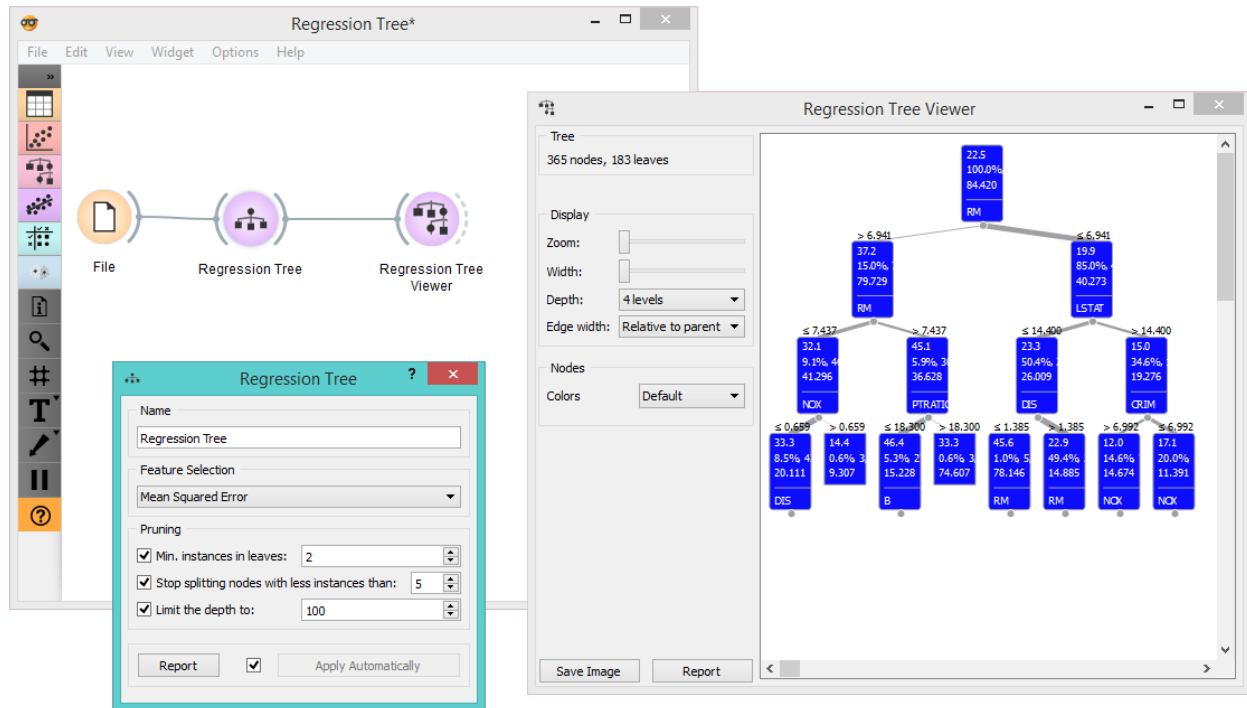
This is a versatile widget with 2-D visualization of a [regression tree](#). The user can select a node, instructing the widget to output the data associated with the node, thus enabling explorative data analysis.



1. Information on the input.
2. Set the zoom and define the tree width. The nodes display tooltips when hovering over them.
3. The edges between the nodes in the tree graph are drawn based on the selected edge width.
  - All the edges will be of equal width if *Fixed* is chosen.
  - When *Relative to root* is selected, the width of the edge will correspond to the proportion of instances in the corresponding node with respect to all the instances in the training data. Under this selection, the edge will get thinner and thinner when traversing toward the bottom of the tree.
  - *Relative to parent* makes the edge width correspond to the proportion of instances in the nodes with respect to the instances in their parent node.
4. The nodes box defines the target class, which you can change based on the classes in the data. You can also use *Set Colors* and color the tree according to the number of instances in a node or impurity. You can also choose to keep the default coloring.
5. Press *Save Graph* to save the regression tree graph as a file to your computer in a .svg or .png format.
6. Regression tree.

## Examples

Below, is a simple schema, where we have read the data, constructed the regression tree and viewed it in our tree viewer. We loaded the *Housing* data set and limited the depth of the tree to only 4 levels because of the vastness of the data set. It is worth remembering that if both the viewer and *Regression Tree* are open, any run of the tree induction algorithm will immediately affect the visualization. You can thus use this combination to explore how parameters of the induction algorithm influence the structure of the resulting tree.



Clicking on any node will output the related data instances. This is explored in the *Scatterplot*. Make sure that the tree data is passed as a data subset; this can be done by connecting the *Scatterplot* to the *File* widget first, and connecting it to the **Tree Viewer** widget next.

## Random Forest Regression

Random forest regression

### Signals

Inputs:

- **Data**

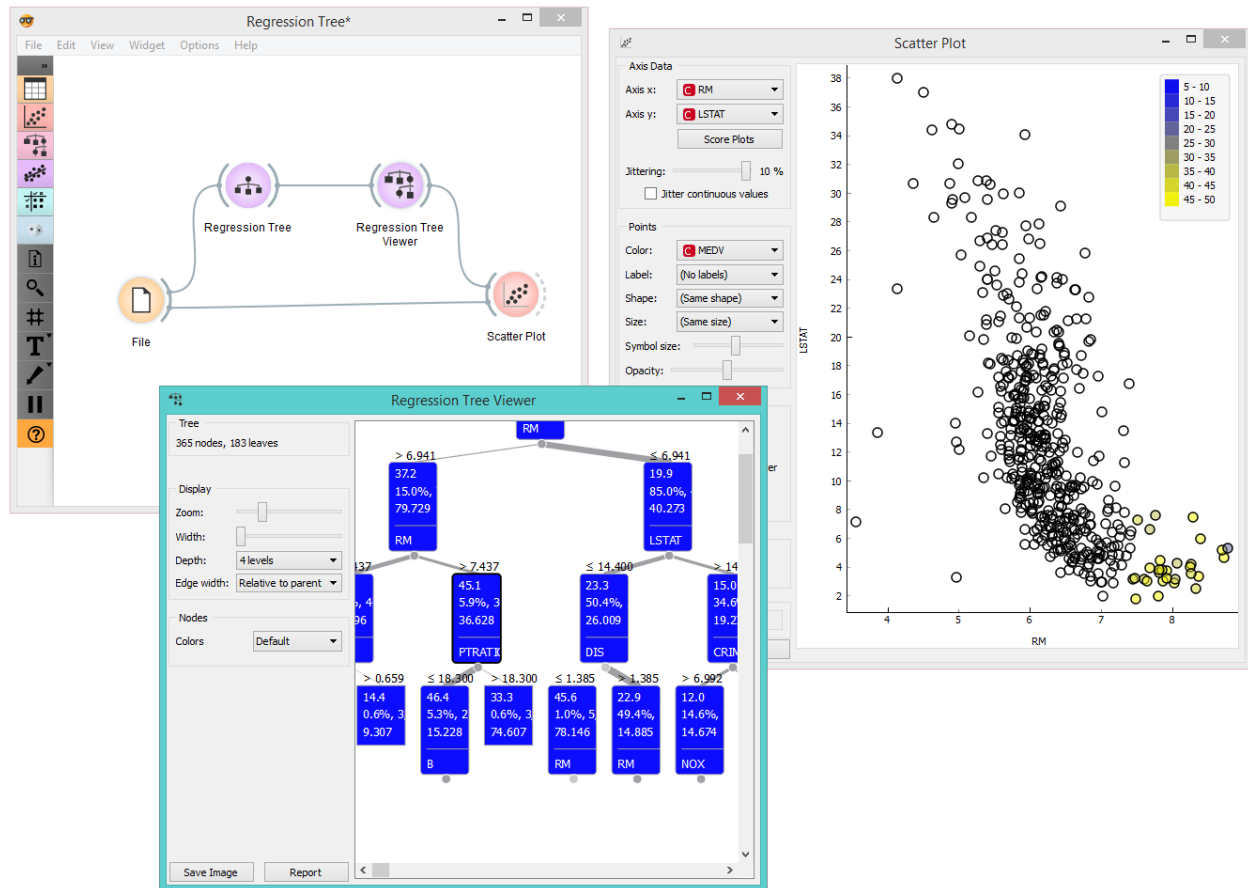
Data set

- **Preprocessor**

Preprocessed data.

Outputs:

- **Learner**





Random forest learning algorithm with settings as specified in the dialog.

- **Predictor**

Trained regressor.

## Description

**Random forest** is an ensemble learning method used for classification, regression and other tasks. It was first proposed by Tin Kam Ho and further developed by Leo Breiman and Adele Cutler. When given a data set, **Random Forest** builds a set of regression trees. Each tree is developed from a bootstrap sample from the training data. When developing individual trees, an arbitrary subset of attributes is drawn (hence the term “Random”) from which the best attribute for the split is selected.

1. Specify the name of the learner or predictor. The default name is *Random Forest Regression*.
2. Specify how many regression trees will be included in the forest (*Number of trees in the forest*), and how many attributes will be arbitrarily drawn for consideration at each node. If the latter is not specified (option *Number of attributes...* left unchecked), this number is equal to the square root of the number of attributes in the data. You can also choose to control the random number generator (*Fixed seed for random generator*).
3. Original Brieman’s proposal is to grow the trees without any pre-pruning, but since pre-pruning often works quite well and is faster, the user can set the depth to which the trees will be grown (*Limit depth of individual trees*). Another pre-pruning option is stop splitting nodes when the maximal depth is reached (*Do not split subsets smaller than*).

4. Produce a report.
5. Click *Apply* to communicate the changes to other widgets. Alternatively, tick *Apply Automatically* and changes will be communicated automatically.

## Example

The example below compares different learners, namely **Random Forest Regression**, *Linear Regression* and *Mean Learner*, in the *Test&Score* widget.

The screenshot displays the Orange Visual Programming interface. A workflow is shown on the left, connecting a 'File' widget to three learner widgets: 'Random Forest Regression', 'Linear Regression', and 'Mean Learner'. These learners are connected to a 'Test & Score' widget. The 'Test & Score' widget's settings are shown in a separate window, displaying the 'Evaluation Results' table below.

Method	MSE	RMSE	MAE	R2
Random Forest Regression	12.936	3.597	2.351	0.847
Linear Regression	23.370	4.834	3.376	0.723
Mean Learner	84.644	9.200	6.662	-0.003

Below the workflow, the settings for the 'Random Forest Regression' and 'Linear Regression' widgets are shown. The 'Random Forest Regression' widget settings include:

- Name: Random Forest Regression
- Basic Properties:
  - Number of trees: 10
  - Number of attributes considered at each split: 5
  - Fixed seed for random generator: 0
- Growth Control:
  - Limit depth of individual trees: 3
  - Do not split subsets smaller than: 5
- Buttons: Report, Apply Automatically (checked)

The 'Linear Regression' widget settings include:

- Name: Linear Regression
- Regularization:
  - No regularization (selected)
  - Ridge regression (L2)
  - Lasso regression (L1)
  - Elastic net regression
- Regularization strength: Alpha: 0.0001
- Elastic net mixing: L1 : L2 (0.50 : 0.50)
- Buttons: Report, Apply Automatically (checked)

In order to demonstrate how to use the *Predictor* output, we used the *Housing* data set and connected the *Random Forest Regression* widget with the *Predictions* widget. The results are displayed in the appended data table.

## Unsupervised

### PCA

PCA linear transformation of input data.

### Signals

Inputs:

The screenshot shows the Orange Visual Programming interface. A workflow is built with three widgets: 'File', 'Random Forest Regression', and 'Predictions'. The 'Random Forest Regression' widget is highlighted, and its settings window is open. The 'Predictions' window displays a table of results.

**Random Forest Regression Settings:**

- Name: Random Forest Regression
- Basic Properties:
  - Number of trees: 10
  - ☐ Number of attributes considered at each split: 5
  - ☐ Fixed seed for random generator: 0
- Growth Control:
  - ☐ Limit depth of individual trees: 3
  - ☒ Do not split subsets smaller than: 5
- Buttons: Report, ☒ Apply Automatically

**Predictions Table:**

	Random Forest Regression	MEDV	CRIM	ZN
1	24.788	24.000	0.006	18.000
2	22.458	21.600	0.027	0.000
3	35.400	34.700	0.027	0.000
4	33.873	33.400	0.032	0.000
5	34.691	36.200	0.069	0.000
6	27.160	28.700	0.030	0.000
7	20.968	22.900	0.088	12.500
8	22.892	27.100	0.145	12.500
9	17.584	16.500	0.211	12.500
10	19.887	18.900	0.170	12.500
11	16.328	15.000	0.225	12.500



- **Data**

A data set.

**Outputs:**

- **Transformed Data**

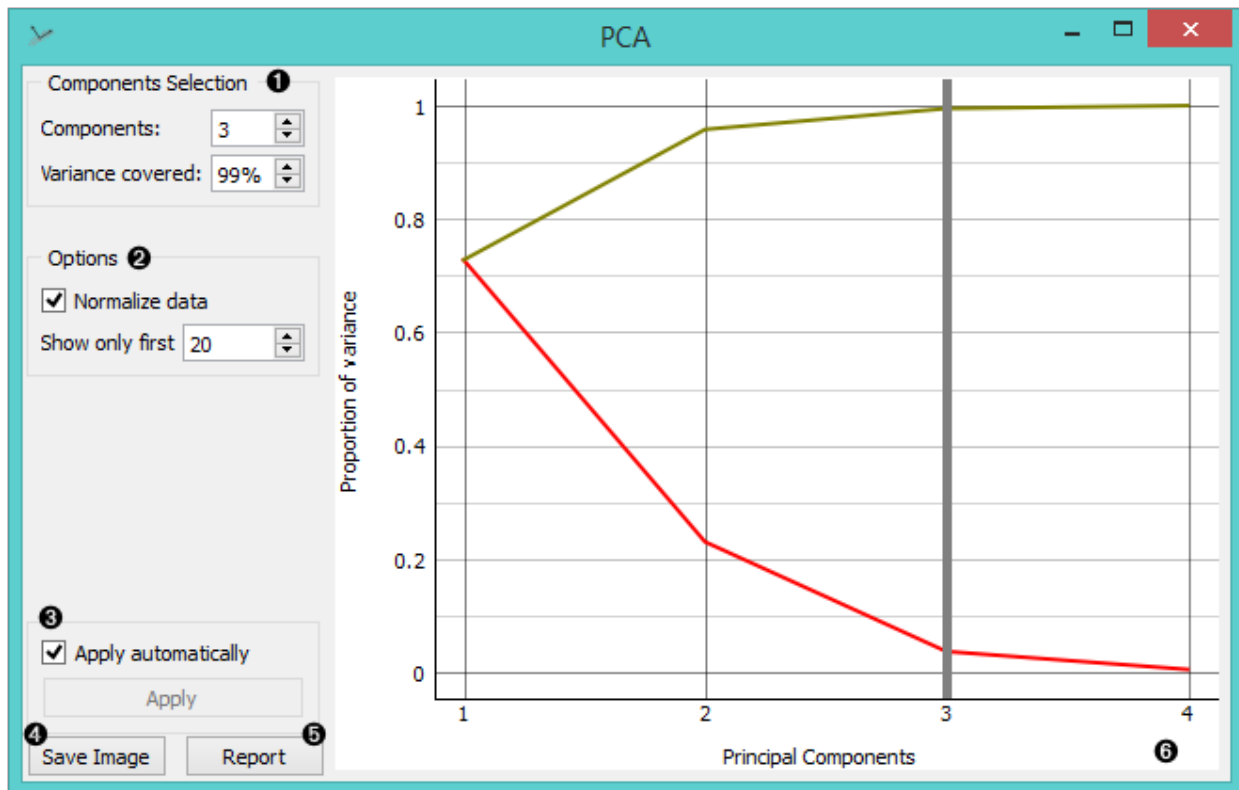
PCA transformed input data.

- **Components**

Eigenvectors.

**Description**

Principal Component Analysis (PCA) computes the PCA linear transformation of the input data. It outputs either a transformed data set with weights of individual instances or weights of principal components.



1. Select how many principal components you wish in your output. It is best to choose as few as possible with variance covered as high as possible. You can also set how much variance you wish to cover with your principal components.
2. You can normalize data to adjust the values to common scale.
3. When *Apply Automatically* is ticked, the widget will automatically communicate all changes. Alternatively, click *Apply*.
4. Press *Save Image* if you want to save the created image to your computer.
5. Produce a report.

- Principal components graph, where the red (lower) line is the variance covered per component and the green (upper) line is cumulative variance covered by components.

The number of components of the transformation can be selected either in the *Components Selection* input box or by dragging the vertical cutoff line in the graph.

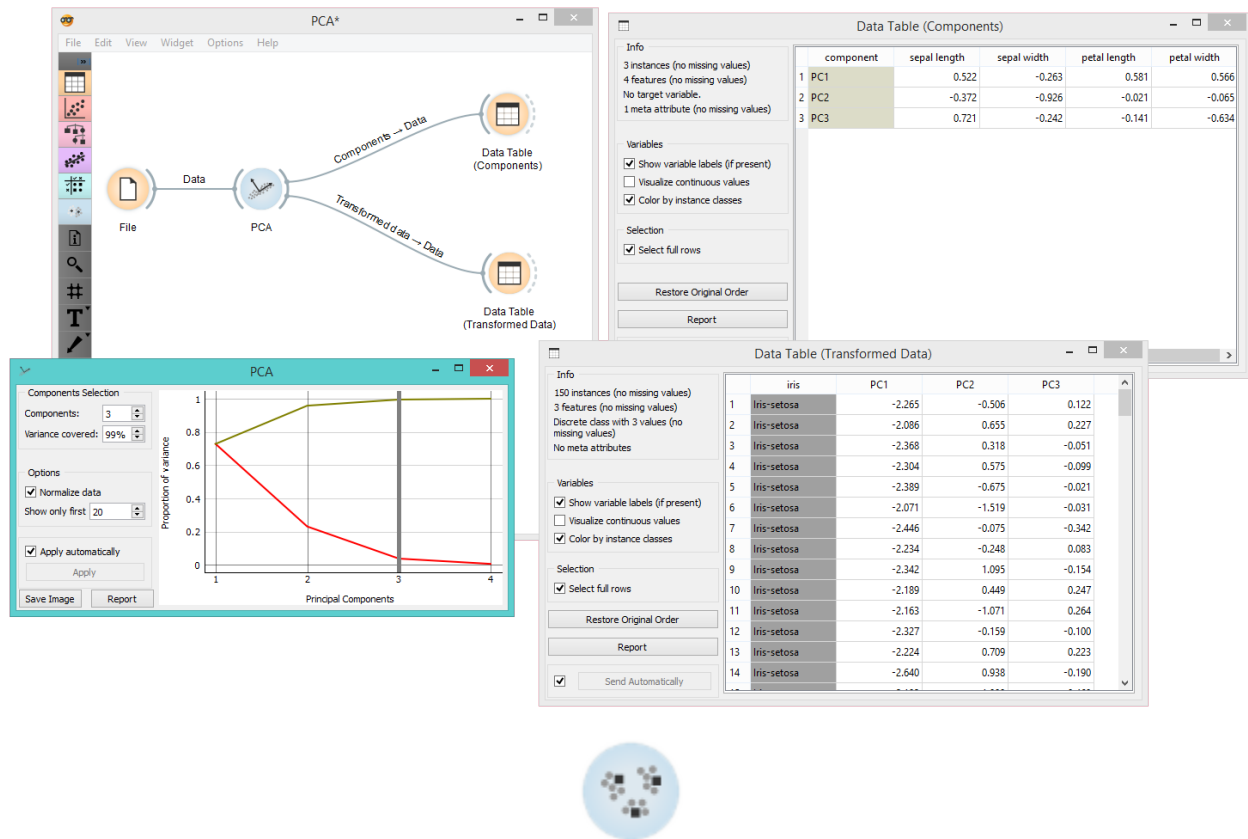
## Examples

PCA can be used to simplify visualizations of large data sets. Below, we used the *Iris* data set to show how we can improve the visualization of the data set with PCA. The transformed data in the *Scatter Plot* show a much clearer distinction between classes than the default settings.



The widget provides two outputs: transformed data and principal components. Transformed data are weights for individual instances in the new coordinate system, while components are the system descriptors (weights for principal components). When fed into the *Data Table*, we can see both outputs in numerical form. We used two data tables in order to provide a more clean visualization of the workflow, but you can also choose to edit the links in such a way that you display the data in just one data table. You only need to create two links and connect the *Transformed data* and *Components* inputs to the *Data* output.

## Correspondence Analysis



## Signals

### Inputs:

- **Data**  
A data set.

### Outputs:

- None

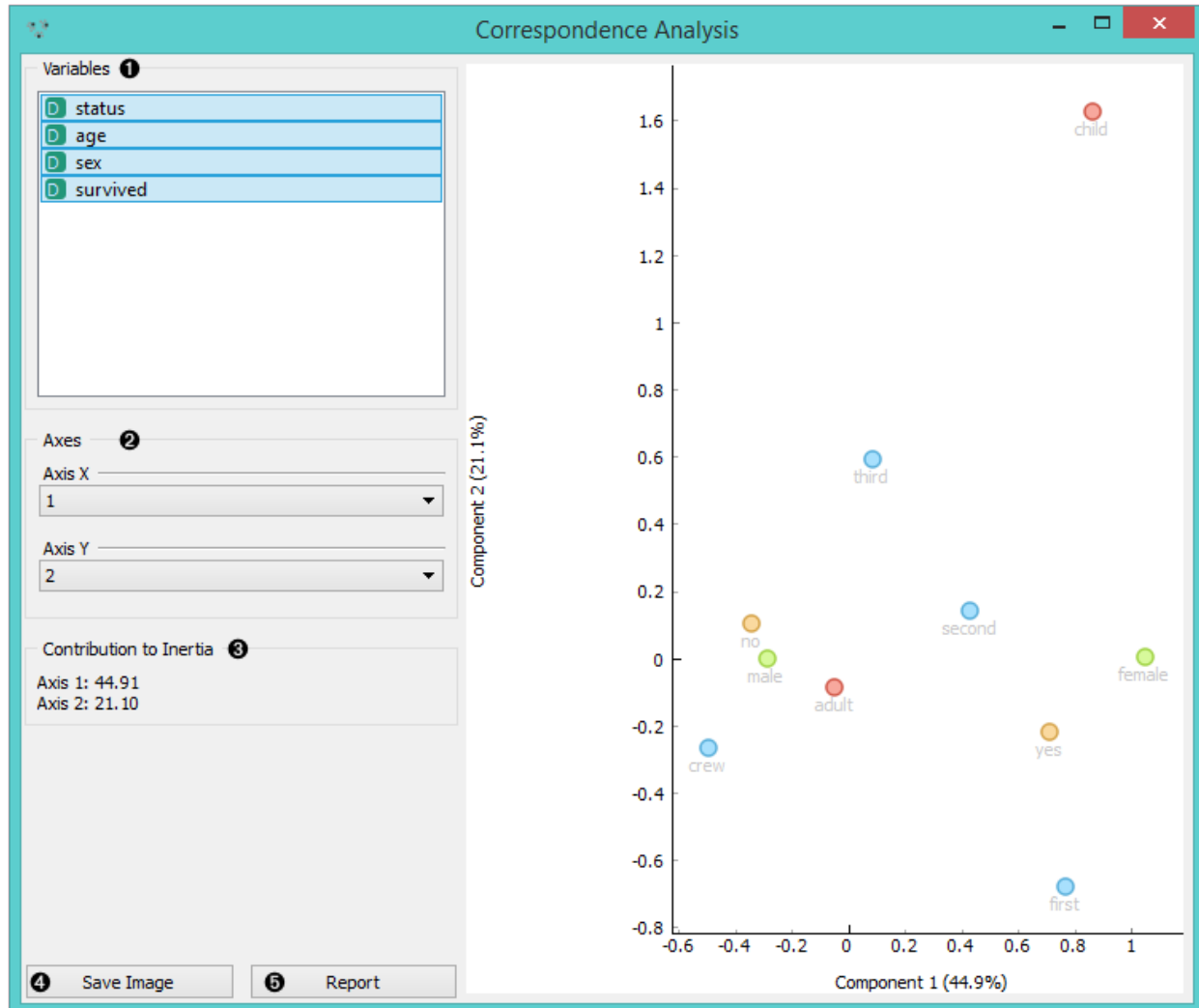
## Description

**Correspondence Analysis (CA)** computes the CA linear transformation of the input data. While it is similar to PCA, CA computes linear transformation on discrete rather than on continuous data.

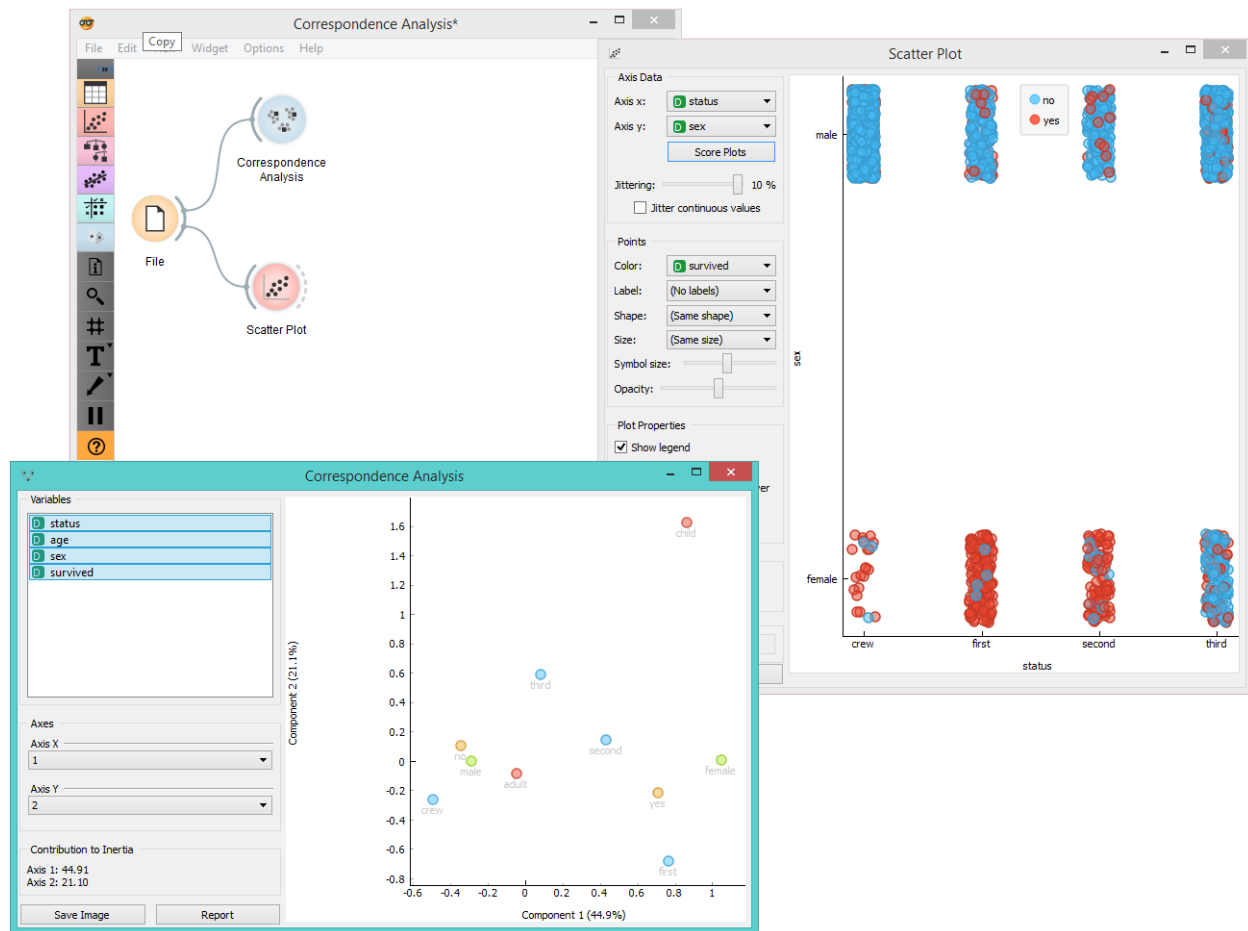
1. Select the variables you want to see plotted.
2. Select the component for each axis.
3. **Inertia** values (percentage of independence from transformation, i.e. variables are in the same dimension).
4. Produce a report.

## Example

Below, is a simple comparison between the **Correspondence Analysis** and *Scatter plot* widgets on the *Titanic* data set. While the *Scatter plot* shows fairly well which class and sex had a good survival rate and which one didn't,



**Correspondence Analysis** can plot several variables in a 2-D graph, thus making it easy to see the relations between variable values. It is clear from the graph that “no”, “male” and “crew” are related to each other. The same goes for “yes”, “female” and “first”.



## Distance Map



Visualizes distances between items.

## Signals

### Inputs:

- **Distances**

A distance matrix.

### Outputs:



- **Data**

Instances corresponding to the selected elements of the matrix.

- **Features**

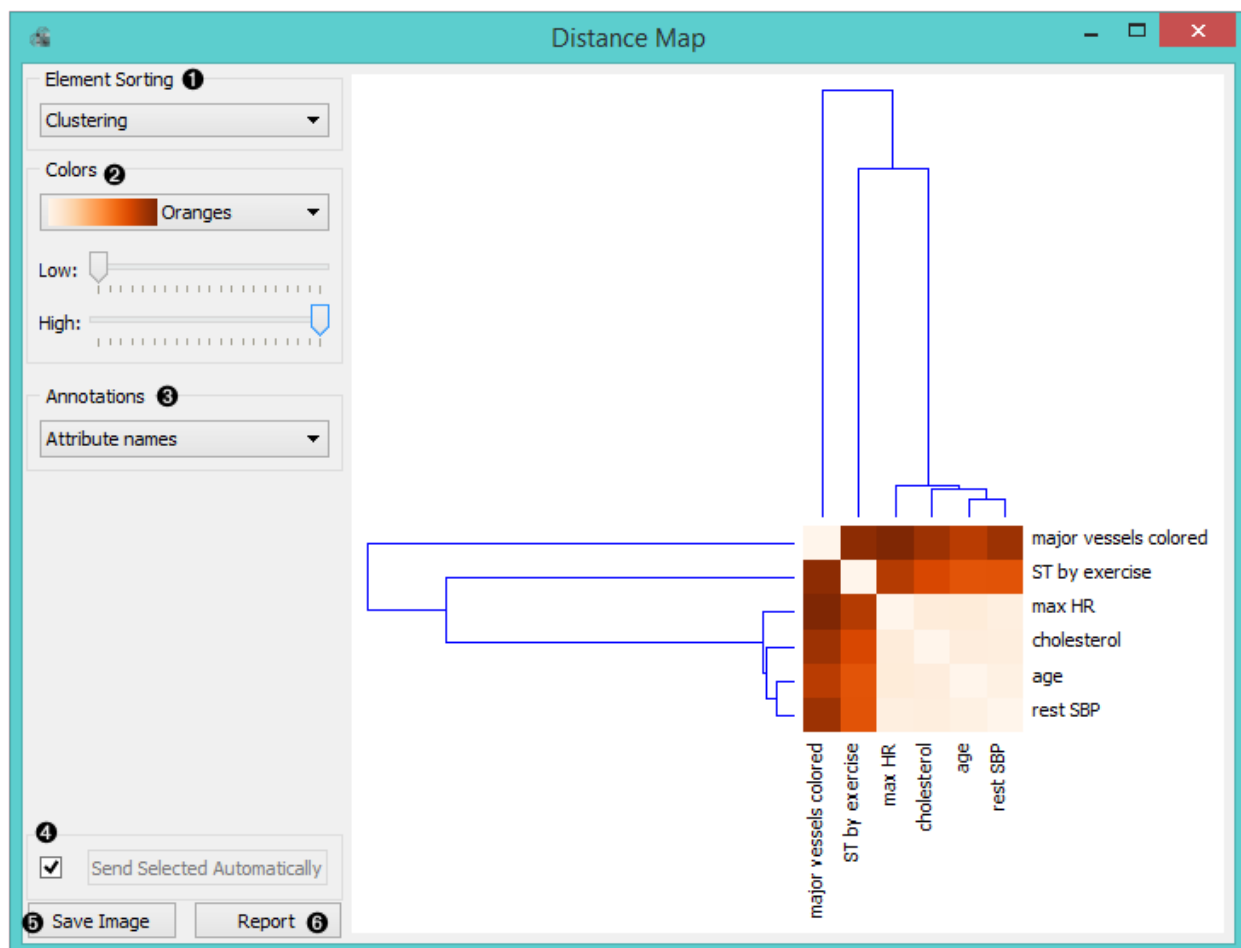
Attributes corresponding to the selected elements of the matrix.

## Description

The **Distance Map** visualizes distances between objects. The visualization is the same as if we printed out a table of numbers, except that the numbers are replaced by colored spots.

Distances are most often those between instances (“rows” in the *Distances* widget) or attributes (“columns” in *Distances* widget). The only suitable input for **Distance Map** is the *Distances* widget. For the output, the user can select a region of the map and the widget will output the corresponding instances or attributes. Also note that the **Distances** widget ignores discrete values and calculates distances only for continuous data, thus it can only display distance map for discrete data if you *Continue* them first.

The snapshot shows distances between columns in the *heart disease* data, where smaller distances are represented with light and larger with dark orange. The matrix is symmetric and the diagonal is a light shade of orange - no attribute is different from itself. Symmetricity is always assumed, while the diagonal may also be non-zero.



1. *Element sorting* arranges elements in the map by

- None (lists instances as found in the data set)
- **Clustering** (clusters data by similarity)
- **Clustering with ordered leaves** (maximizes the sum of similarities of adjacent elements)

## 2. Colors

- **Colors** (select the color palette for your distance map)
- **Low** and **High** are thresholds for the color palette (low for instances or attributes with low distances and high for instances or attributes with high distances).

## 3. Select Annotations.

4. If *Send Selected Automatically* is on, the data subset is communicated automatically, otherwise you need to press *Send Selected*.

5. Press *Save Image* if you want to save the created image to your computer.

6. Produce a report.

Normally, a color palette is used to visualize the entire range of distances appearing in the matrix. This can be changed by setting the low and high threshold. In this way we ignore the differences in distances outside this interval and visualize the interesting part of the distribution.

Below, we visualized the most correlated attributes (distances by columns) in the *heart disease* data set by setting the color threshold for high distances to the minimum. We get a predominantly black square, where attributes with the lowest distance scores are represented by a lighter shade of the selected color schema (in our case: orange). Beside the diagonal line, we see that in our example *ST by exercise* and *major vessels colored* are the two attributes closest together.

The user can select a region in the map with the usual click-and-drag of the cursor. When a part of the map is selected, the widget outputs all items from the selected cells.

## Examples

The first workflow shows a very standard use of the **Distance Map** widget. We select 70% of the original *Iris* data as our sample and view the distances between rows in **Distance Map**.

In the second example, we use the *heart disease* data again and select a subset of women only from the *Scatter Plot*. Then, we visualize distances between columns in the **Distance Map**. Since the subset also contains some discrete data, the *Distances* widget warns us it will ignore the discrete features, thus we will see only continuous instances/attributes in the map.

## Distances

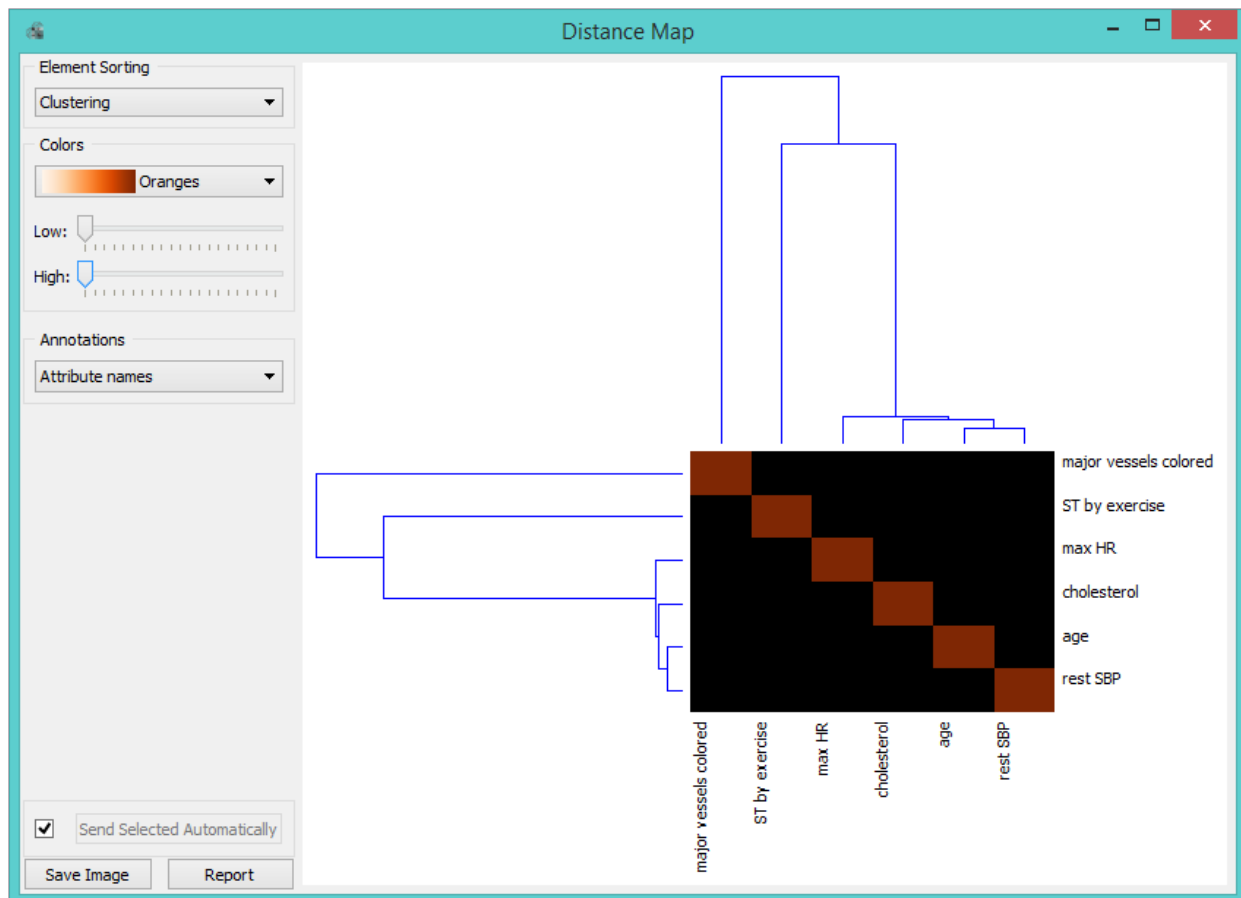
Computes distances between rows/columns in a data set.

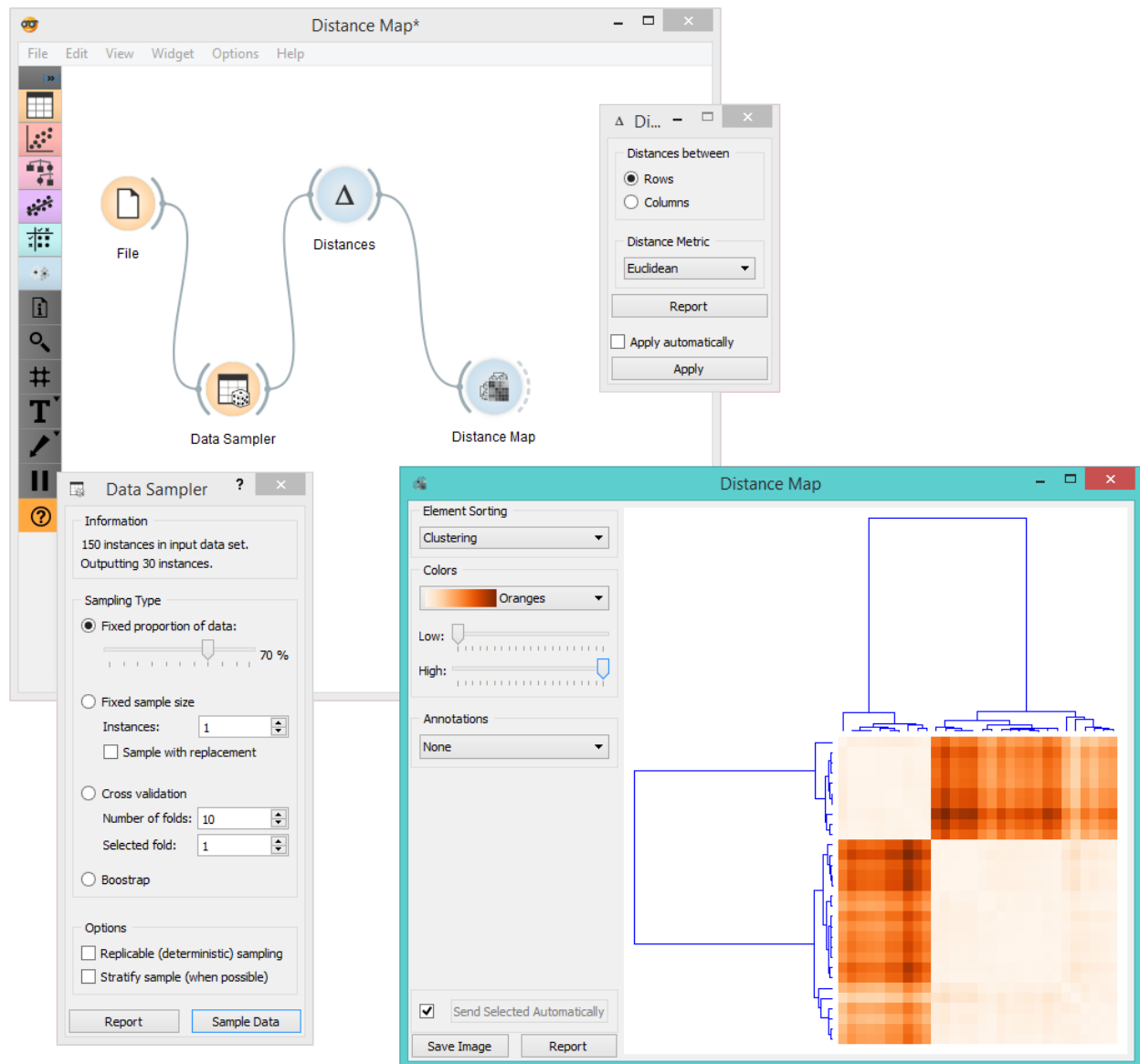
## Signals

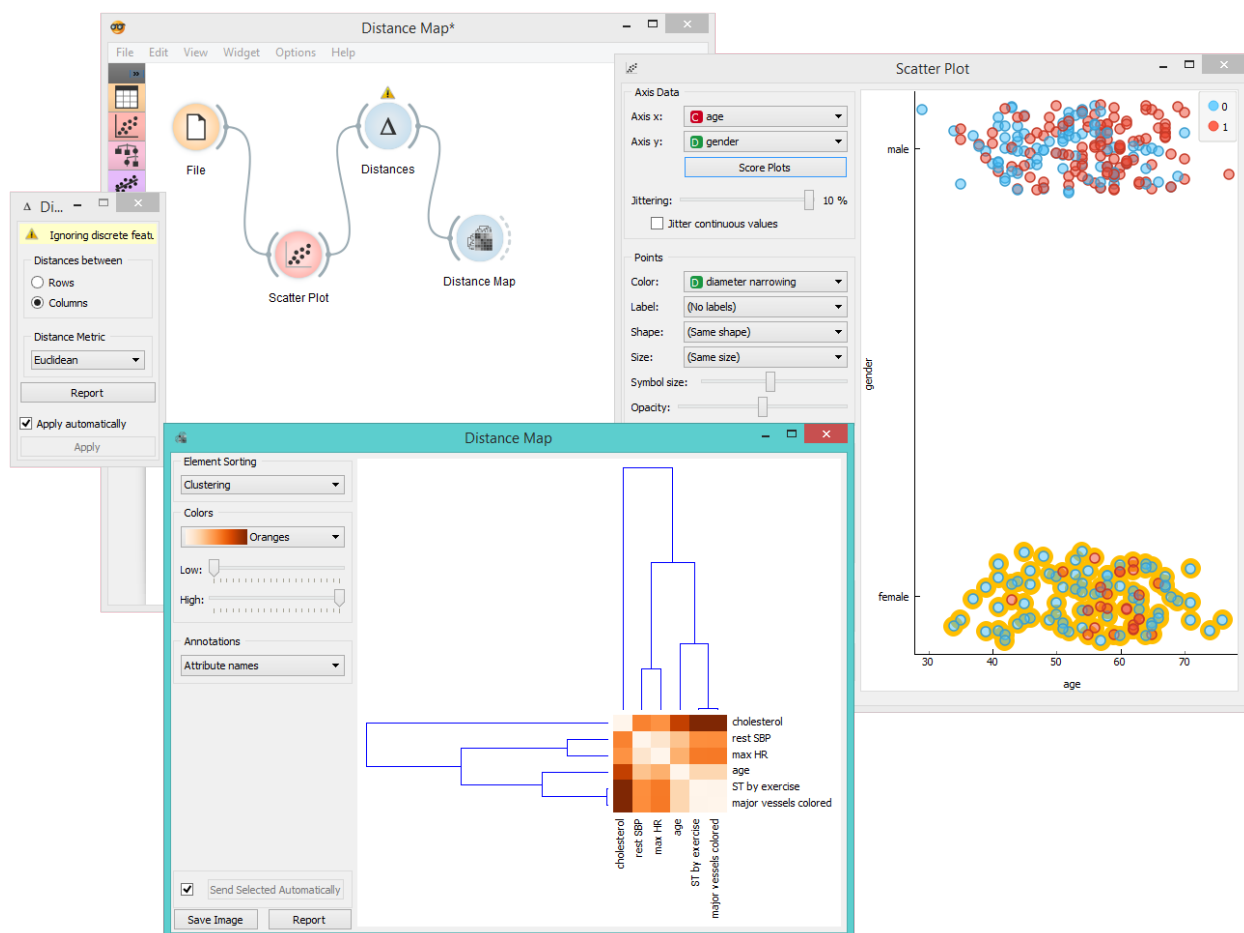
### Inputs:

- **Data**  
A data set

### Outputs:





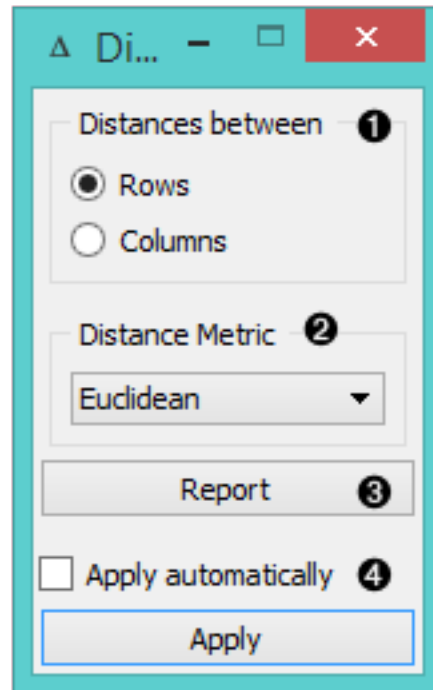


- **Distances**

A distance matrix

## Description

The **Distances** widget computes distances between rows or columns in a data set.



1. Choose whether to measure distances between rows or columns.
2. Choose the *Distance Metric*:
  - **Euclidean** (“straight line”, distance between two points)
  - **Manhattan** (the sum of absolute differences for all attributes)
  - **Cosine** (the cosine of the angle between two vectors of an inner product space)
  - **Jaccard** (the size of the intersection divided by the size of the union of the sample sets)
  - **Spearman** (linear correlation between the rank of the values, remapped as a distance in a [0, 1] interval)
  - **Spearman absolute** (linear correlation between the rank of the absolute values, remapped as a distance in a [0, 1] interval)
  - **Pearson** (linear correlation between the values, remapped as a distance in a [0, 1] interval)
  - **Pearson absolute** (linear correlation between the absolute values, remapped as a distance in a [0, 1] interval)

In case of missing values, the widget automatically imputes the average value of the row or the column.

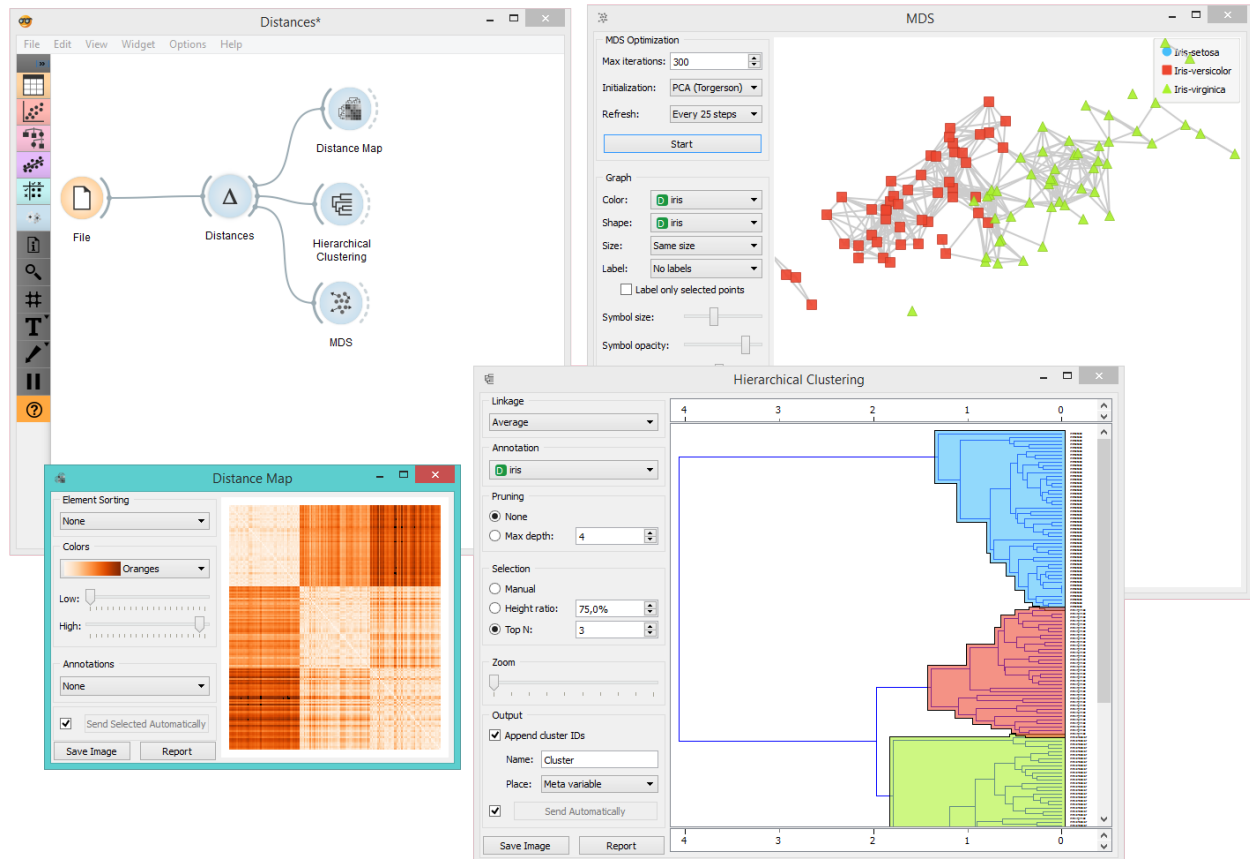
Since the widget cannot compute distances between discrete and continuous attributes, it only uses continuous attributes and ignores the discrete ones. If you want to use discrete attributes, continuize them with the *Continuize* widget first.

3. Produce a report.

4. Tick *Apply Automatically* to automatically commit changes to other widgets. Alternatively, press ‘*Apply*’.

## Example

This widget needs to be connected to another widget to display results, for instance to *Distance Map* to visualize distances, *Hierarchical Clustering* to cluster the attributes, or *MDS* to visualize the distances in a plane.



## Distance Matrix



Visualizes distance measures in a distance matrix.

## Signals

### Inputs:

- **Distances**

A distance matrix.

### Outputs:

- **Distances**

A distance matrix.

- **Table**

Distance measures in a distance matrix.

## Description

The **Distance Matrix** widget creates a distance matrix, which is a two-dimensional array containing the distances, taken pairwise, between the elements of a set. The number of elements in the data set defines the size of the matrix. Data matrices are essential for hierarchical clustering and they are extremely useful in bioinformatics as well, where they are used to represent protein structures in a coordinate-independent manner.

	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-versicolor	Iris-versicolor	Iris-versicolor	Iris-versicolor
Iris-versicolor	2.955	2.948	3.092	2.951	2.982	1.526	1.030	1.536	0.43
Iris-versicolor	2.152	2.406	2.285	2.435	2.291	2.632	2.112	2.657	0.91
Iris-versicolor	3.094	3.071	3.209	3.097	3.126	1.572	1.010	1.543	0.45
Iris-versicolor	3.076	2.960	3.176	2.990	3.069	1.421	0.843	1.425	0.76
Iris-versicolor	3.108	3.023	3.217	3.050	3.114	1.428	0.843	1.418	0.66
Iris-versicolor	3.373	3.243	3.503	3.240	3.350	0.949	0.458	0.964	0.97
Iris-versicolor	1.881	2.112	2.027	2.131	2.005	2.661	2.142	2.715	1.11
Iris-versicolor	3.023	2.970	3.142	2.990	3.040	1.490	0.922	1.487	0.54
Iris-virginica	5.324	5.132	5.418	5.167	5.305	1.844	1.808	1.616	2.66
Iris-virginica	4.164	4.104	4.274	4.135	4.193	1.449	1.063	1.253	1.34
Iris-virginica	5.365	5.171	5.491	5.167	5.325	1.407	1.688	1.187	2.70
Iris-virginica	4.706	4.562	4.815	4.584	4.696	1.245	1.183	0.990	1.95
Iris-virginica	5.085	4.923	5.197	4.942	5.070	1.463	1.493	1.212	2.35
Iris-virginica	6.174	5.958	6.300	5.950	6.124	2.121	2.500	1.936	3.50

Labels: None Report ☒ Send Automatically

1. Elements in the data set and the distances between them
2. Label the table. The options are: *none*, *enumeration*, *according to variables*.
3. Produce a report.
4. Click *Send* to communicate changes to other widgets. Alternatively, tick the box in front of the *Send* button and changes will be communicated automatically (*Send Automatically*).

The only two suitable inputs for **Distance Matrix** are the *Distances* widget and the *Distance Transformation* widget. The output of the widget is a data table containing the distance matrix. The user can decide how to label the table and the distance matrix (or instances in the distance matrix) can then be visualized or displayed in a separate data table.

## Example

The example below displays a very standard use of the **Distance Matrix** widget. We compute the distances between rows in the sample from the *Iris* data set and output them in the **Distance Matrix**. It comes as no surprise that Iris Virginica and Iris Setosa are the furthest apart.



The screenshot shows the Orange Visual Programming interface. A workflow is visible in the 'Distance Matrix\*' window, consisting of a 'File' widget connected to a 'Distances' widget, which is then connected to a 'Distance Matrix' widget. A settings panel for the 'Distances' widget is open, showing 'Distances between' set to 'Rows', 'Distance Metric' set to 'Euclidean', and 'Apply automatically' checked. The 'Distance Matrix' widget's output window displays a table of distances for the Iris dataset.

	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-versicolor	Iris-versicolor	Iris-versicolor	Iris-versicolor
Iris-versicolor	2.955	2.948	3.092	2.951	2.982	1.526	1.030	1.536	0.43
Iris-versicolor	2.152	2.406	2.285	2.435	2.291	2.632	2.112	2.657	0.91
Iris-versicolor	3.094	3.071	3.209	3.097	3.126	1.572	1.010	1.543	0.45
Iris-versicolor	3.076	2.960	3.176	2.990	3.069	1.421	0.843	1.425	0.76
Iris-versicolor	3.108	3.023	3.217	3.050	3.114	1.428	0.843	1.418	0.66
Iris-versicolor	3.373	3.243	3.503	3.240	3.350	0.949	0.458	0.964	0.97
Iris-versicolor	1.881	2.112	2.027	2.131	2.005	2.661	2.142	2.715	1.11
Iris-versicolor	3.023	2.970	3.142	2.990	3.040	1.490	0.922	1.487	0.54
Iris-virginica	5.324	5.132	5.418	5.167	5.305	1.844	1.808	1.616	2.66
Iris-virginica	4.164	4.104	4.274	4.135	4.193	1.449	1.063	1.253	1.34
Iris-virginica	5.365	5.171	5.491	5.167	5.325	1.407	1.688	1.187	2.70
Iris-virginica	4.706	4.562	4.815	4.584	4.696	1.245	1.183	0.990	1.95
Iris-virginica	5.085	4.923	5.197	4.942	5.070	1.463	1.493	1.212	2.35
Iris-virginica	6.174	5.958	6.300	5.950	6.124	2.121	2.500	1.936	3.50

## Distance Transformation



Transforms distances in a data set.

### Signals

#### Inputs:

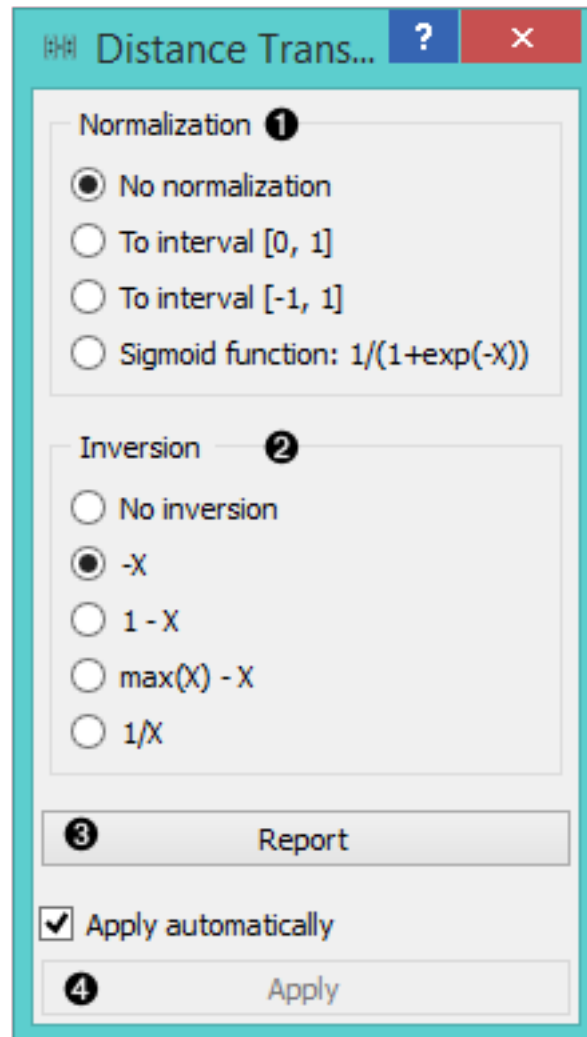
- **Distances**  
A distance matrix

#### Outputs:

- **Distances**  
A distance matrix

### Description

The **Distances Transformation** widget is used for the normalization and inversion of distance matrices. The normalization of data is necessary to bring all the variables into proportion with one another.



1. Choose the type of **Normalization**:
  - **No normalization**
  - **To interval [0, 1]**
  - **To interval [-1, 1]**
  - **Sigmoid function**:  $1/(1+\exp(-X))$
2. Choose the type of **Inversion**:
  - **No inversion**
  - **-X**
  - **1 - X**
  - **max(X) - X**
  - **1/X**
3. Produce a report.
4. After changing the settings, you need to click *Apply* to commit changes to other widgets. Alternatively, tick *Apply automatically*.

## Example

In the snapshot below, you can see how transformation affects the distance matrix. We loaded the *Iris* data set and calculated the distances between rows with the help of the *Distances* widget. In order to demonstrate how **Distance Transformation** affects the *Distance Matrix*, we created the workflow below and compared the transformed distance matrix with the “original” one.

## Distance File

Loads an existing distance file.

### Signals

#### Inputs:

- None

#### Outputs:

- **Distance File**

A distance matrix.

### Description

1. Choose from a list of previously saved distance files.
2. Browse for saved distance files.
3. Reload the selected distance file.
4. Information about the distance file (number of points, labelled/unlabelled)
5. Browse documentation data sets.

The screenshot displays the Orange Visual Programming interface with a workflow for distance transformation. The workflow consists of a 'File' widget connected to two parallel paths. The top path includes a 'Distances' widget, a 'Distance Transformation' widget, and a 'Distance Matrix (Transformed)' widget. The bottom path includes a 'Distances (1)' widget and a 'Distance Matrix' widget. A 'Distance Trans...' dialog box is open, showing settings for normalization and inversion.

**Distance Trans... Dialog Box Settings:**

- Normalization:**
  - ☒ No normalization
  - ☐ To interval [0, 1]
  - ☐ To interval [-1, 1]
  - ☐ Sigmoid function:  $1/(1+\exp(-X))$
- Inversion:**
  - ☐ No inversion
  - ☒ -X
  - ☐ 1 - X
  - ☐ max(X) - X
  - ☐ 1/X
- Buttons:** Report, Apply (with 'Apply automatically' checked).

**Distance Matrix (Transformed) Output:**

	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa
Iris-setosa		-0.539	-0.510	-0.648	-0.141	-0.616	-0.520	
Iris-setosa	-0.539		-0.300	-0.332	-0.608	-1.091	-0.510	
Iris-setosa	-0.510	-0.300		-0.245	-0.510	-1.086	-0.265	
Iris-setosa	-0.648	-0.332	-0.245		-0.648	-1.166	-0.332	
Iris-setosa	-0.141	-0.608	-0.510	-0.648		-0.616	-0.458	
Iris-setosa	-0.616	-1.091	-1.086	-1.166	-0.616		-0.995	
Iris-setosa	-0.520	-0.510	-0.265	-0.332	-0.458	-0.995		
Iris-setosa	-0.173	-0.424	-0.412	-0.500	-0.224	-0.700	-0.424	
Iris-setosa	-0.922	-0.510	-0.436	-0.300	-0.922	-1.459	-0.548	
Iris-setosa	-0.469	-0.173	-0.316	-0.316	-0.529	-1.010	-0.480	
Iris-setosa	-0.374	-0.866	-0.883	-1.000	-0.424	-0.346	-0.866	
Iris-setosa	-0.374	-0.458	-0.374	-0.374	-0.346	-0.812	-0.300	
Iris-setosa	-0.592	-0.141	-0.265	-0.265	-0.640	-1.162	-0.490	

**Distance Matrix Output:**

	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa
Iris-setosa		0.539	0.510	0.648	0.141	0.616	0.520	0.173
Iris-setosa	0.539		0.300	0.332	0.608	1.091	0.510	0.424
Iris-setosa	0.510	0.300		0.245	0.510	1.086	0.265	0.412
Iris-setosa	0.648	0.332	0.245		0.648	1.166	0.332	0.500
Iris-setosa	0.141	0.608	0.510	0.648		0.616	0.458	0.224
Iris-setosa	0.616	1.091	1.086	1.166	0.616		0.995	0.700
Iris-setosa	0.520	0.510	0.265	0.332	0.458	0.995		0.424
Iris-setosa	0.173	0.424	0.412	0.500	0.224	0.700	0.424	
Iris-setosa	0.922	0.510	0.436	0.300	0.922	1.459	0.548	0.787
Iris-setosa	0.469	0.173	0.316	0.316	0.529	1.010	0.480	0.332
Iris-setosa	0.374	0.866	0.883	1.000	0.424	0.346	0.866	0.500
Iris-setosa	0.374	0.458	0.374	0.374	0.346	0.812	0.300	0.224
Iris-setosa	0.592	0.141	0.265	0.265	0.640	1.162	0.490	0.469
Iris-setosa	0.995	0.678	0.500	0.520	0.975	1.572	0.616	0.906
Iris-setosa	0.883	1.360	1.364	1.530	0.917	0.678	1.360	1.044



The screenshot shows the 'Distance File' widget interface. It includes a file selection dropdown, a file icon, a 'Reload' button, an 'Info' section, and a 'Report' button.

**Distance File**

Distance File ①

Iris\_inversion.dst

Info ④

150 points(s), unlabelled

Browse documentation data sets ⑤

Report ⑥

6. Produce a report.

## Example

When you want to use a custom-set distance file that you've saved before, open the **Distance File** widget and select the desired file with the *Browse* icon. This widget loads the existing distance file. In the snapshot below, we loaded the transformed *Iris* distance matrix from the *Save Distance Matrix* example. We displayed the transformed data matrix in the *Distance Map* widget. We also decided to display a distance map of the original *Iris* data set for comparison.



## Save Distance Matrix



Saves a distance matrix.

## Signals

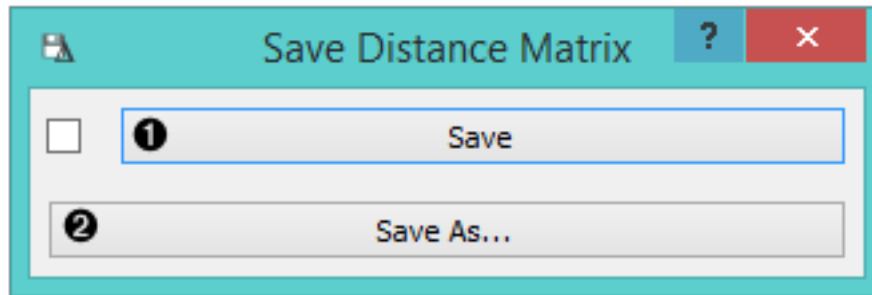
Inputs:

- Distances

A distance matrix.

**Outputs:**

- None

**Description**

1. By clicking *Save*, you choose from previously saved distance matrices. Alternatively, tick the box on the left side of the *Save* button and changes will be communicated automatically.
2. By clicking *Save as*, you save the distance matrix to your computer, you only need to enter the name of the file and click *Save*. The distance matrix will be saved as type *.dst*.

**Example**

In the snapshot below, we used the *Distance Transformation* widget to transform the distances in the *Iris* data set. We then chose to save the transformed version to our computer, so we could use it later on. We decided to output all data instances. You can choose to output just a minor subset of the data matrix. Pairs are marked automatically. If you wish to know what happened to our changed file, go [here](#)

## Hierarchical Clustering

Groups items using a hierarchical clustering algorithm.

**Signals****Inputs:**

- **Distances**  
A distance matrix

**Outputs:**

- **Selected Data**  
A data subset
- **Other Data**  
Remaining data

The workflow in the 'Save Distance Matrix\*' window is as follows:

```

graph LR
    File[File] --> Distances[Distances]
    Distances --> DistanceTransformation[Distance Transformation]
    DistanceTransformation --> DistanceMatrixInversion[Distance Matrix (Inversion)]
    DistanceMatrixInversion --> SaveDistanceMatrix[Save Distance Matrix]
  
```

The 'Save Distance Matrix' dialog box shows the following options:

- ☐ Save
- ☐ Save As...

The 'Distance Matrix (Inversion)' widget displays the following table:

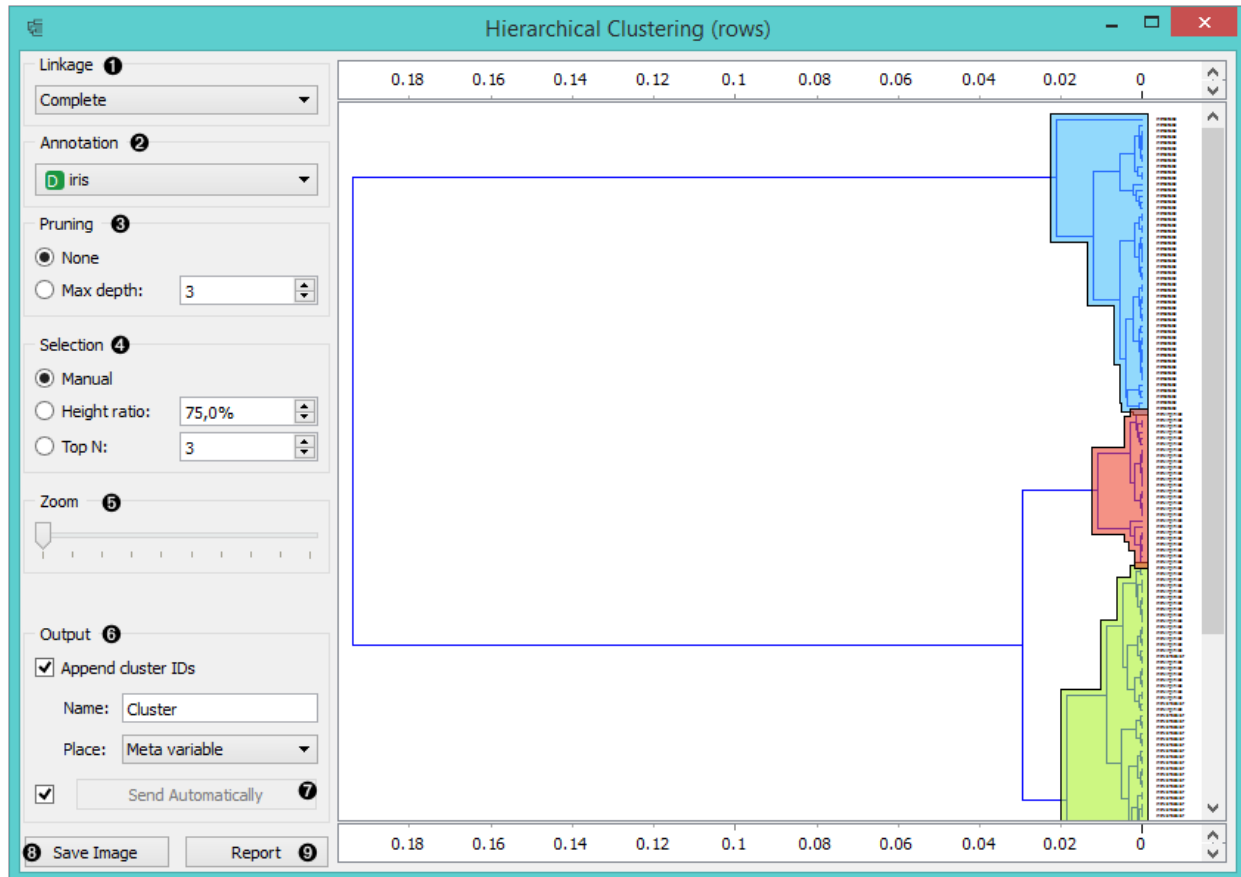
	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa
Iris-setosa		-0.539	-0.510	-0.648	-0.141	-0.616	-0.520	
Iris-setosa	-0.539		-0.300	-0.332	-0.608	-1.091	-0.510	
Iris-setosa	-0.510	-0.300		-0.245	-0.510	-1.086	-0.265	
Iris-setosa	-0.648	-0.332	-0.245		-0.648	-1.166	-0.332	
Iris-setosa	-0.141	-0.608	-0.510	-0.648		-0.616	-0.458	
Iris-setosa	-0.616	-1.091	-1.086	-1.166	-0.616		-0.995	
Iris-setosa	-0.520	-0.510	-0.265	-0.332	-0.458	-0.995		
Iris-setosa	-0.173	-0.424	-0.412	-0.500	-0.224	-0.700	-0.424	
Iris-setosa	-0.922	-0.510	-0.436	-0.300	-0.922	-1.459	-0.548	
Iris-setosa	-0.469	-0.173	-0.316	-0.316	-0.529	-1.010	-0.480	
Iris-setosa	-0.374	-0.866	-0.883	-1.000	-0.424	-0.346	-0.866	
Iris-setosa	-0.374	-0.458	-0.374	-0.374	-0.346	-0.812	-0.300	
Iris-setosa	-0.592	-0.141	-0.265	-0.265	-0.640	-1.162	-0.490	
Iris-setosa	-0.995	-0.678	-0.500	-0.520	-0.975	-1.572	-0.616	

Labels:  Report ☒ Send Automatically



## Description

The widget computes **hierarchical clustering** of arbitrary types of objects from a matrix of distances and shows a corresponding **dendrogram**.



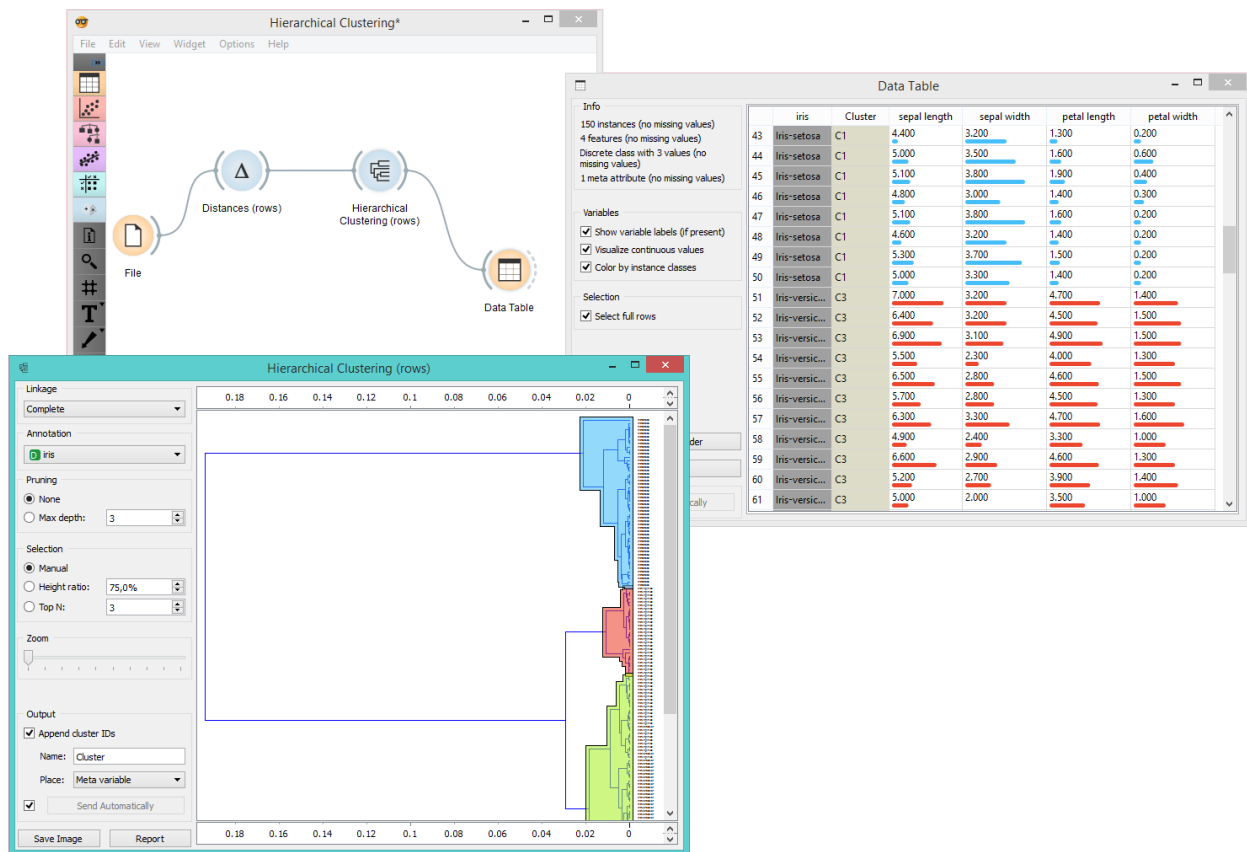
1. The widget supports four ways of measuring distances between clusters:
  - **Single linkage** computes the distance between the closest elements of the two clusters
  - **Average linkage** computes the average distance between elements of the two clusters
  - **Weighted linkage** uses the **WPGMA** method
  - **Complete linkage** computes the distance between the clusters' most distant elements
2. Labels of nodes in the dendrogram can be chosen in the **Annotation** box.
3. Huge dendrograms can be pruned in the *Pruning* box by selecting the maximum depth of the dendrogram. This only affects the display, not the actual clustering.
4. The widget offers three different selection methods:
  - **Manual** (Clicking inside the dendrogram will select a cluster. Multiple clusters can be selected by holding Ctrl/Cmd. Each selected cluster is shown in a different color and is treated as a separate cluster in the output.)
  - **Height ratio** (Clicking on the bottom or top ruler of the dendrogram places a cutoff line in the graph. Items to the right of the line are selected.)
  - **Top N** (Selects the number of top nodes.)



5. Use *Zoom* and scroll to zoom in or out.
6. If the items being clustered are instances, they can be added a cluster index (*Append cluster IDs*). The ID can appear as an ordinary **Attribute**, **Class attribute** or a **Meta attribute**. In the second case, if the data already has a class attribute, the original class is placed among meta attributes.
7. The data can be automatically output on any change (*Auto send is on*) or, if the box isn't ticked, by pushing *Send Data*.
8. Clicking this button produces an image that can be saved.
9. Produce a report.

## Examples

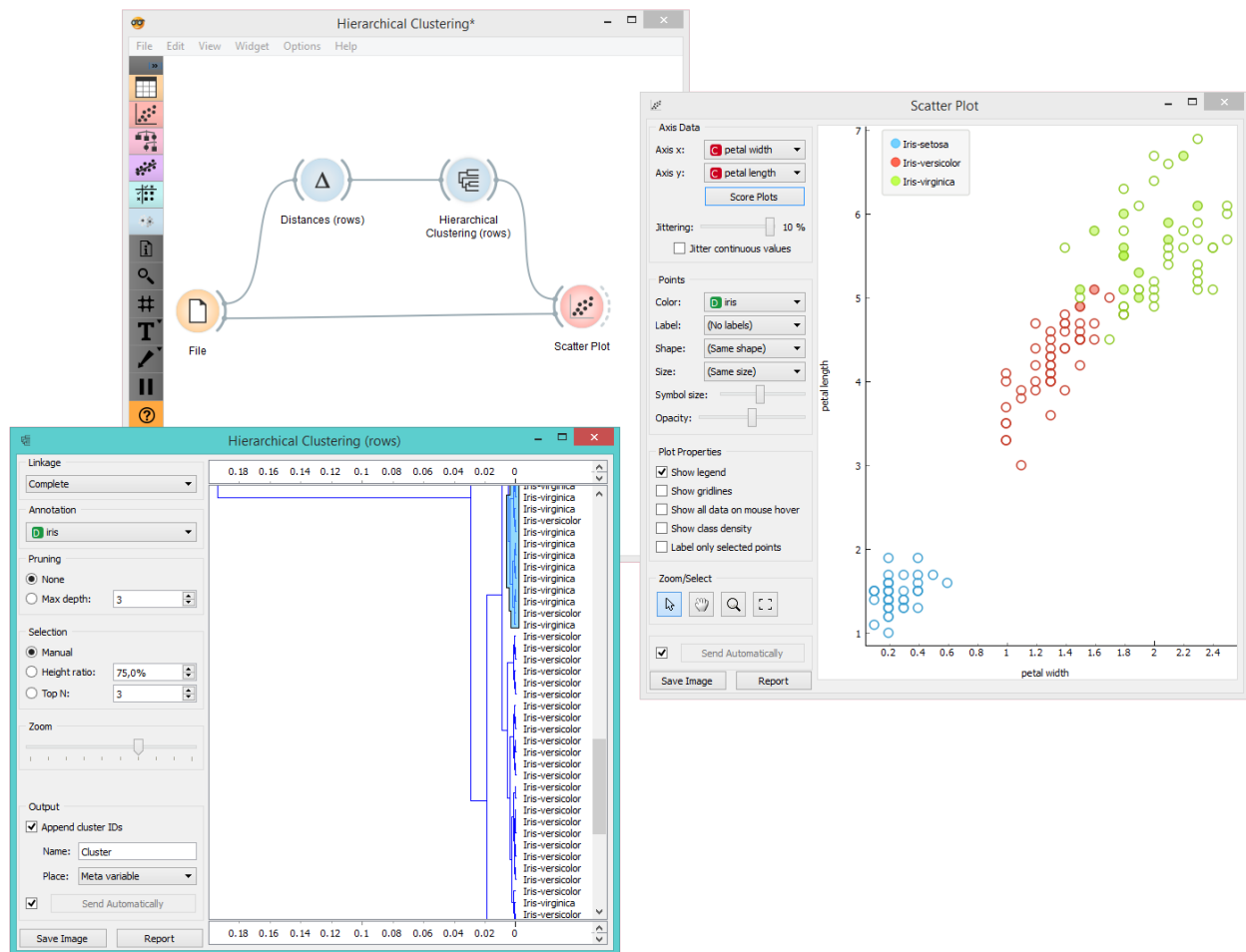
The workflow below shows the output of **Hierarchical Clustering** for the *Iris* data set in *Data Table* widget. We see that if we choose *Append cluster IDs* in hierarchical clustering, we can see an additional column in the *Data Table* named *Cluster*. This is a way to check how hierarchical clustering clustered individual instances.



In the second example, we loaded the *Iris* data set again, but this time we added the *Scatter Plot*, showing all the instances from the *File* widget, while at the same time receiving the selected instances signal from **Hierarchical Clustering**. This way we can observe the position of the selected cluster(s) in the projection.

## k-Means

Groups items using the k-Means clustering algorithm.



## Signals

### Inputs:

- **Data**

A data set.

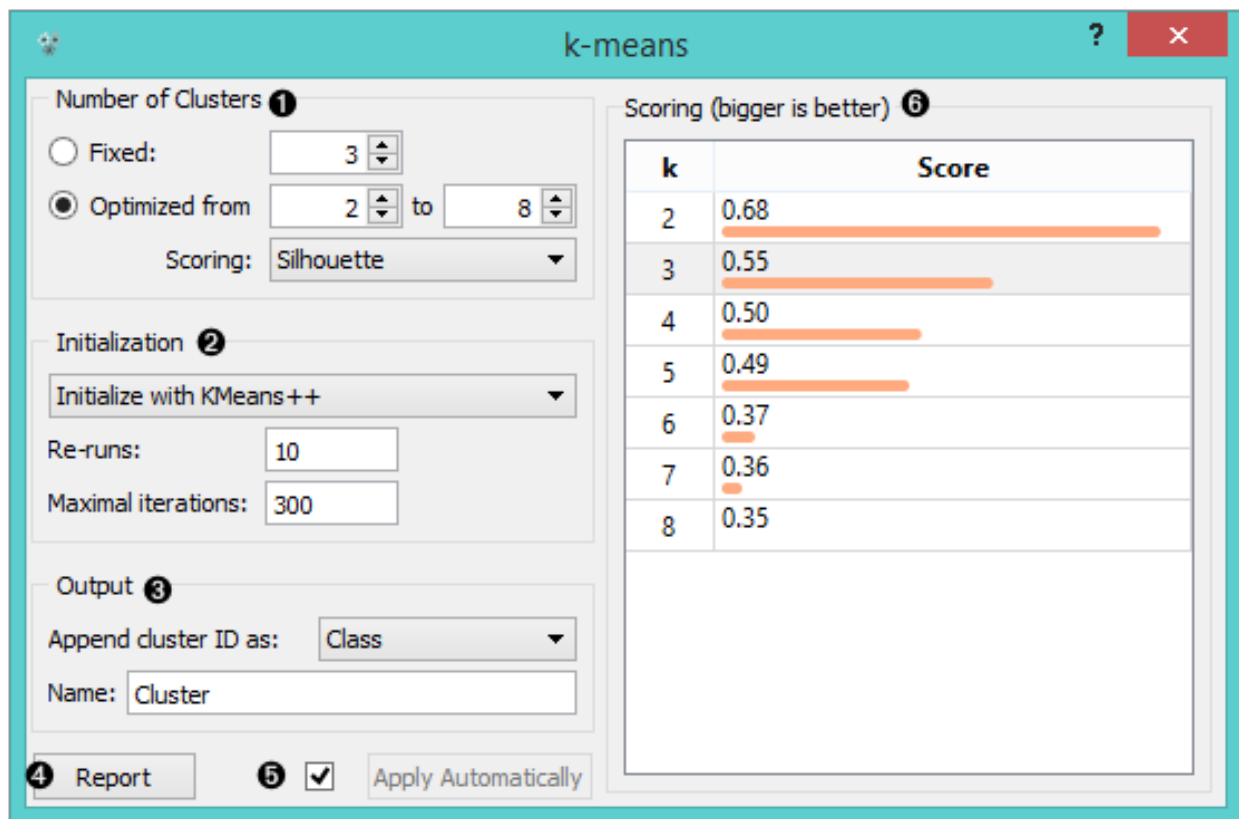
### Outputs:

- **Data**

A data set with cluster index as a class attribute.

## Description

The widget applies the [k-Means clustering](#) algorithm to the data and outputs a new data set in which the cluster index is used as a class attribute. The original class attribute, if it exists, is moved to meta attributes. Scores of clustering results for various k are also shown in the widget.



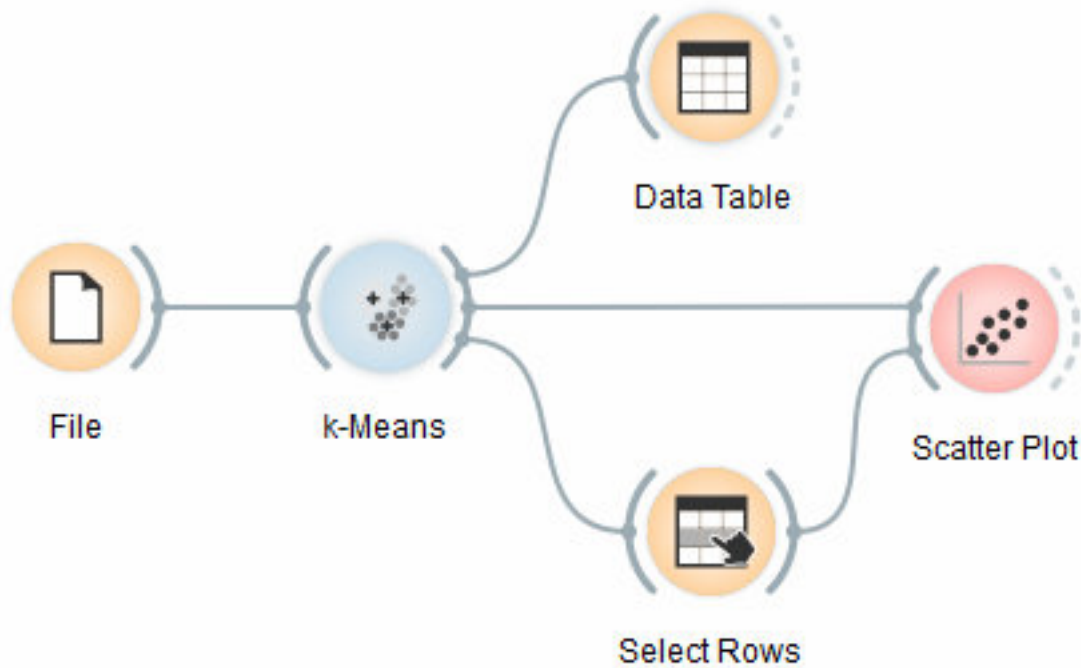
1. Select the number of clusters.

- **Fixed:** algorithm clusters data in a specified number of clusters.
- **Optimized:** widget shows clustering scores for the selected cluster range.
- **Silhouette** (contrasts average distance to elements in the same cluster with the average distance to elements in other clusters)
- **Inter-cluster distance** (measures distances between clusters, normally between centroids)

- **Distance to centroids** (measures distances to the arithmetic means of clusters)
2. Select the initialization method (the way the algorithm begins clustering):
    - **k-Means++** (first center is selected randomly, subsequent are chosen from the remaining points with probability proportioned to squared distance from the closest center)
    - **Random initialization** (clusters are assigned randomly at first and then updated with further iterations)
- Re-runs** (how many times the algorithm is run) and **maximal iterations** (the maximum number of iteration within each algorithm run) can be set manually.
3. The widget outputs a new data set with appended cluster information. Select how to append cluster information (as class, feature or meta attribute) and name the column.
  4. If *Apply Automatically* is ticked, the widget will commit changes automatically. Alternatively, click *Apply*.
  5. Produce a report.
  6. Check scores of clustering results for various k.

## Examples

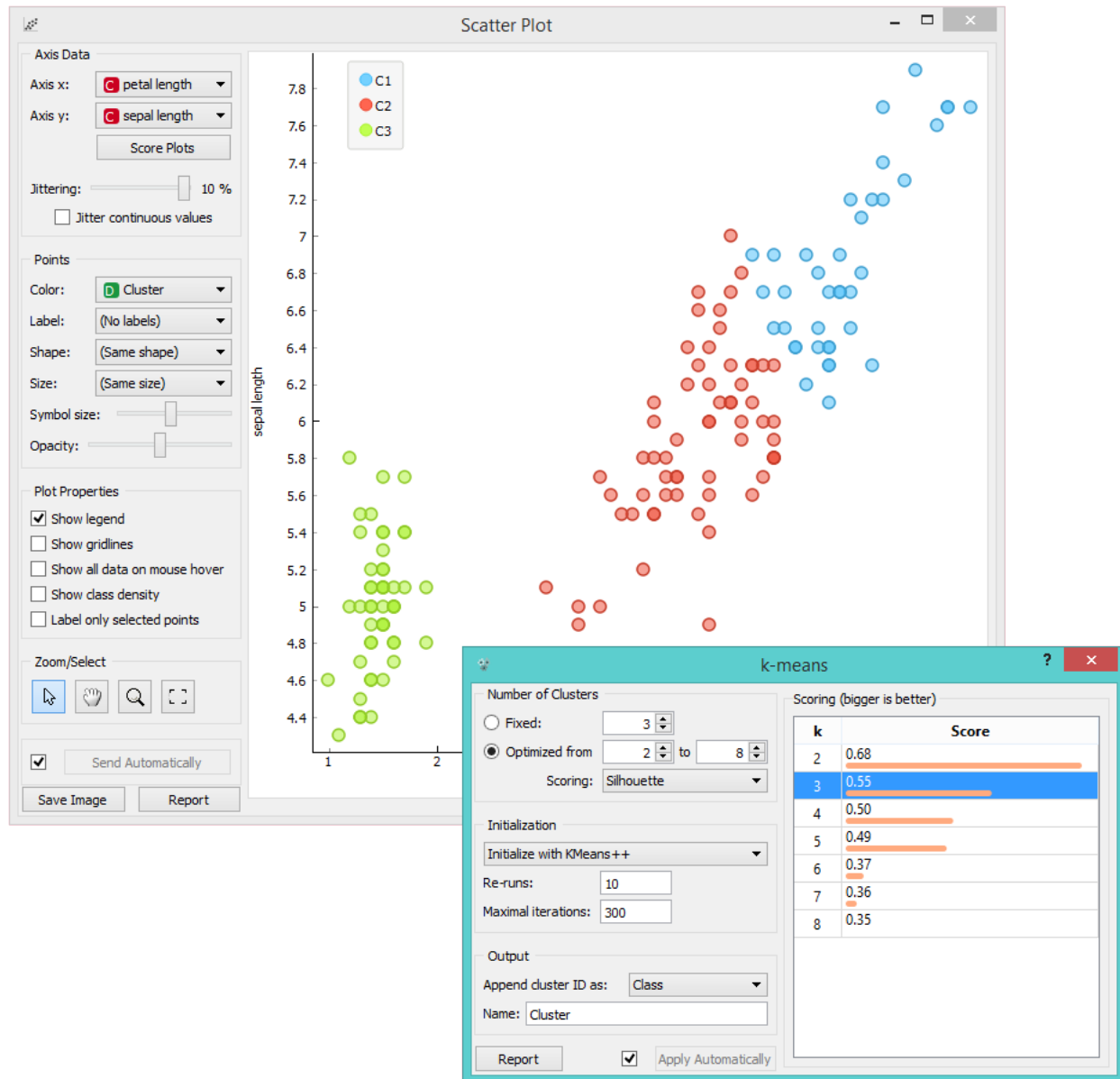
We are going to explore the widget with the following schema.



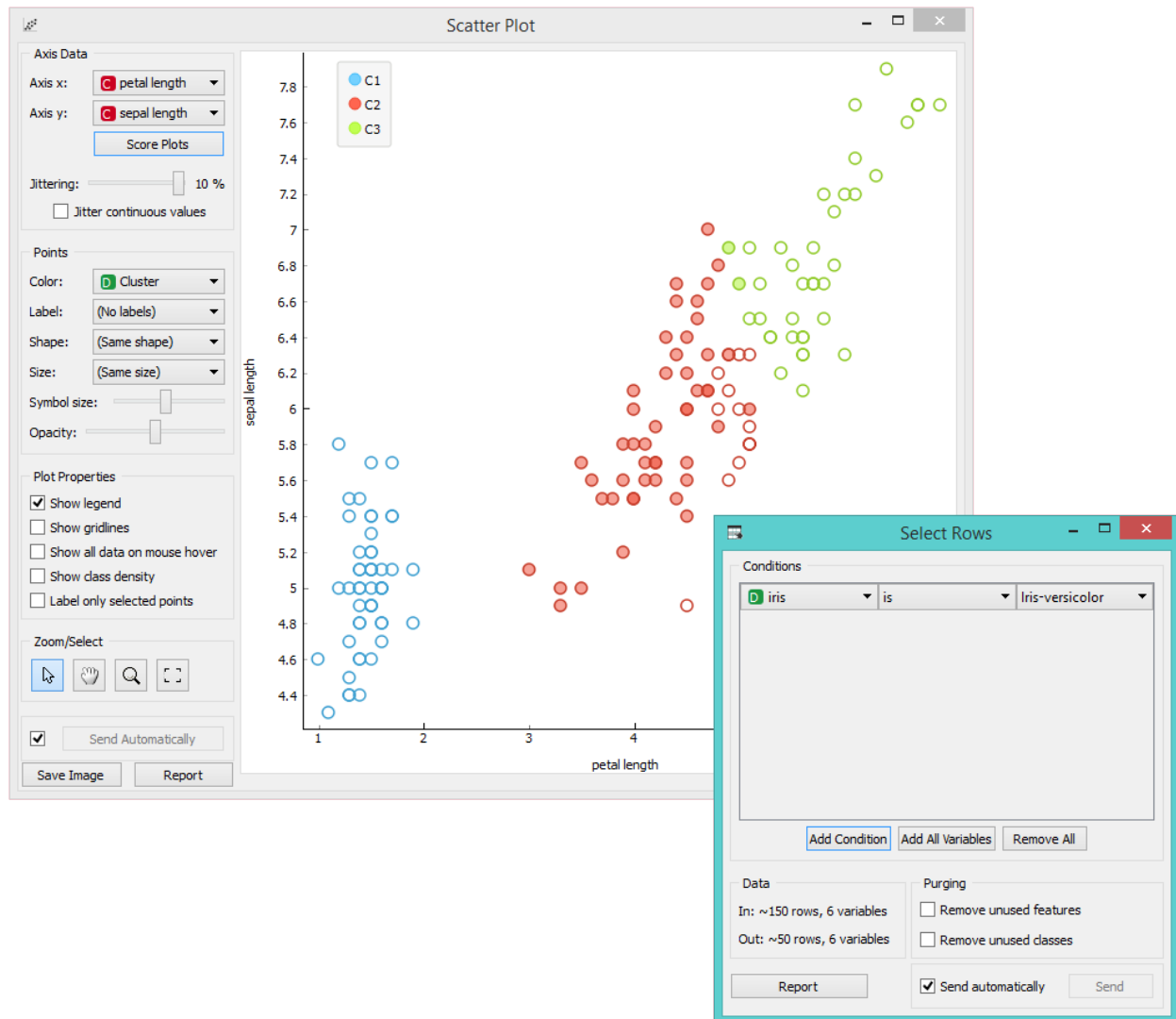
First, we load the *Iris* data set, divide it into three clusters and show it in the *Data Table*, where we can observe which instance went into which cluster. The interesting parts are the *Scatter Plot* and *Select Rows*.

Since **k-Means** added the cluster index as a class attribute, the scatter plot will color the points according to the clusters they are in.

What we are really interested in is how well the clusters induced by the (unsupervised) clustering algorithm match the actual classes in the data. We thus take *Select Rows* widget, in which we can select individual classes and have the



corresponding points marked in the scatter plot. The match is perfect for *setosa*, and pretty good for the other two classes.



You may have noticed that we left the **Remove unused values/attributes** and **Remove unused classes** in *Select Rows* unchecked. This is important: if the widget modifies the attributes, it outputs a list of modified instances and the scatter plot cannot compare them to the original data.

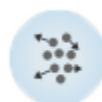
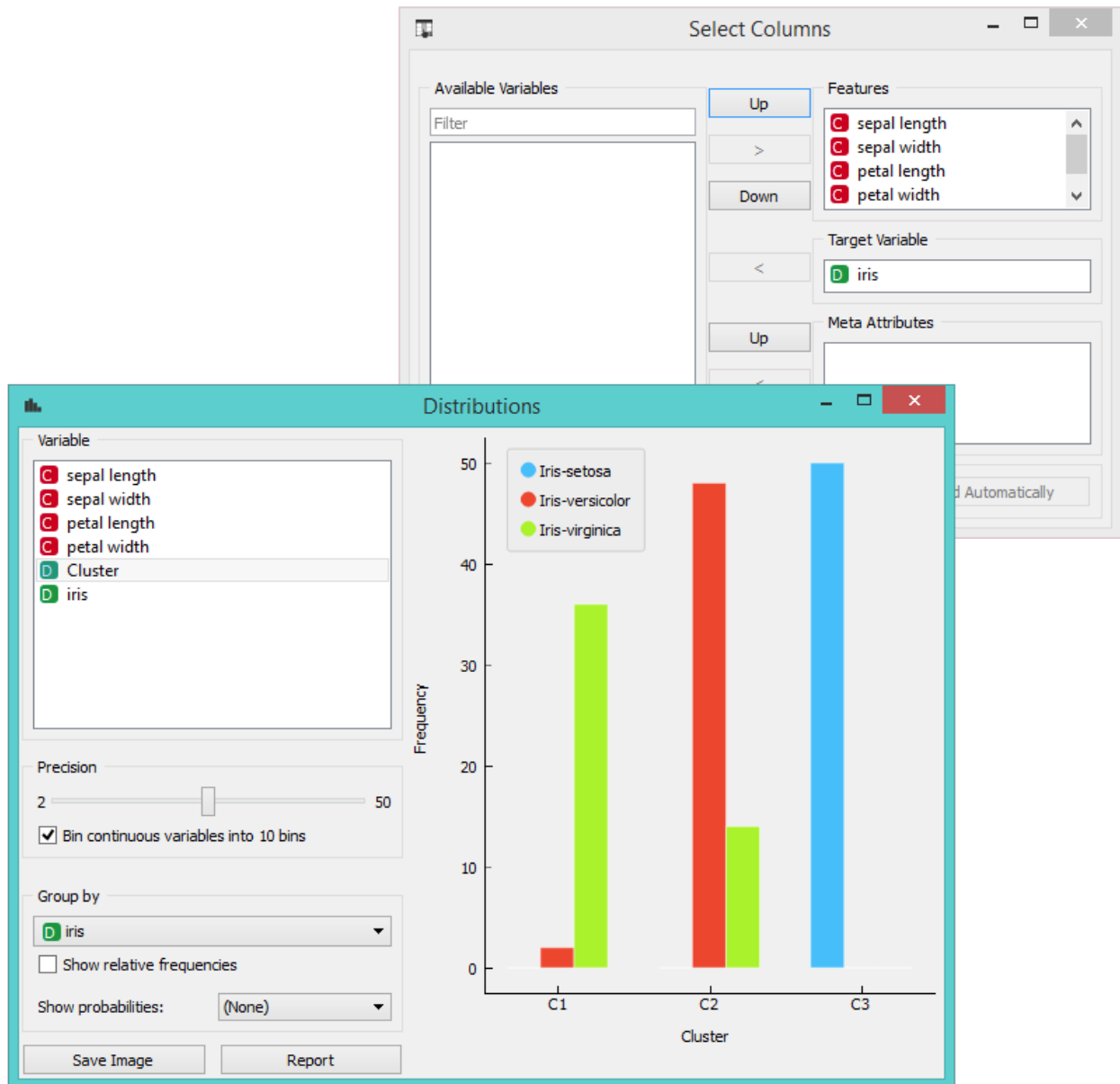
Perhaps a simpler way to test the match between clusters and the original classes is to use the *Distributions* widget.

The only (minor) problem here is that this widget only visualizes normal (and not meta) attributes. We solve this by using *Select Columns*: we reinstate the original class *Iris* as the class and put the cluster index among the attributes.

The match is perfect for *setosa*: all instances of *setosa* are in the third cluster (blue). 48 *versicolors* are in the second cluster (red), while two ended up in the first. For *virginicae*, 36 are in the first cluster and 14 in the second.

## MDS

Multidimensional scaling (MDS) projects items onto a plane fitted to given distances between points.



## Signals

### Inputs:

- **Distances**  
A distance matrix
- **Data**  
A data set

### Outputs:

- **Data**  
A data set with MDS coordinates.
- **Data subset**  
Selected data

## Description

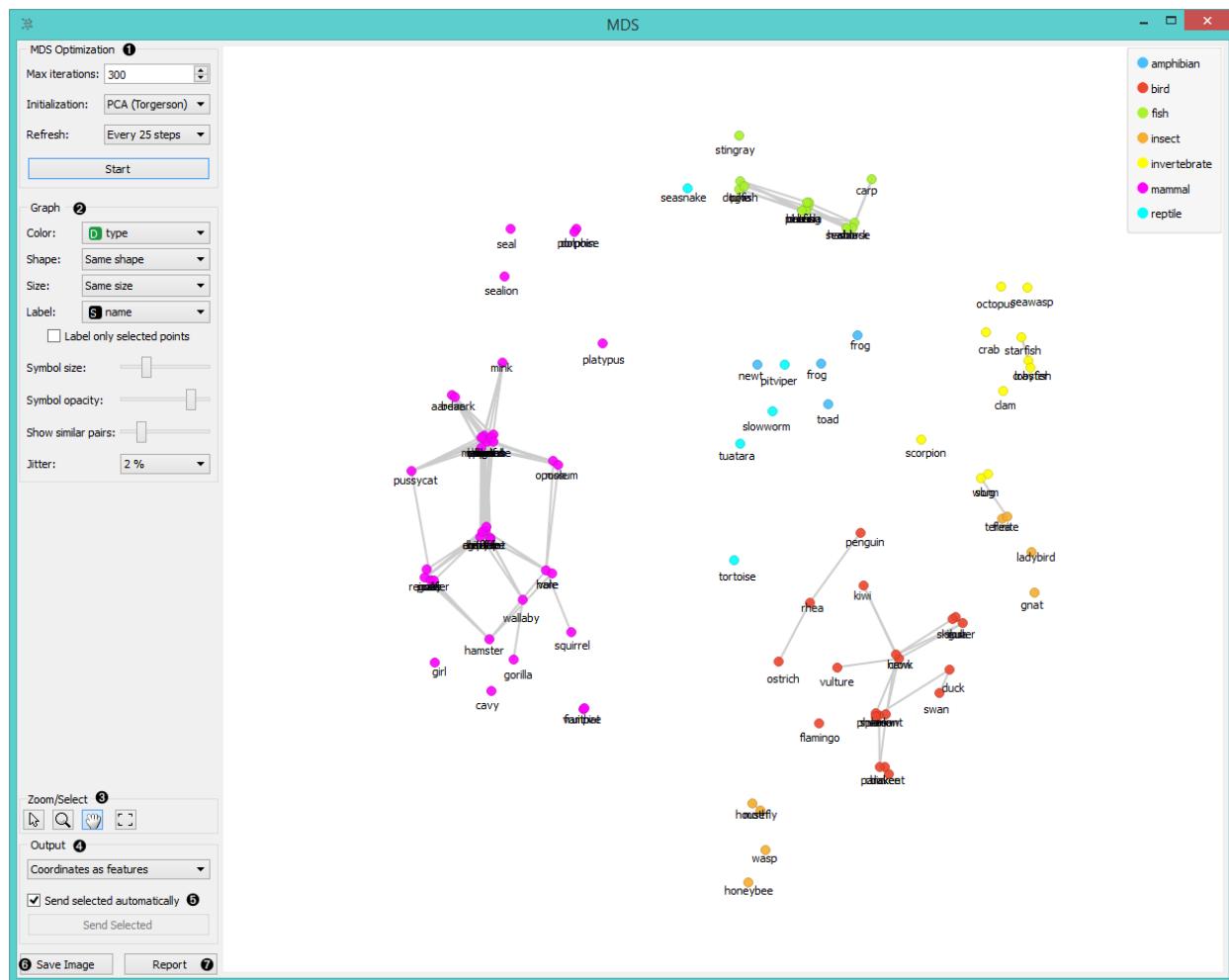
**Multidimensional scaling** is a technique which finds a low-dimensional (in our case a two-dimensional) projection of points, where it tries to fit distances between points as well as possible. The perfect fit is typically impossible to obtain since the data is high-dimensional or the distances are not [Euclidean](#).

In the input, the widget needs either a data set or a matrix of distances. When visualizing distances between rows, you can also adjust the color of the points, change their shape, mark them, and output them upon selection.

The algorithm iteratively moves the points around in a kind of a simulation of a physical model: if two points are too close to each other (or too far away), there is a force pushing them apart (or together). The change of the point's position at each time interval corresponds to the sum of forces acting on it.

1. The widget redraws the projection during optimization. Optimization is run automatically in the beginning and later by pushing *Start*.
  - **Max iterations:** The optimization stops either when the projection changes only minimally at the last iteration or when a maximum number of iterations has been reached.
  - **Initialization:** PCA (Torgerson) positions the initial points along principal coordinate axes. *Random* sets the initial points to a random position and then readjusts them.
  - **Refresh:** Set how often you want to refresh the visualization. It can be at *Every iteration*, *Every 5/10/25/50 steps* or never (*None*). Setting a lower refresh interval makes the animation more visually appealing, but can be slow if the number of points is high.
2. Defines how the points are visualized. These options are available only when visualizing distances between rows (selected in the [Distances](#) widget).
  - **Color:** Color of points by attribute (gray for continuous, colored for discrete).
  - **Shape:** Shape of points by attribute (only for discrete).
  - **Size:** Set the size of points (*Same size* or select an attribute) or let the size depend on the value of the continuous attribute the point represents (Stress).
  - **Label:** Discrete attributes can serve as a label.
  - **Symbol size:** Adjust the size of the dots.
  - **Symbol opacity:** Adjust the transparency level of the dots.
  - **Show similar pairs:** Adjust the strength of network lines.





- **Jitter:** Set [jittering](#) to prevent the dots from overlapping.
3. Adjust the graph with *Zoom/Select*. The arrow enables you to select data instances. The magnifying glass enables zooming, which can be also done by scrolling in and out. The hand allows you to move the graph around. The rectangle readjusts the graph proportionally.
  4. Select the desired output:
    - **Original features only** (input data set)
    - **Coordinates only** (MDS coordinates)
    - **Coordinates as features** (input data set + MDS coordinates as regular attributes)
    - **Coordinates as meta attributes** (input data set + MDS coordinates as meta attributes)
  5. Sending the instances can be automatic if *Send selected automatically* is ticked. Alternatively, click *Send selected*.
  6. **Save Image** allows you to save the created image either as .svg or .png file to your device.
  7. Produce a report.

The MDS graph performs many of the functions of the Visualizations widget. It is in many respects similar to the [Scatter Plot](#) widget, so we recommend reading that widget's description as well.

## Example

The above graphs were drawn using the following simple schema. We used the *iris.tab* data set. Using the [Distances](#) widget we input the distance matrix into the **MDS** widget, where we see the *Iris* data displayed in a 2-dimensional plane. We can see the appended coordinates in the [Data Table](#) widget.

## References

Wickelmaier, F. (2003). An Introduction to MDS. Sound Quality Research Unit, Aalborg University. Available [here](#).

## Manifold Learning

Nonlinear dimensionality reduction.

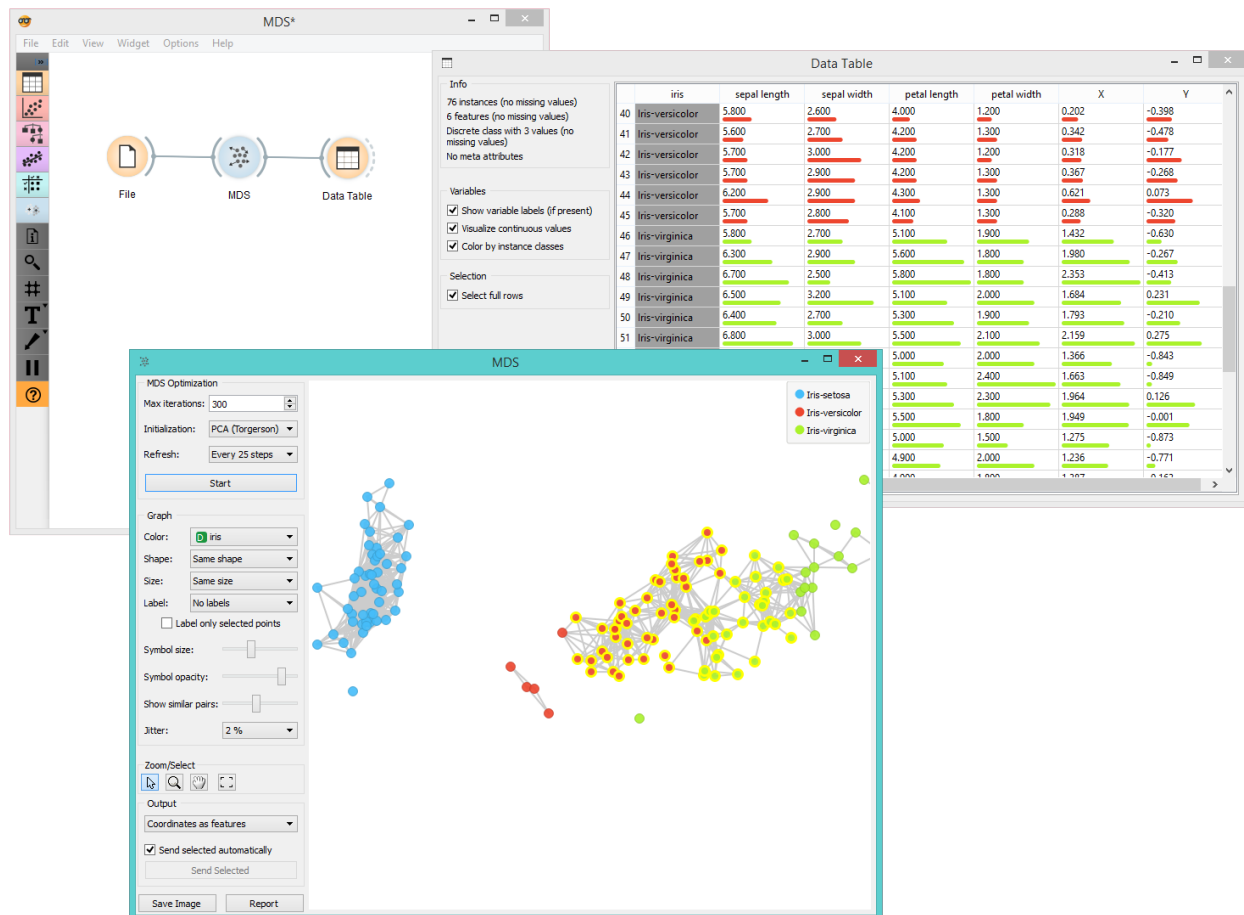
## Signals

### Inputs:

- **Data**  
A data set

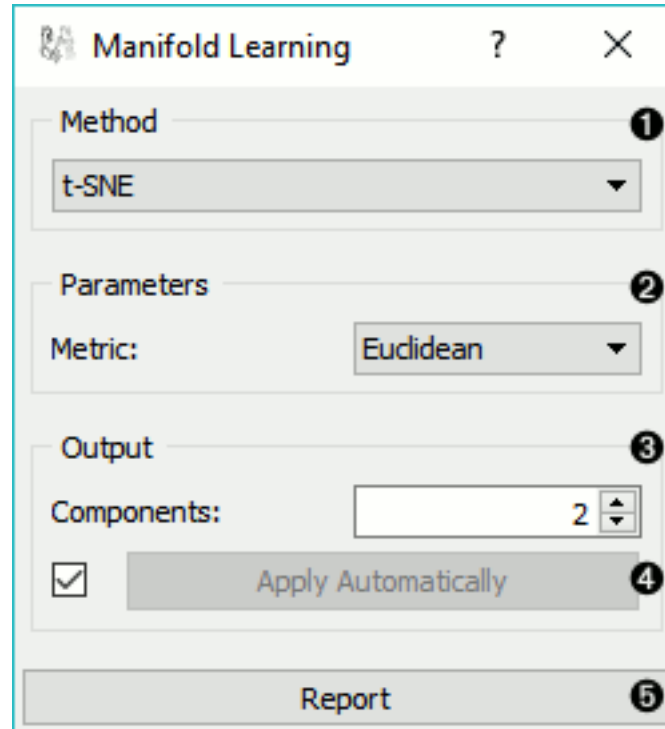
### Outputs:

- **Transformed Data**  
A data set with new, reduced coordinates.



## Description

**Manifold Learning** is a technique which finds a non-linear manifold within the higher-dimensional space. The widget then outputs new coordinates which correspond to a two-dimensional space. Such data can be later visualized with *Scatter Plot* or other visualization widgets.



1. Method for manifold learning: - t-SNE - MDS, see also *MDS widget* - Isomap - Locally Linear Embedding - Spectral Embedding
2. Set parameters for the method: - t-SNE (distance measures):
  - *Euclidean* distance
  - *Manhattan*
  - *Chebyshev*
  - *Jaccard*
  - *Mahalanobis*
  - *Cosine*
  - **MDS (iterations and initialization):**
    - *max iterations*: maximum number of optimization iterations
    - *initialization*: method for initialization of the algorithm (PCA or random)
  - **Isomap:**
    - number of *neighbors*
  - **Locally Linear Embedding:**
    - *method*:

- \* standard
- \* modified
- \* hessian eigenmap
- \* local
- number of *neighbors*
- *max iterations*
- **Spectral Embedding:**
  - *affinity*:
    - \* nearest neighbors
    - \* RFB kernel

3. Output: the number of reduced features (components).

4. If *Apply automatically* is ticked, changes will be propagated automatically. Alternatively, click *Apply*.

5. Produce a report.

**Manifold Learning** widget produces different embeddings for high-dimensional data.

... figure:: images/collage-manifold.png

From left to right, top to bottom: t-SNE, MDS, Isomap, Locally Linear Embedding and Spectral Embedding.

## Example

*Manifold Learning* widget transforms high-dimensional data into a lower dimensional approximation. This makes it great for visualizing data sets with many features. We used *voting.tab* to map 16-dimensional data onto a 2D graph. Then we used *Scatter Plot* to plot the embeddings.

## Evaluation

### Calibration Plot

Shows the match between classifiers' probability predictions and actual class probabilities.

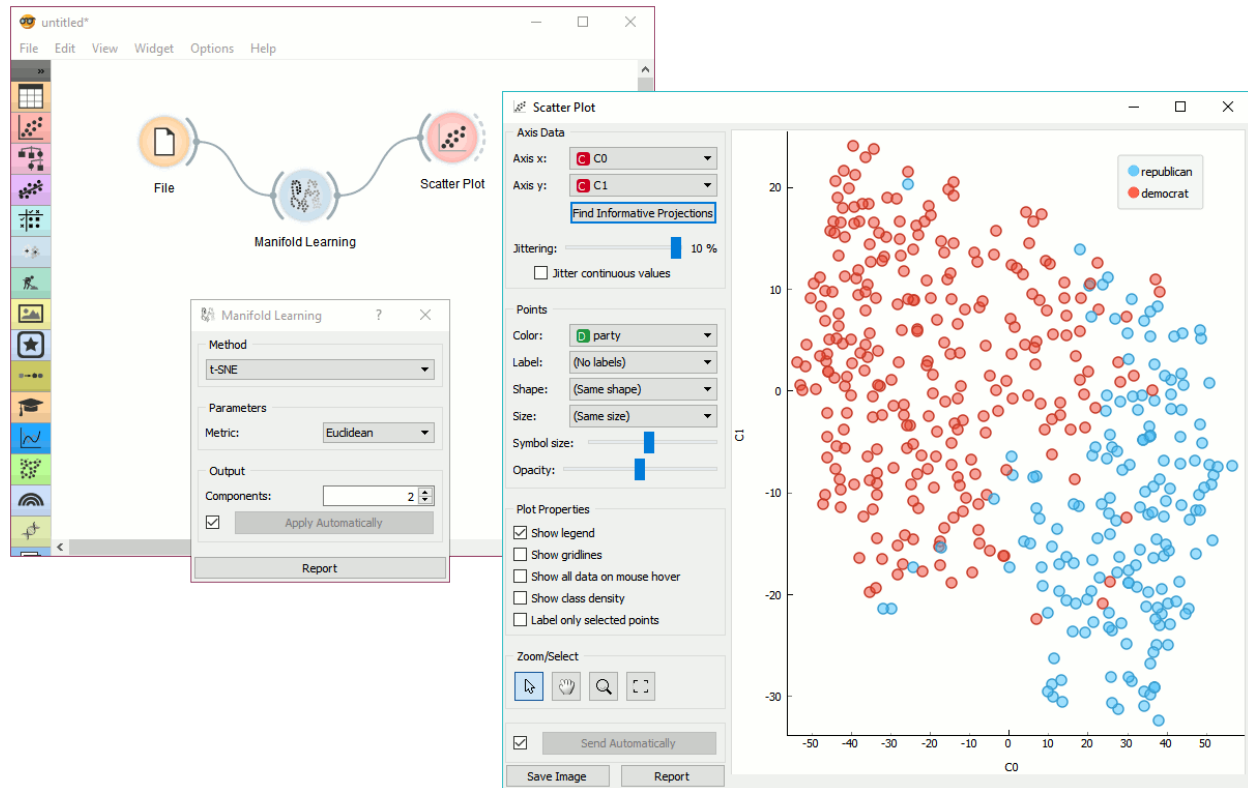
### Signals

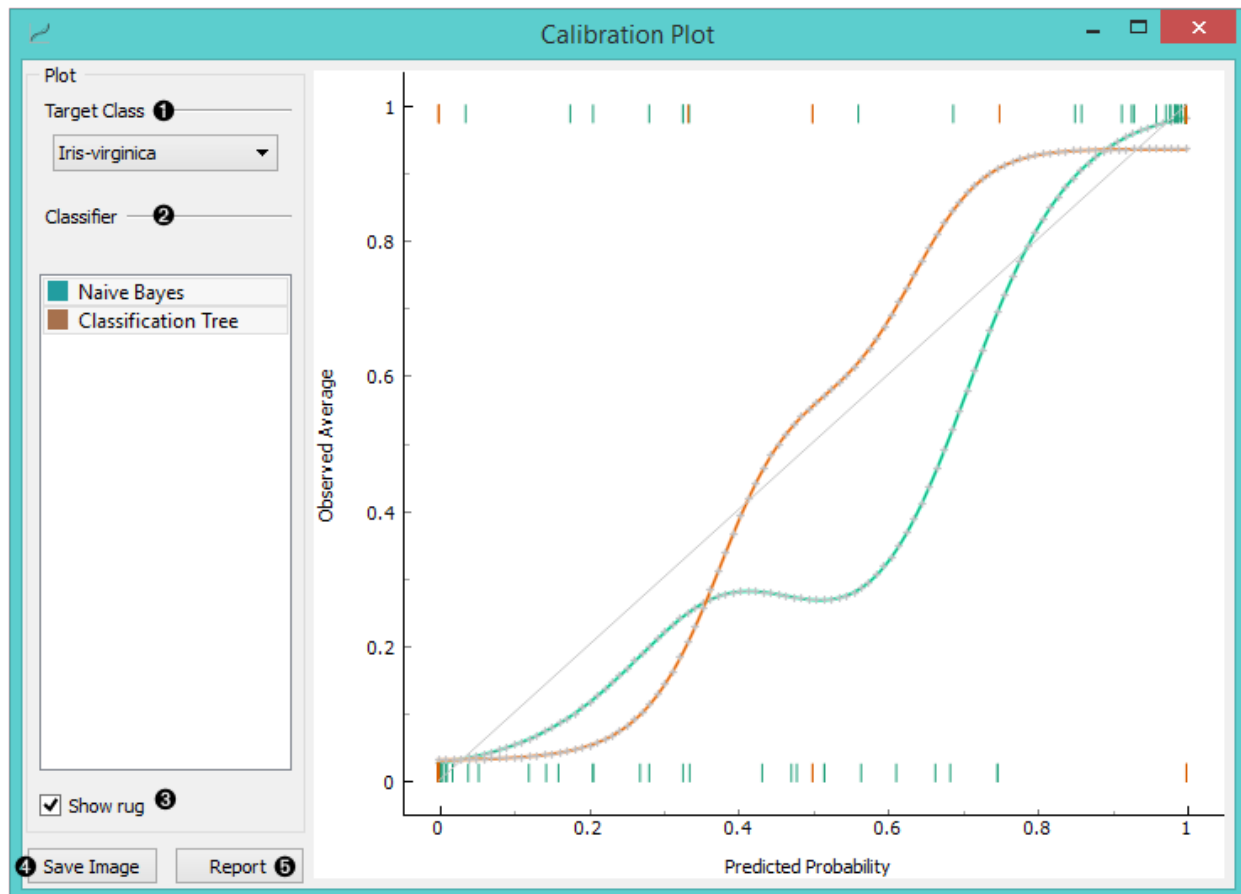
#### Inputs:

- **Evaluation Results**
  - Results of testing classification algorithms.

#### Outputs:

- None





## Description

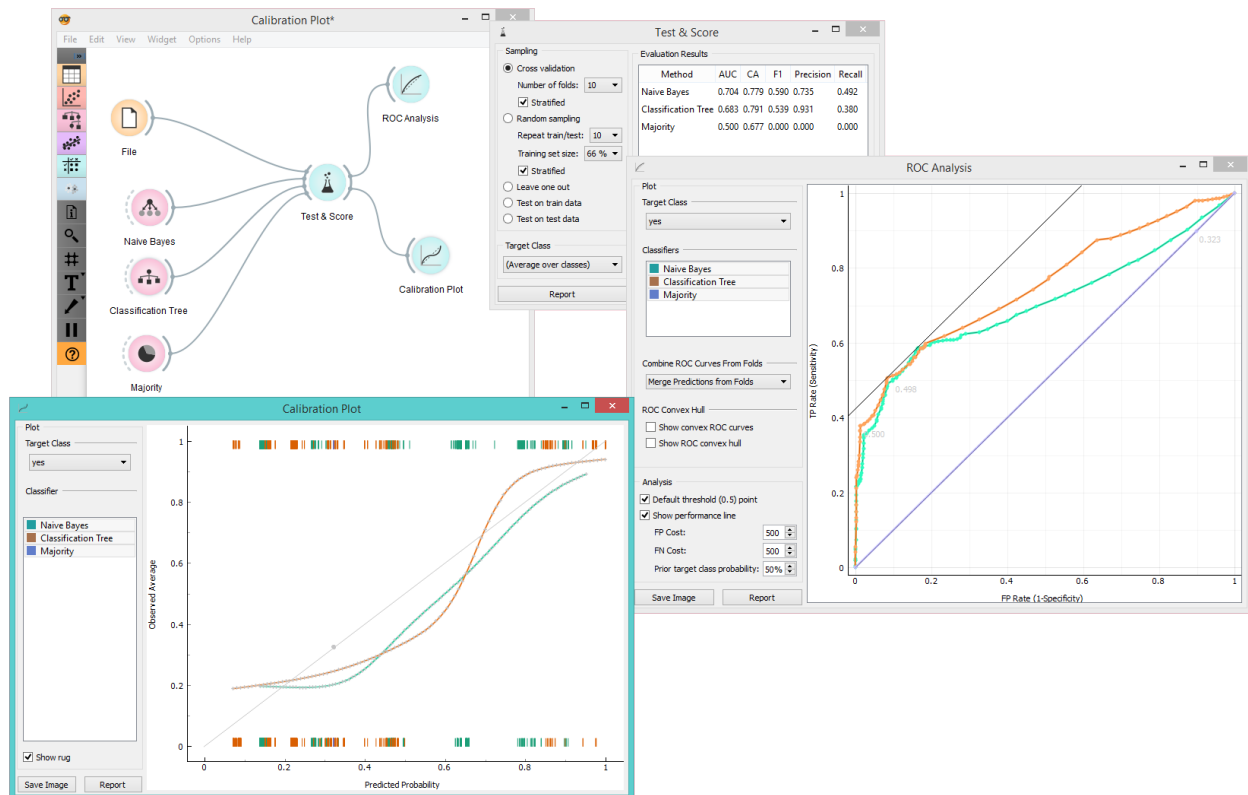
The **Calibration Plot** plots class probabilities against those predicted by the classifier(s).

1. Select the desired target class from the drop down menu.
2. Choose which classifiers to plot. The diagonal represents optimal behaviour; the closer the classifier's curve gets, the more accurate its prediction probabilities are. Thus we would use this widget to see whether a classifier is overly optimistic (gives predominantly positive results) or pesimistic (gives predominantly negative results).
3. If *Show rug* is enabled, ticks are displayed at the bottom and the top of the graph, which represent negative and positive examples respectively. Their position corresponds to the classifier's probability prediction and the color shows the classifier. At the bottom of the graph, the points to the left are those which are (correctly) assigned a low probability of the target class, and those to the right are incorrectly assigned high probabilities. At the top of the graph, the instances to the right are correctly assigned high probabilities and vice versa.
4. Press *Save Image* if you want to save the created image to your computer in a .svg or .png format.
5. Produce a report.

## Example

At the moment, the only widget which gives the right type of signal needed by the **Calibration Plot** is **Test&Score**. The Calibration Plot will hence always follow Test&Score and, since it has no outputs, no other widgets follow it.

Here is a typical example, where we compare three classifiers (namely *Naive Bayes*, *Classification Tree* and *Majority*) and input them into **Test&Score**. We used the *Titanic* data set. Test&Score then displays evaluation results for each classifier. Then we draw **Calibration Plot** and **ROC Analysis** widgets from Test&Score to further analyze the performance of classifiers. **Calibration Plot** enables you to see prediction accuracy of class probabilities in a plot.





## Confusion Matrix



Shows proportions between the predicted and actual class.

### Signals

#### Inputs:

- **Evaluation results**

Results of testing the algorithms; typically from **Test Learners**

#### Outputs:

- **Selected Data**

A data subset from the selected cells in the confusion matrix.

### Description

The **Confusion Matrix** gives the number/proportion of instances between the predicted and actual class. The selection of the elements in the matrix feeds the corresponding instances into the output signal. This way, one can observe which specific instances were misclassified and how.

The widget usually gets the evaluation results from *Test Learners*; an example of the schema is shown below.

1. When evaluation results contain data on multiple learning algorithms, we have to choose one in the *Learners* box.

The snapshot shows the confusion matrix for *Classification Tree* and *Naive Bayesian* classifier trained and tested on the *Iris* data. The righthand side of the widget contains the matrix for the naive Bayesian classifier (since this classifier is selected on the left). Each row corresponds to a correct class, while columns represent the predicted classes. For instance, four instances of *Iris-versicolor* were misclassified as *Iris-virginica*. The rightmost column gives the number of instances from each class (there are 50 irises of each of the three classes) and the bottom row gives the number of instances classified into each class (e.g., 48 instances were classified into virginica).

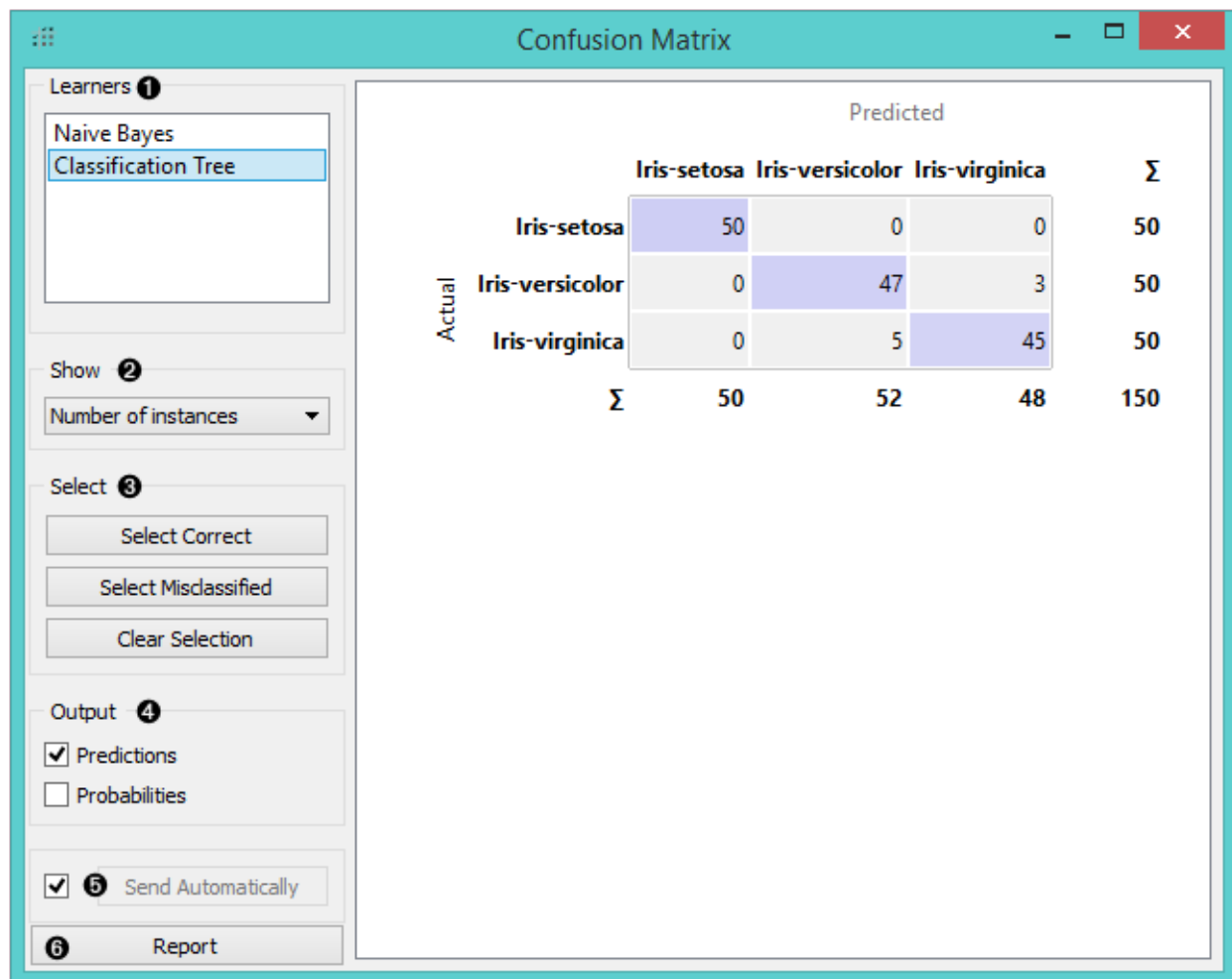
2. In *Show*, we select what data we would like to see in the matrix.

- **Number of instances** shows correctly and incorrectly classified instances numerically.
- **Proportions of predicted** shows how many instances classified as, say, *Iris-versicolor* are in which true class; in the table we can read the 0% of them are actually setosae, 88.5% of those classified as versicolor are versicolors, and 7.7% are virginicae.
- **Proportions of actual** shows the opposite relation: of all true versicolors, 92% were classified as versicolors and 8% as virginicae.

3. In *Select*, you can choose the desired output.

- **Correct** sends all correctly classified instances to the output by selecting the diagonal of the matrix.
- **Misclassified** selects the misclassified instances.
- **None** annuls the selection.

As mentioned before, one can also select individual cells of the table to select specific kinds of misclassified instances (e.g. the versicolors classified as virginicae).

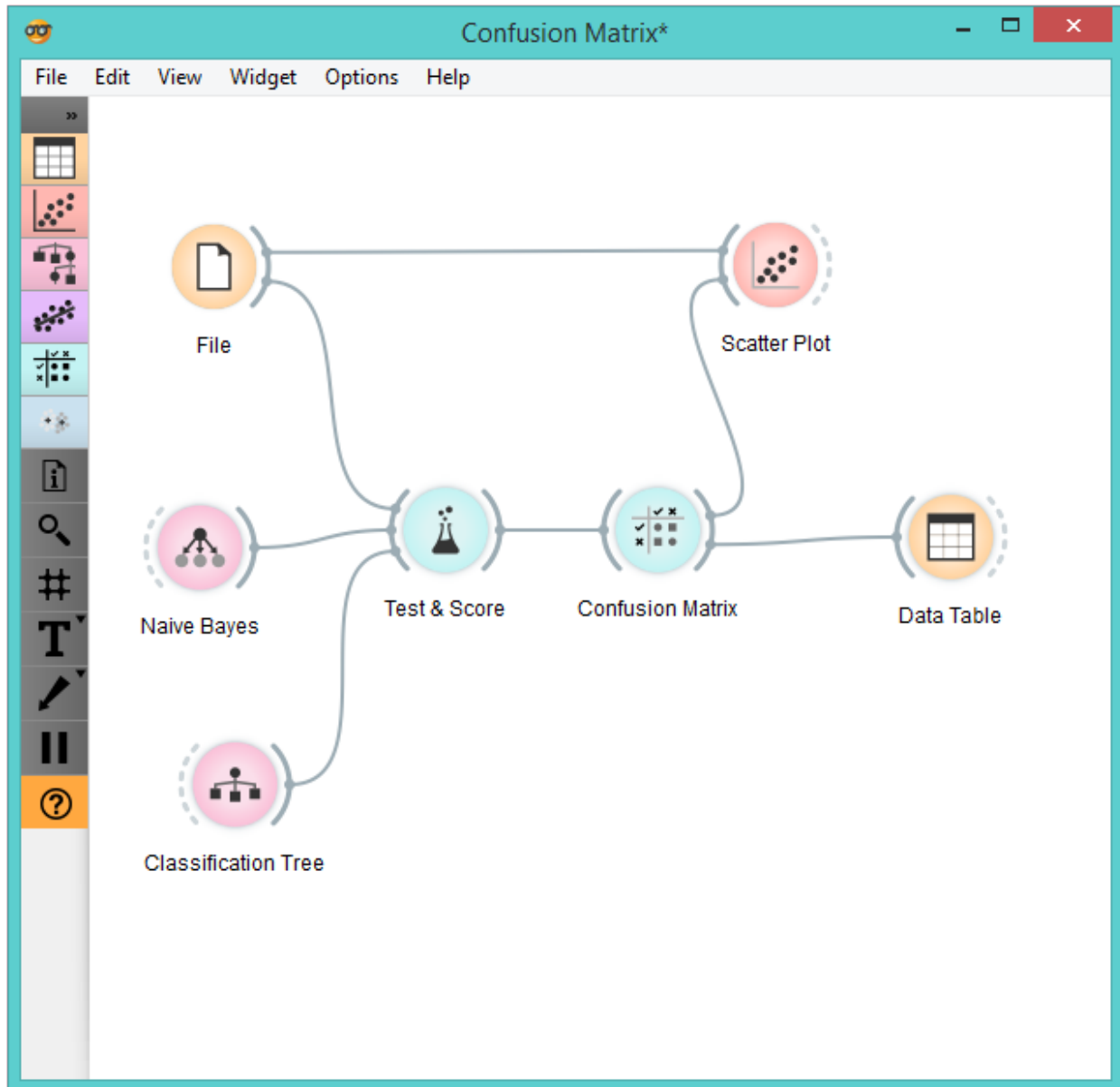


		Predicted			$\Sigma$
		Iris-setosa	Iris-versicolor	Iris-virginica	
Actual	Iris-setosa	100.0 %	0.0 %	0.0 %	50
	Iris-versicolor	0.0 %	88.7 %	6.4 %	50
	Iris-virginica	0.0 %	11.3 %	93.6 %	50
$\Sigma$		50	53	47	150

4. When sending selected instances, the widget can add new attributes, such as predicted classes or their probabilities, if the corresponding options *Predictions* and/or *Probabilities* are checked.
5. The widget outputs every change if *Send Automatically* is ticked. If not, the user will need to click *Send Selected* to commit the changes.
6. Produce a report.

## Example

The following schema demonstrates well what this widget can be used for.

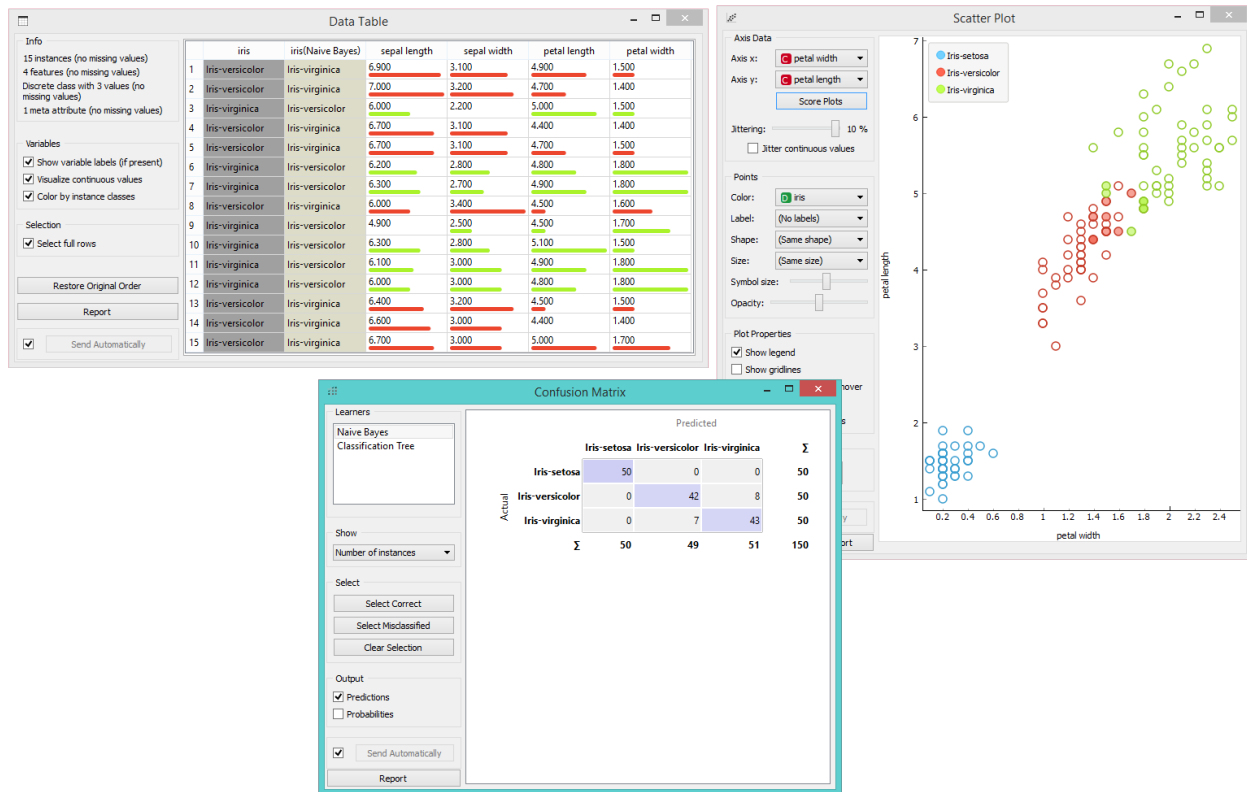


*Test&Score* gets the data from *File* and two learning algorithms from *Naive Bayes* and *Classification Tree*. It performs cross-validation or some other train-and-test procedures to get class predictions by both algorithms for all (or some)

data instances. The test results are fed into the **Confusion Matrix**, where we can observe how many instances were misclassified and in which way.

In the output, we used *Data Table* to show the instances we selected in the confusion matrix. If we, for instance, click *Misclassified*, the table will contain all instances which were misclassified by the selected method.

The *Scatterplot* gets two sets of data. From the *File* widget it gets the complete data, while the confusion matrix sends only the selected data, misclassifications for instance. The scatter plot will show all the data, with bold symbols representing the selected data.



## Lift Curve



Measures the performance of a chosen classifier against a random classifier.

## Signals

### Inputs:

- **Evaluation Results**

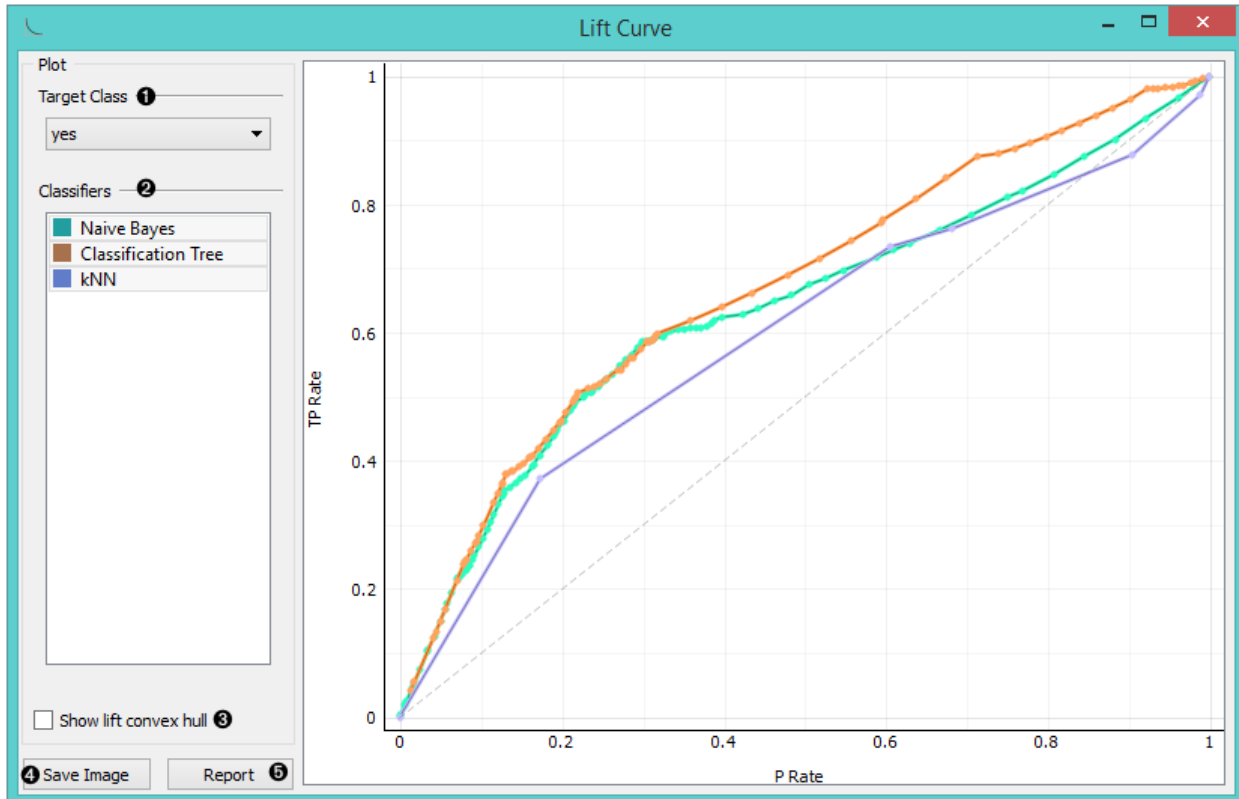
Results of classifiers' tests on data.

### Outputs:

- None

## Description

The **Lift curve** shows the relation between the number of instances which were predicted positive and those that are indeed positive and thus measures the performance of a chosen classifier against a random classifier. The graph is constructed with the cumulative number of cases (in descending order of probability) on the x-axis and the cumulative number of true positives on the y-axis. Lift curve is often used in segmenting the population, e.g., plotting the number of responding customers against the number of all customers contacted. You can also determine the optimal classifier and its threshold from the graph.



1. Choose the desired *Target class*. The default class is chosen alphabetically.
2. If test results contain more than one classifier, the user can choose which curves she or he wants to see plotted. Click on a classifier to select or deselect the curve.
3. *Show lift convex hull* plots a convex hull over lift curves for all classifiers (yellow curve). The curve shows the optimal classifier (or combination thereof) for each desired TP/P rate.
4. Press *Save Image* if you want to save the created image to your computer in a .svg or .png format.
5. Produce a report.
6. 2-D pane with **P rate** (population) as x-axis and **TP rate** (true positives) as a y-axis. The diagonal line represents the behaviour of a random classifier. Click and drag to move the pane and scroll in or out to zoom. Click on the “A” sign at the bottom left corner to realign the pane.

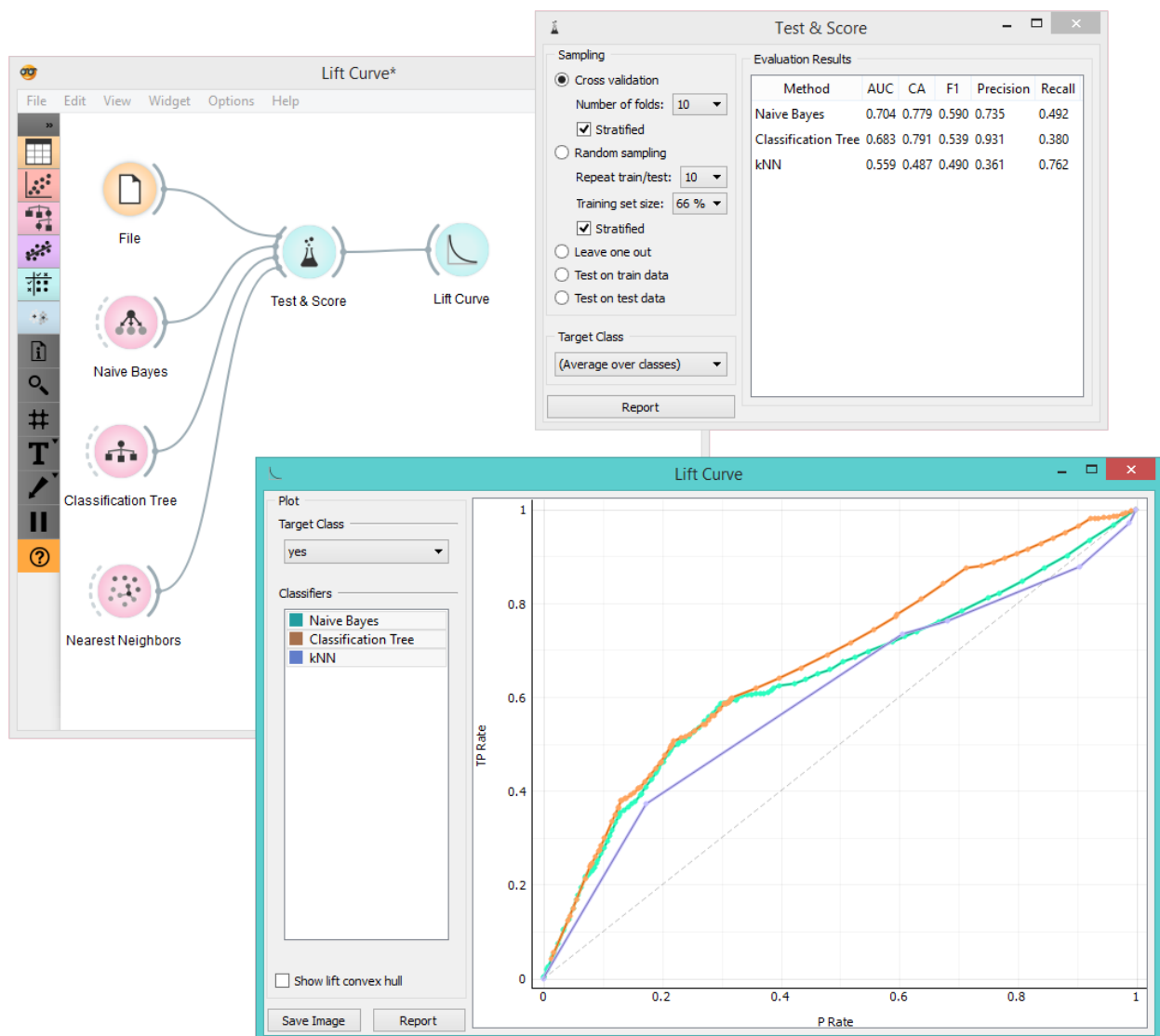
**Note:** The perfect classifier would have a steep slope towards 1 until all classes are guessed correctly and then run

straight along 1 on y-axis to (1,1).

## Example

At the moment, the only widget which gives the right type of the signal needed by the **Lift Curve** is *Test & Score*.

In the example below, we try to see the prediction quality for the class ‘survived’ on the *Titanic* data set. We compared three different classifiers in the Test Learners widget and sent them to Lift Curve to see their performance against a random model. We see the *Classification Tree* classifier is the best out of the three, since it best aligns with *lift convex hull*. We also see that its performance is the best for the first 30% of the population (in order of descending probability), which we can set as the threshold for optimal classification.



## References

Handouts of the University of Notre Dame on Data Mining - Lift Curve. Available [here](#).

## Predictions



Shows classifiers' predictions on the data.

## Signals

### Inputs

- **Data**  
A data set.
- **Predictors**  
Predictors to be used on the data.

### Outputs

- **Predictions**  
Original data with added predictions.

## Description

The widget receives a data set and one or more predictors (classifiers, not learning algorithms - see the example below). It outputs the data and the predictions.

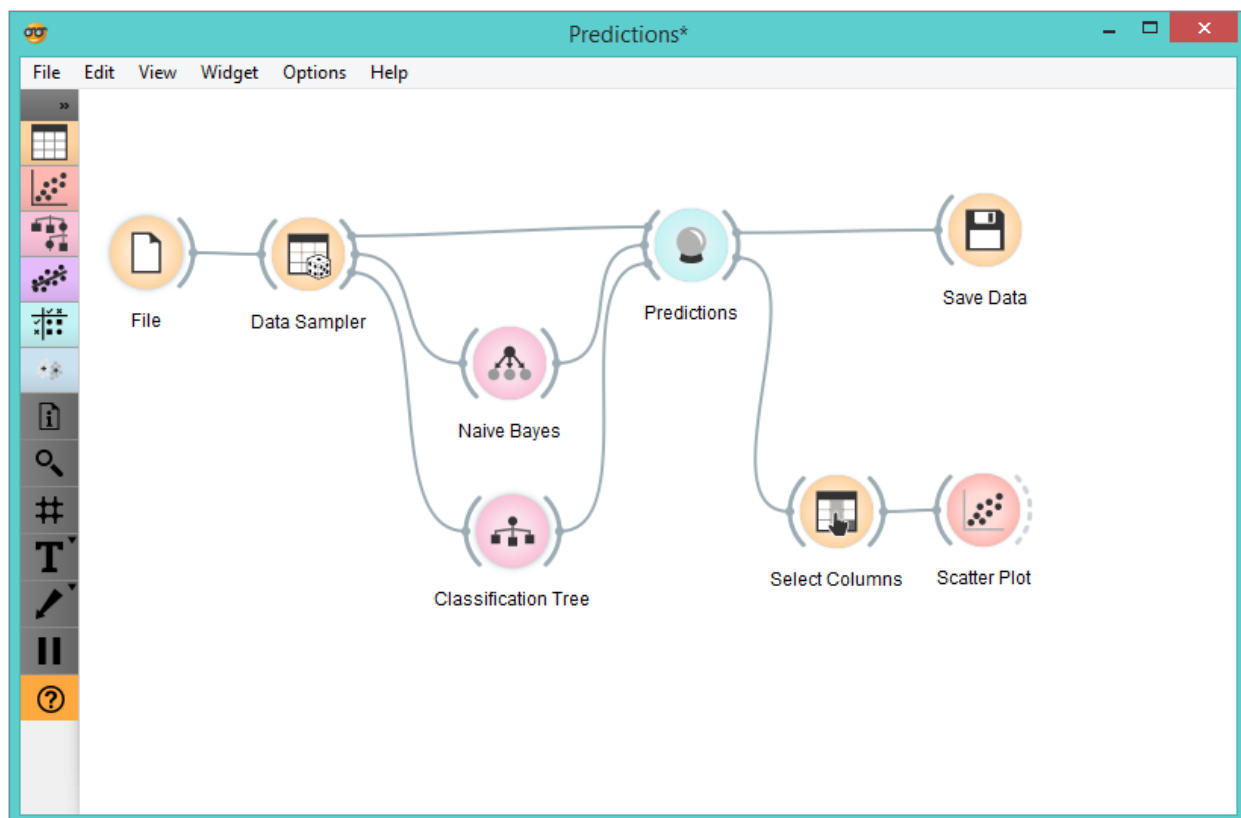
	Classification Tree	Naive Bayes	iris	sepal length	sepal width	petal length	petal width
1	Iris-setosa	Iris-setosa	Iris-setosa	4.600	3.600	1.000	0.200
2	Iris-versicolor	Iris-versicolor	Iris-versicolor	4.900	2.400	3.300	1.000
3	Iris-versicolor	Iris-versicolor	Iris-versicolor	5.700	2.800	4.100	1.300
4	Iris-setosa	Iris-setosa	Iris-setosa	5.400	3.700	1.500	0.200
5	Iris-versicolor	Iris-versicolor	Iris-versicolor	6.600	3.000	4.400	1.400
6	Iris-versicolor	Iris-versicolor	Iris-virginica	4.900	2.500	4.500	1.700
7	Iris-versicolor	Iris-versicolor	Iris-versicolor	5.500	2.500	4.000	1.300
8	Iris-virginica	Iris-virginica	Iris-virginica	5.700	2.500	5.000	2.000
9	Iris-setosa	Iris-setosa	Iris-setosa	4.600	3.100	1.500	0.200
10	Iris-versicolor	Iris-versicolor	Iris-versicolor	5.600	3.000	4.100	1.300
11	Iris-setosa	Iris-setosa	Iris-setosa	4.800	3.000	1.400	0.300
12	Iris-virginica	Iris-virginica	Iris-virginica	6.700	3.300	5.700	2.100
13	Iris-virginica	Iris-virginica	Iris-virginica	7.600	3.000	6.600	2.100
14	Iris-setosa	Iris-setosa	Iris-setosa	4.500	2.300	1.300	0.300
15	Iris-setosa	Iris-setosa	Iris-setosa	5.000	3.300	1.400	0.200
16	Iris-versicolor	Iris-virginica	Iris-virginica	7.200	3.000	5.800	1.600
17	Iris-virginica	Iris-versicolor	Iris-virginica	6.300	2.700	4.900	1.800
18	Iris-virginica	Iris-virginica	Iris-virginica	5.800	2.700	5.100	1.900
19	Iris-setosa	Iris-setosa	Iris-setosa	5.100	3.300	1.700	0.500
20	Iris-virginica	Iris-virginica	Iris-virginica	6.700	3.100	5.600	2.400

1. Information on the input

2. The user can select the options for classification. If *Show predicted class* is ticked, the appended data table provides information on predicted class. If *Show predicted probabilities* is ticked, the appended data table provides information on probabilities predicted by the classifiers. The user can also select the predicted class he or she wants displayed in the appended data table. The option *Draw distribution bars* provides a nice visualization of the predictions.
3. By ticking the *Show full data set*, the user can append the entire data table to the *Predictions* widget.
4. Select the desired output.
5. The appended data table
6. Produce a report.

Despite its simplicity, the widget allows for quite an interesting analysis of decisions of [predictive models](#); there is a simple demonstration at the bottom of the page. [Confusion Matrix](#) is a related widget and although many things can be done with any of them, there are tasks for which one of them might be much more convenient than the other. The output of the widget is another data set, where predictions are appended as new meta attributes. You can select which features you wish to output (original data, predictions, probabilities). The resulting data set can be appended to the widget, but you can still choose to display it in a separate data table.

## Example

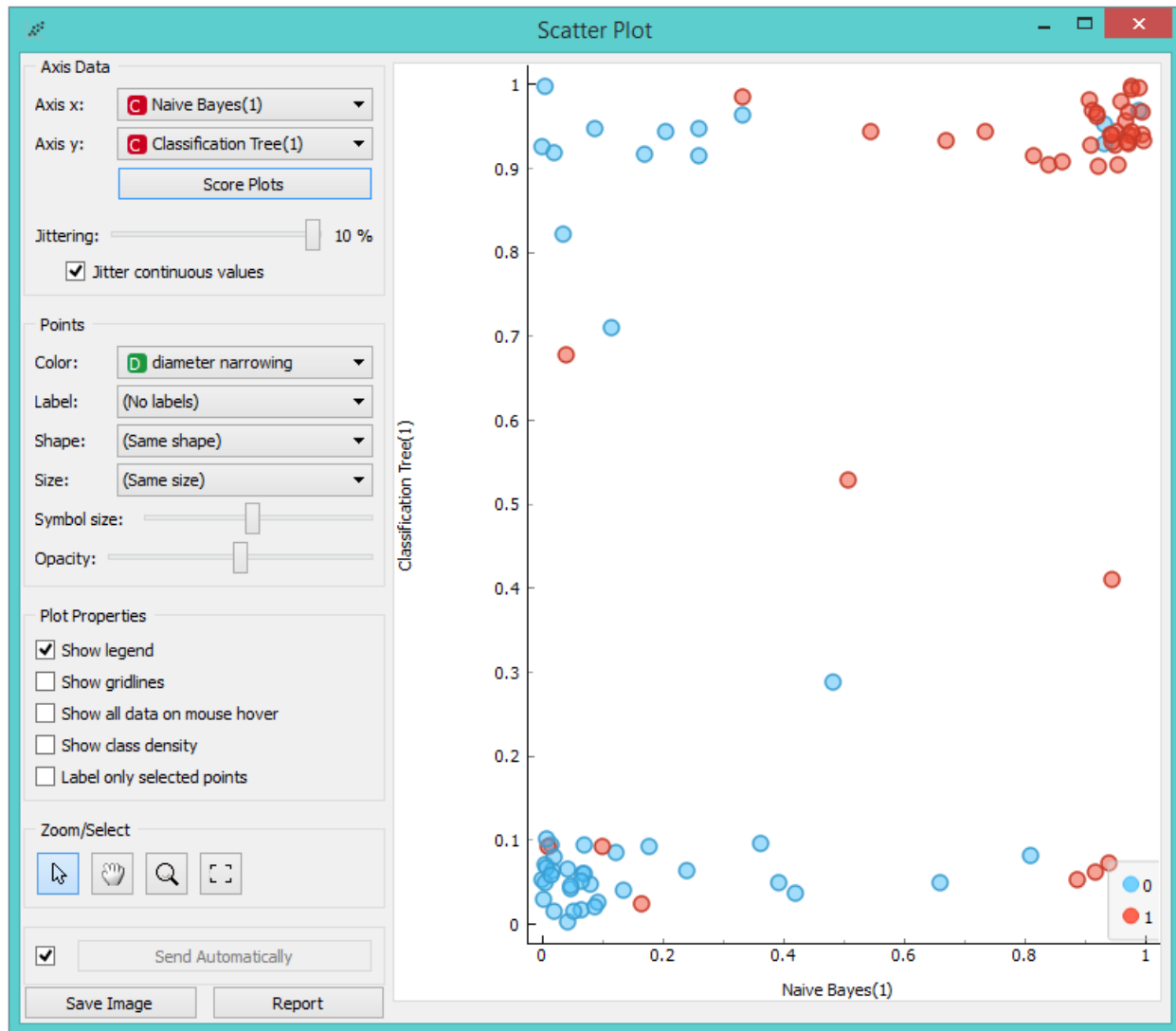


We randomly split the data into two subsets. The larger subset, containing 70 % of data instances, is sent to [Naive Bayes](#) and [Classification Tree](#), so they can produce the corresponding classifiers. Classifiers are then sent into **Predictions**, among with the remaining 30 % of the data. Predictions shows how these examples are classified.

To save the predictions, we simply attach the [Save](#) widget to **Predictions**. The final file is a data table and can be saved as in a .tab or .tsv format.



Finally, we can analyze the classifiers' predictions. For that, we first take *Select Columns* with which we move the meta attributes with probability predictions to features. The transformed data is then given to the *Scatterplot*, which we set to use the attributes with probabilities as the x and y axes, while the class is (already by default) used to color the data points.



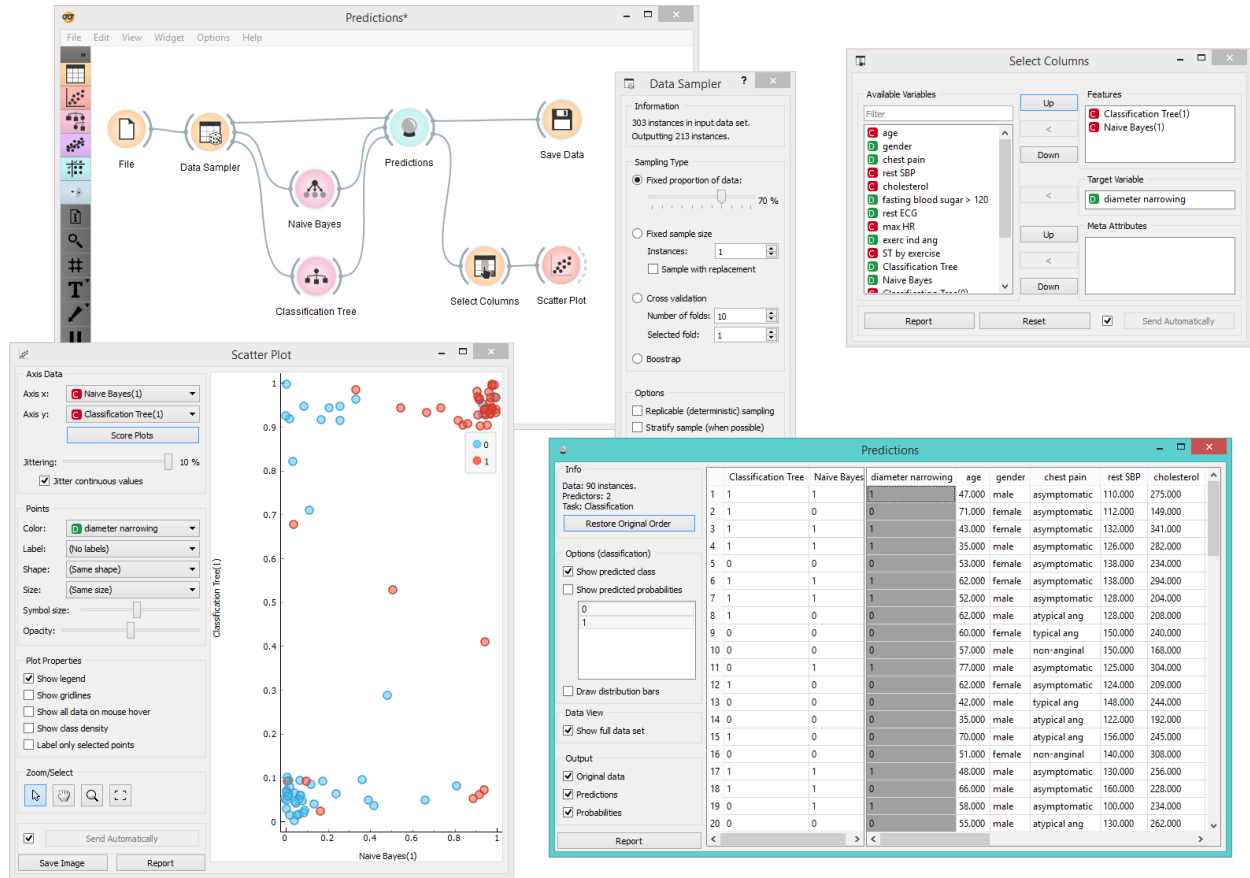
To get the above plot, we selected *Jitter continuous values*, since the classification tree gives just a few distinct probabilities. The blue points in the bottom left corner represent the people with no diameter narrowing, which were correctly classified by both classifiers. The upper right red points represent the patients with narrowed vessels, which were correctly classified by both.

Note that this analysis is done on a rather small sample, so these conclusions may be ungrounded. Here is the entire workflow:

Another example of using this widget is given in the documentation for the widget *Confusion Matrix*.

## ROC Analysis

Plots a true positive rate against a false positive rate of a test.



## Signals

### Inputs:

- **Evaluation Results**

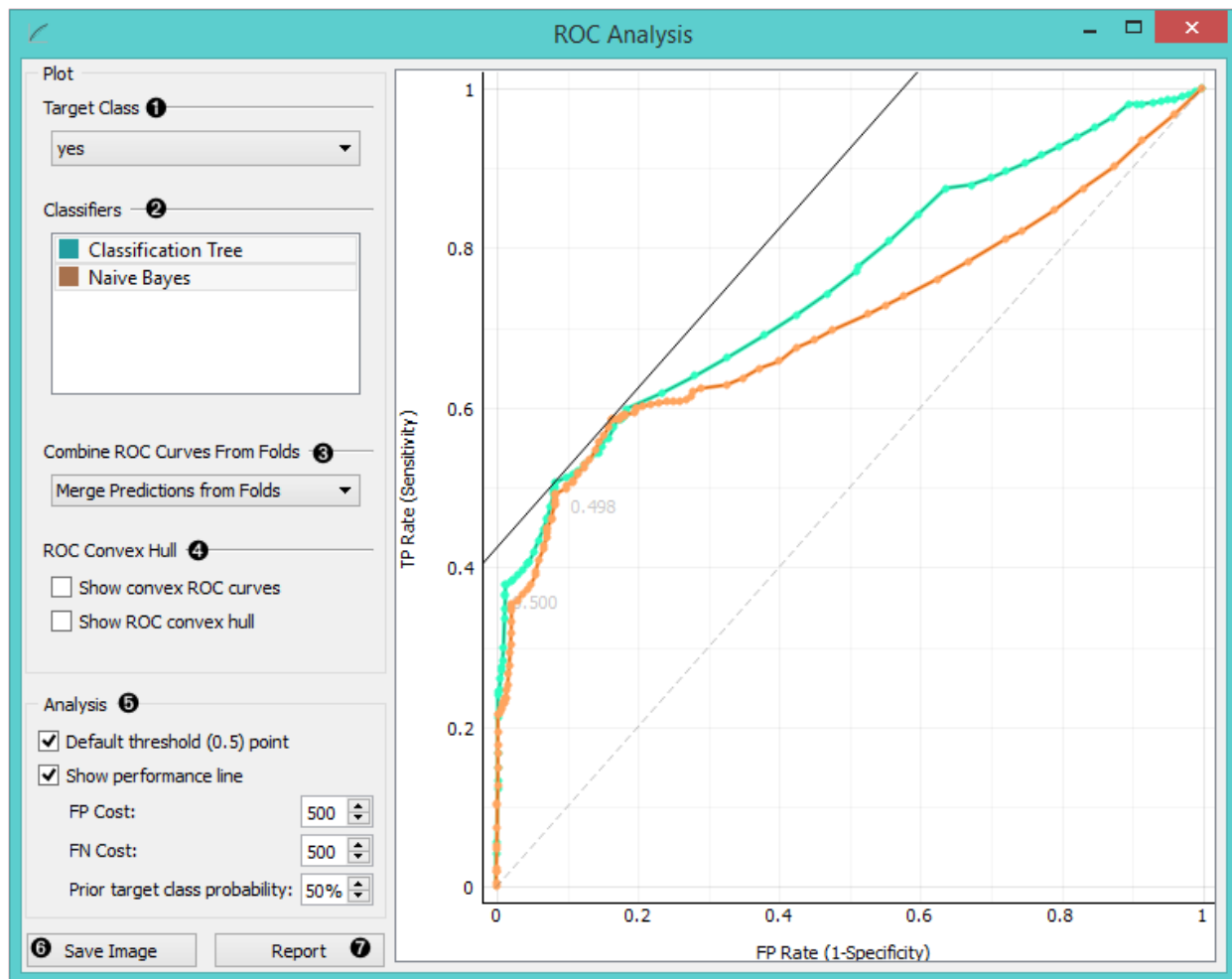
Results of classifiers' tests on data

### Outputs:

- None

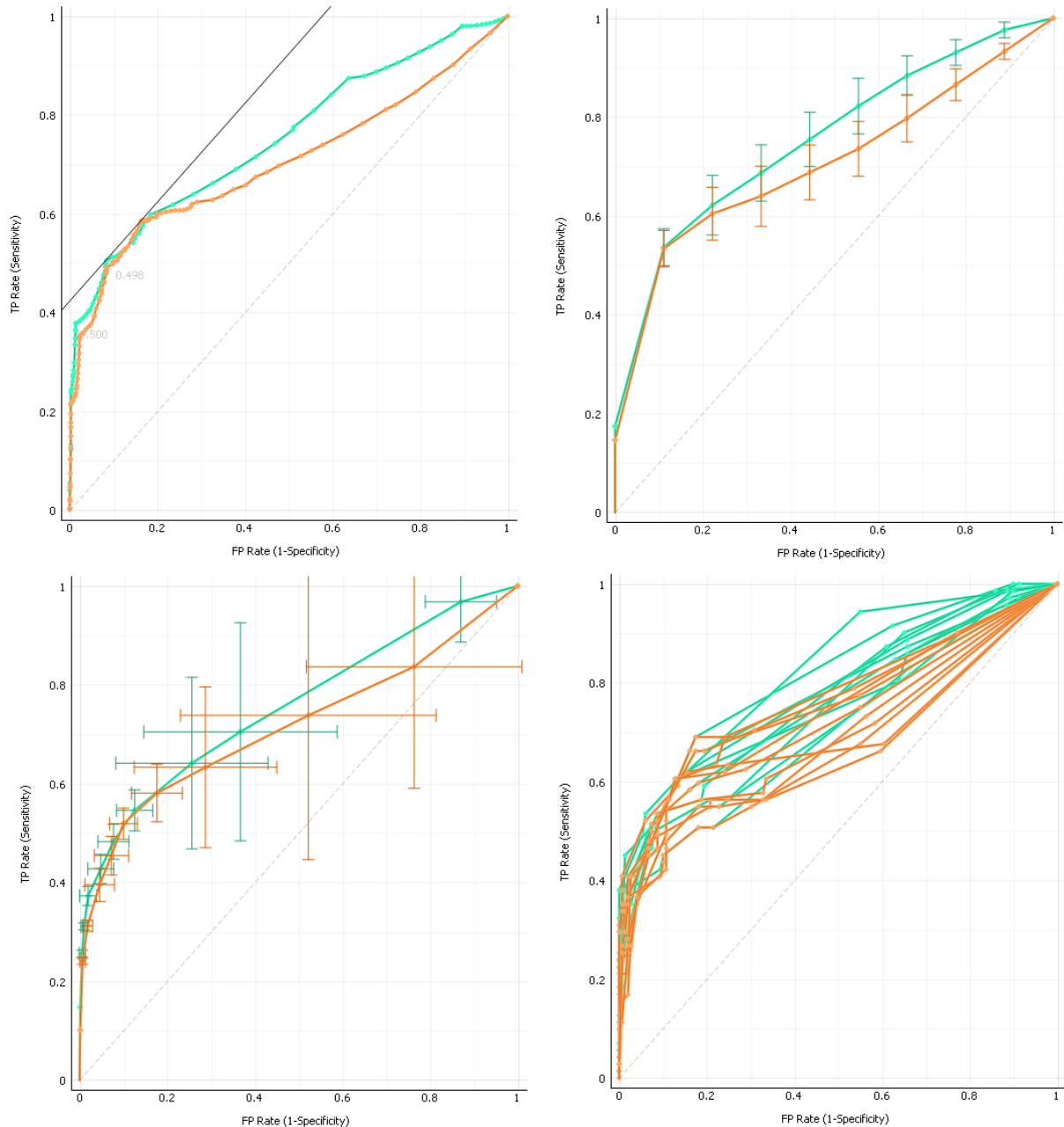
## Description

The widget shows ROC curves for the tested models and the corresponding convex hull. It serves as a mean of comparison between classification models. The curve plots a false positive rate on an x-axis (1-specificity; probability that target=1 when true value=0) against a true positive rate on a y-axis (sensitivity; probability that target=1 when true value=1). The closer the curve follows the left-hand border and then the top border of the ROC space, the more accurate the classifier. Given the costs of false positives and false negatives, the widget can also determine the optimal classifier and threshold.



1. Choose the desired *Target Class*. The default class is chosen alphabetically.

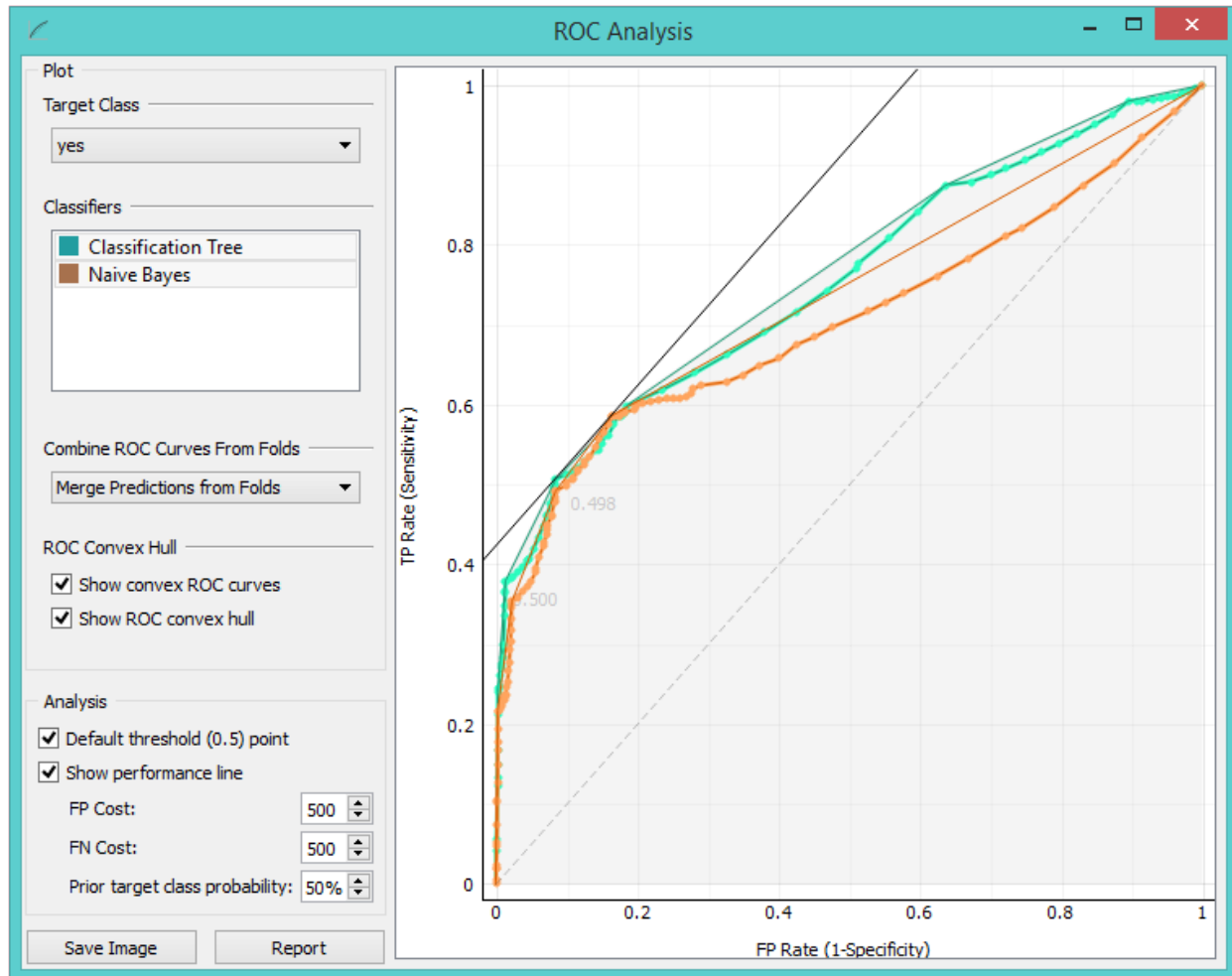
2. If test results contain more than one classifier, the user can choose which curves she or he wants to see plotted. Click on a classifier to select or deselect it.
3. When the data comes from multiple iterations of training and testing, such as k-fold cross validation, the results can be (and usually are) averaged.



The averaging options are:

- **Merge predictions from folds** (top left), which treats all the test data as if they came from a single iteration
- **Mean TP rate** (top right) averages the curves vertically, showing the corresponding confidence intervals
- **Mean TP and FP at threshold** (bottom left) traverses over threshold, averages the positions of curves and shows horizontal and vertical confidence intervals

- **Show individual curves** (bottom right) does not average but prints all the curves instead
4. Option *Show convex ROC curves* refers to convex curves over each individual classifier (the thin lines positioned over curves). *Show ROC convex hull* plots a convex hull combining all classifiers (the gray area below the curves). Plotting both types of convex curves makes sense since selecting a threshold in a concave part of the curve cannot yield optimal results, disregarding the cost matrix. Besides, it is possible to reach any point on the convex curve by combining the classifiers represented by the points on the border of the concave region.



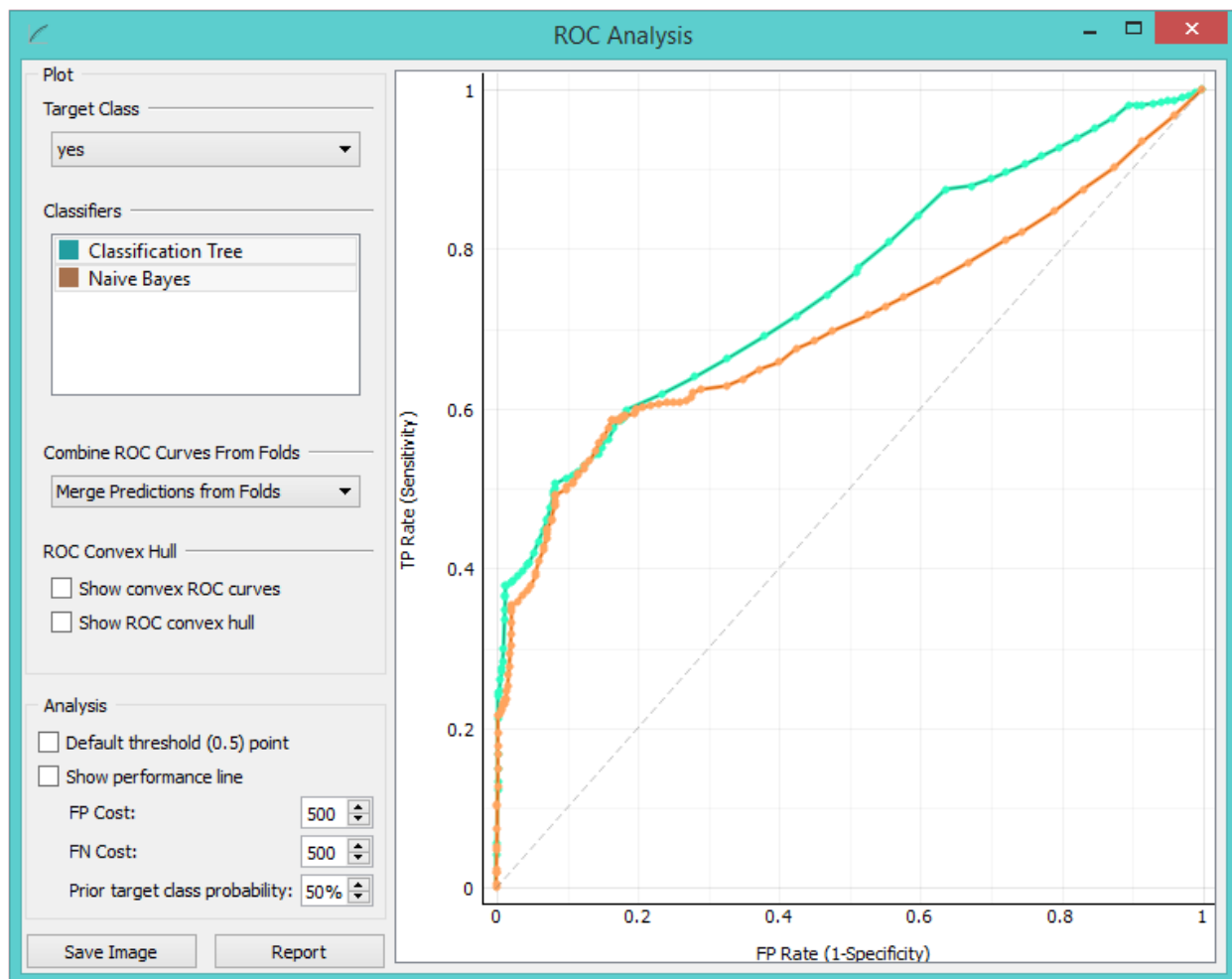
The diagonal dotted line represents the behaviour of a random classifier. The full diagonal line represents iso-performance. A black “A” symbol at the bottom of the graph proportionally readjusts the graph.

5. The final box is dedicated to the analysis of the curve. The user can specify the cost of false positives (FP) and false negatives (FN), and the prior target class probability.

*Default threshold (0.5) point* shows the point on the ROC curve achieved by the classifier if it predicts the target class if its probability equals or exceeds 0.5.

*Show performance line* shows iso-performance in the ROC space so that all the points on the line give the same profit/loss. The line further to the upper left is better than the one down and right. The direction of the line depends upon costs and probabilities. This gives a recipe for depicting the optimal threshold for the given costs: this is the point where the tangent with the given inclination touches the curve and it is marked in the plot. If we push the iso-performance higher or more to the left, the points on the iso-performance line cannot be reached by the learner. Going down or to the right, decreases the performance.

The widget allows setting the costs from 1 to 1000. Units are not important, as are not the magnitudes. What matters is the relation between the two costs, so setting them to 100 and 200 will give the same result as 400 and 800.



Defaults: both costs equal (500), Prior target class probability 50% (from the data).

False positive cost: 830, False negative cost 650, Prior target class probability 73%.

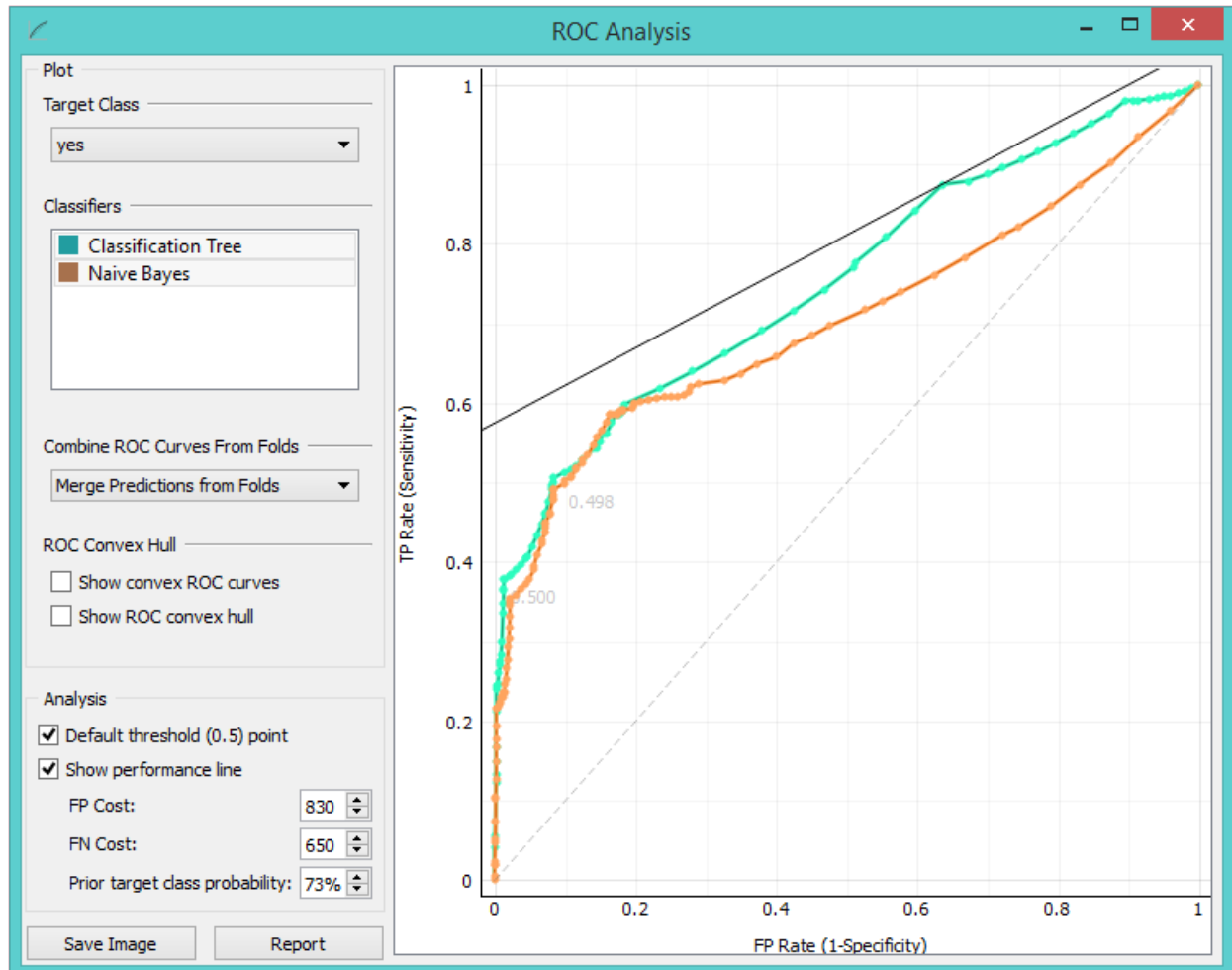
6. Press *Save Image* if you want to save the created image to your computer in a .svg or .png format.
7. Produce a report.

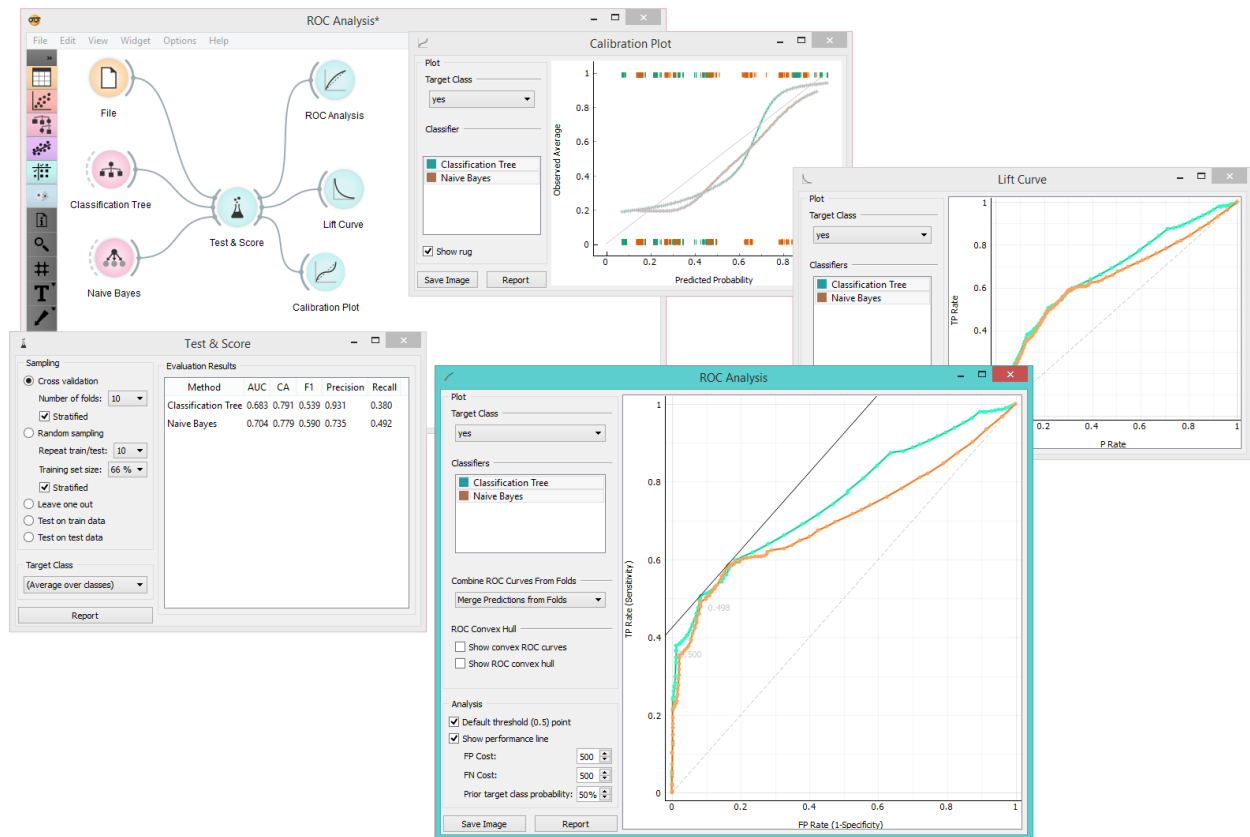
## Example

At the moment, the only widget which gives the right type of signal needed by the **ROC Analysis** is *Test&Score*. Below, we compare two classifiers, namely *Classification Tree* and *Naive Bayes*, in *Test&Score* and then compare their performance in **ROC Analysis**, *Life Curve* and *Calibration Plot*.

## Test & Score

Tests learning algorithms on data.







## Signals

### Inputs

- **Data**

Data for training and, if there is no separate test data set, also testing.

- **Test Data**

Separate data for testing.

- **Learner**

One or more learning algorithms.

### Outputs

- **Evaluation results**

Results of testing the algorithms.

## Description

The widget tests learning algorithms. Different sampling schemes are available, including using separate test data. The widget does two things. First, it shows a table with different classifier performance measures, such as [classification accuracy](#) and [area under ROC](#). Second, it outputs evaluation results, which can be used by other widgets for analyzing the performance of classifiers, such as [ROC Analysis](#) or [Confusion Matrix](#).

The *Learner* signal has an uncommon property: it can be connected to more than one widget to test multiple learners with the same procedures.

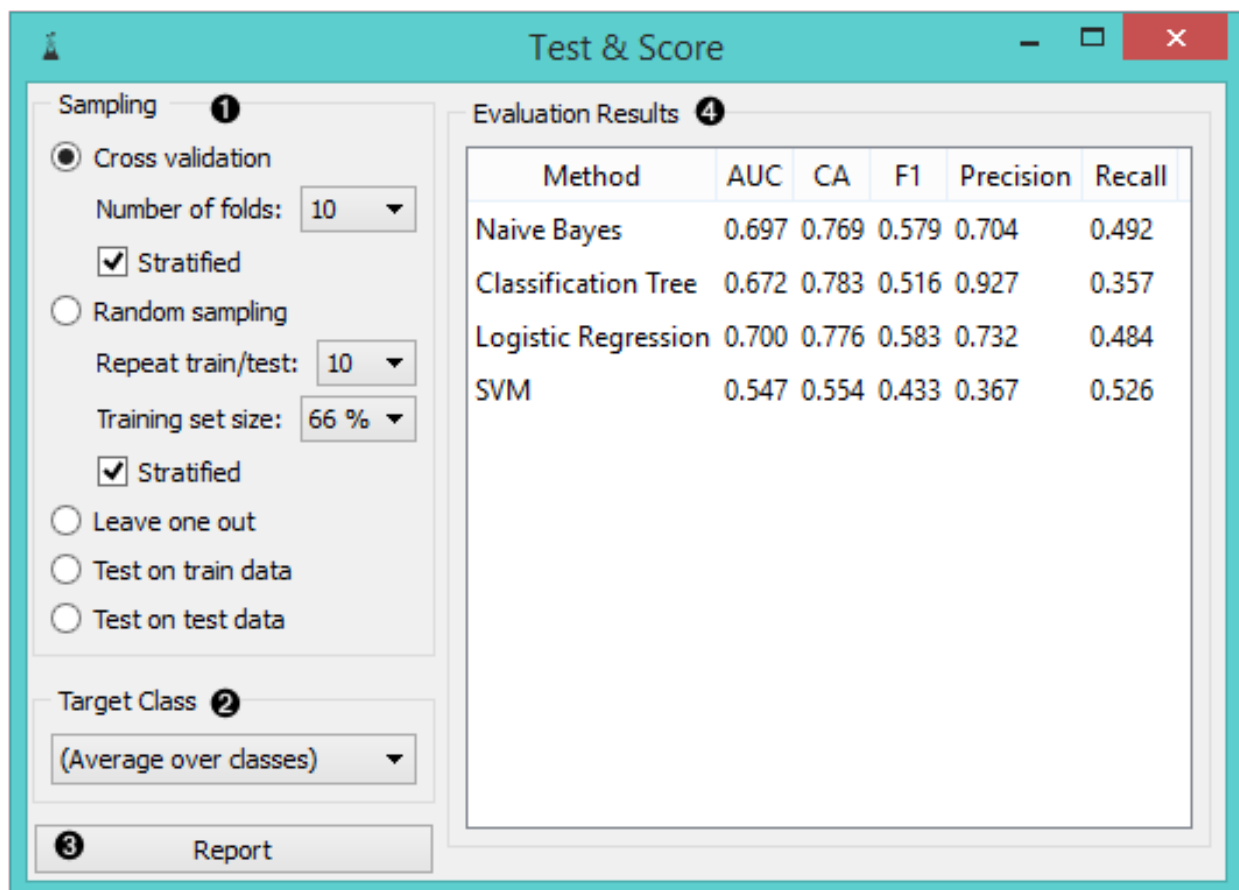
1. The widget supports various sampling methods.

- **Cross-validation** splits the data into a given number of folds (usually 5 or 10). The algorithm is tested by holding out examples from one fold at a time; the model is induced from other folds and examples from the held out fold are classified. This is repeated for all the folds.
- **Leave-one-out** is similar, but it holds out one instance at a time, inducing the model from all others and then classifying the held out instances. This method is obviously very stable, reliable ... and very slow.
- **Random sampling** randomly splits the data into the training and testing set in the given proportion (e.g. 70:30); the whole procedure is repeated for a specified number of times.
- **Test on train data** uses the whole data set for training and then for testing. This method practically always gives wrong results.
- **Test on test data:** the above methods use the data from *Data* signal only. To input another data set with testing examples (for instance from another file or some data selected in another widget), we select *Separate Test Data* signal in the communication channel and select Test on test data.

2. Only *Test on test data* requires a target class, e.g. having the disease or being of subvariety *Iris setosa*. When *Target class* is (None), the methods return the average value. Target class can be selected at the bottom of the widget.

3. Produce a report.

4. The widget will compute a number of performance statistics:



**Test & Score**

**Sampling**

☒ Cross validation  
Number of folds: 10  
☒ Stratified

☐ Random sampling  
Repeat train/test: 10  
Training set size: 66 %  
☒ Stratified

☐ Leave one out  
☐ Test on train data  
☐ Test on test data

**Target Class**  
(Average over classes)

**Report**

**Evaluation Results**

Method	AUC	CA	F1	Precision	Recall
Naive Bayes	0.697	0.769	0.579	0.704	0.492
Classification Tree	0.672	0.783	0.516	0.927	0.357
Logistic Regression	0.700	0.776	0.583	0.732	0.484
SVM	0.547	0.554	0.433	0.367	0.526

## Classification

- **Area under ROC** is the area under the receiver-operating curve.
- **Classification accuracy** is the proportion of correctly classified examples.
- **F-1** is a weighted harmonic mean of precision and recall (see below).
- **Precision** is the proportion of true positives among instances classified as positive, e.g. the proportion of *Iris virginica* correctly identified as *Iris virginica*.
- **Recall** is the proportion of true positives among all positive instances in the data, e.g. the number of sick among all diagnosed as sick.

## Regression

**Test & Score**

**Sampling**

- ☒ Cross validation
  - Number of folds: 10
  - ☒ Stratified
- ☐ Random sampling
  - Repeat train/test: 10
  - Training set size: 66 %
  - ☒ Stratified
- ☐ Leave one out
- ☐ Test on train data
- ☐ Test on test data

**Report**

**Evaluation Results**

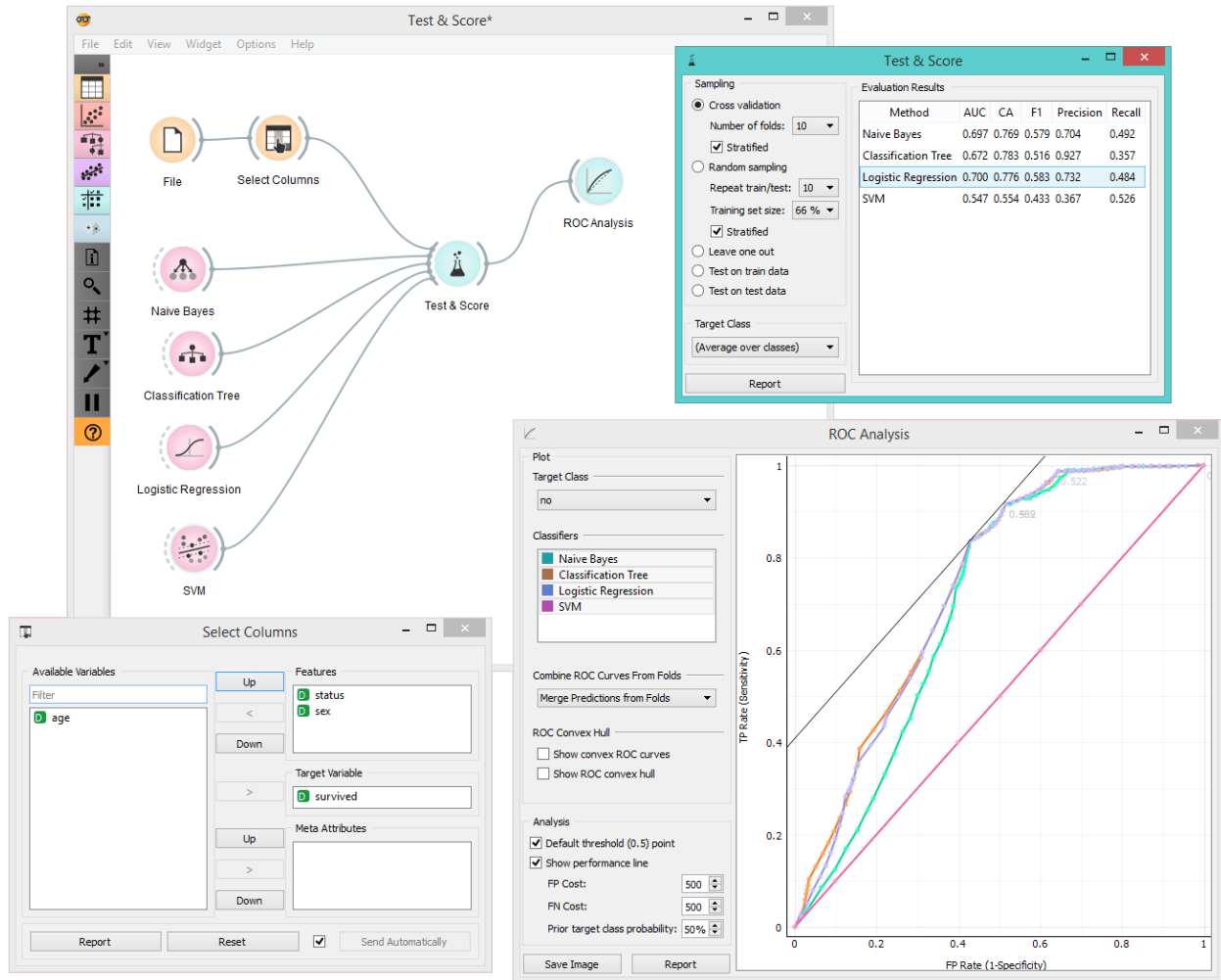
Method	MSE	RMSE	MAE	R2
Mean Learner	84.644	9.200	6.662	-0.003
Nearest Neighbors	38.676	6.219	4.352	0.542
SVM Regression	66.314	8.143	5.141	0.214
SGD Regression	24.297	4.929	3.296	0.712

- **MSE** measures the average of the squares of the errors or deviations (the difference between the estimator and what is estimated).
- **RMSE** is the square root of the arithmetic mean of the squares of a set of numbers (a measure of imperfection of the fit of the estimator to the data)
- **MAE** is used to measure how close forecasts or predictions are to eventual outcomes.

- $R^2$  is interpreted as the proportion of the variance in the dependent variable that is predictable from the independent variable.

## Example

In a typical use of the widget, we give it a data set and a few learning algorithms and we observe their performance in the table inside the *Test&Score* widget and in the *ROC*. The data is often preprocessed before testing; in this case we did some manual feature selection (*Select Columns* widget) on *Titanic* data set, where we want to know only the sex and status of the survived and omit the age.



Another example of using this widget is presented in the documentation for the *Confusion Matrix* widget.