Orange Visual Programming Documentation

Release 3

Orange Data Mining

Mar 22, 2019

Contents

CHAPTER 1

Getting Started

Here we need to copy the getting started guide.

1.1 Loading your Data

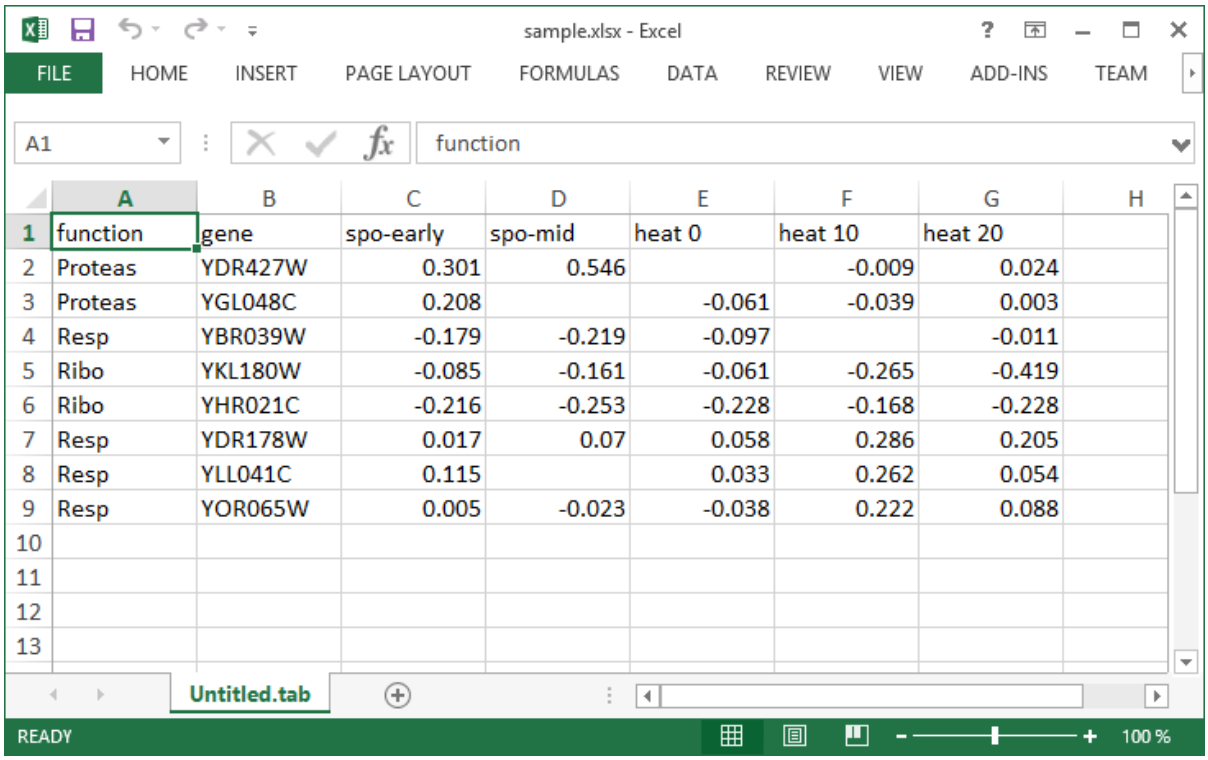
Orange comes with its own data format, but can also handle native Excel (.xlsx or .xls), comma- or tab-delimited data files. The input data set is usually a table, with data instances (samples) in rows and data attributes in columns. Attributes can be of different *types* (continuous, discrete, time, 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.

1.1.1 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 proving information on the attribute role (**class**, **meta**, **weight** or **ignore**).

1.1.2 Example: Data from Excel

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

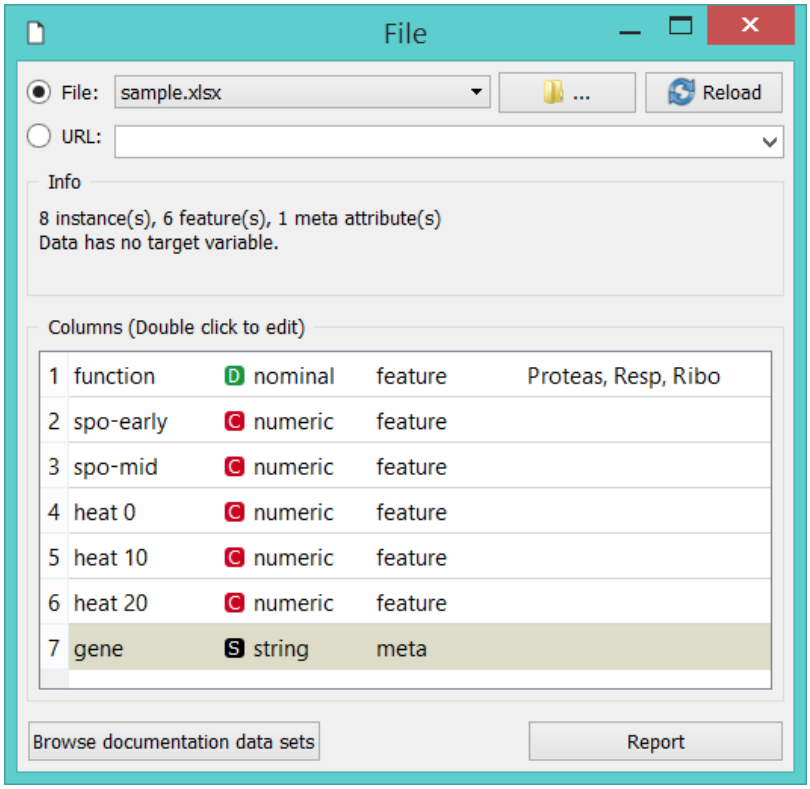


	A	B	C	D	E	F	G	H
1	function	gene	spo-early	spo-mid	heat 0	heat 10	heat 20	
2	Proteas	YDR427W	0.301	0.546		-0.009	0.024	
3	Proteas	YGL048C	0.208		-0.061	-0.039	0.003	
4	Resp	YBR039W	-0.179	-0.219	-0.097		-0.011	
5	Ribo	YKL180W	-0.085	-0.161	-0.061	-0.265	-0.419	
6	Ribo	YHR021C	-0.216	-0.253	-0.228	-0.168	-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.

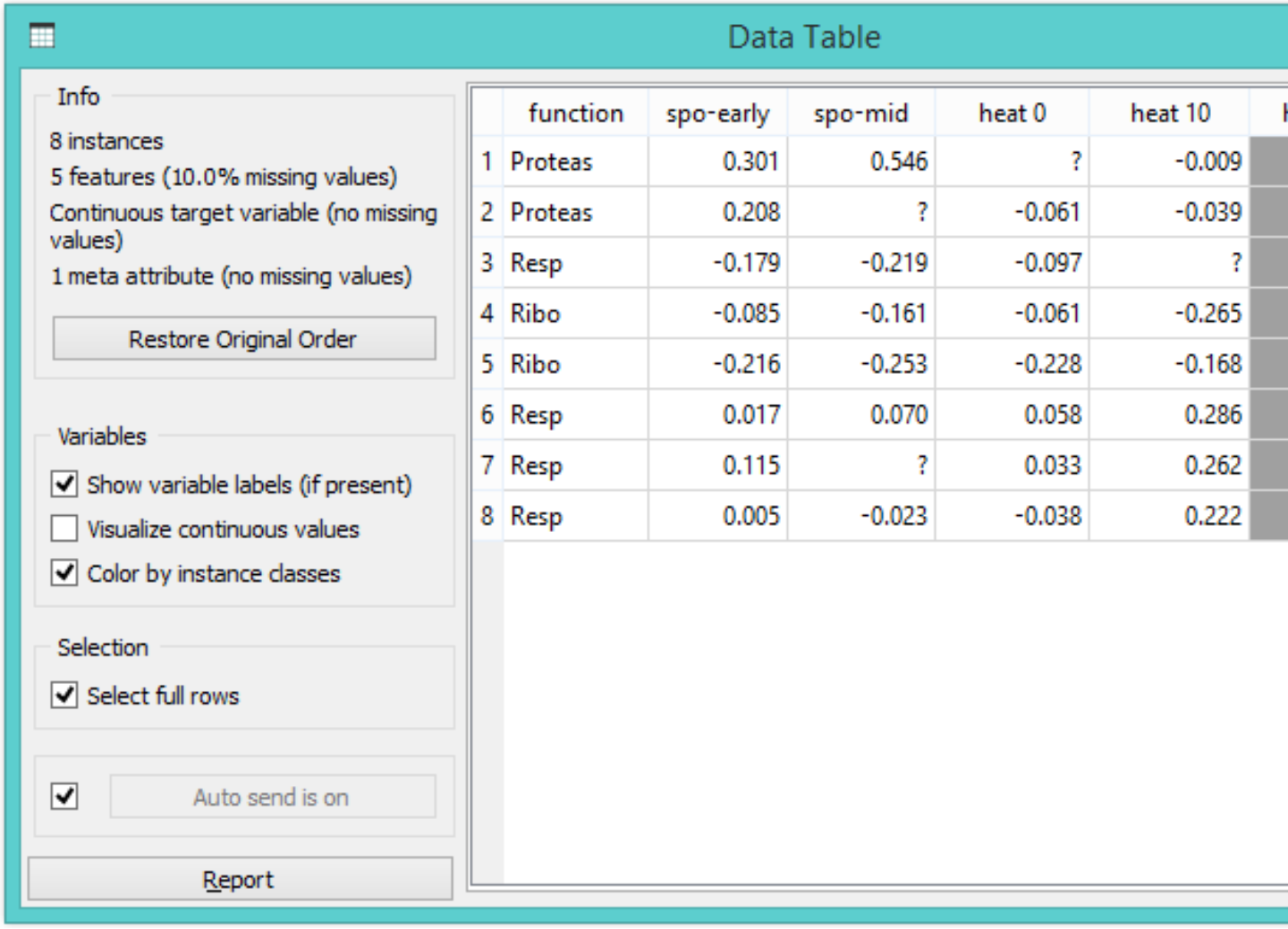
Let us start with a simple workflow that reads the data and displays it in a table:

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:

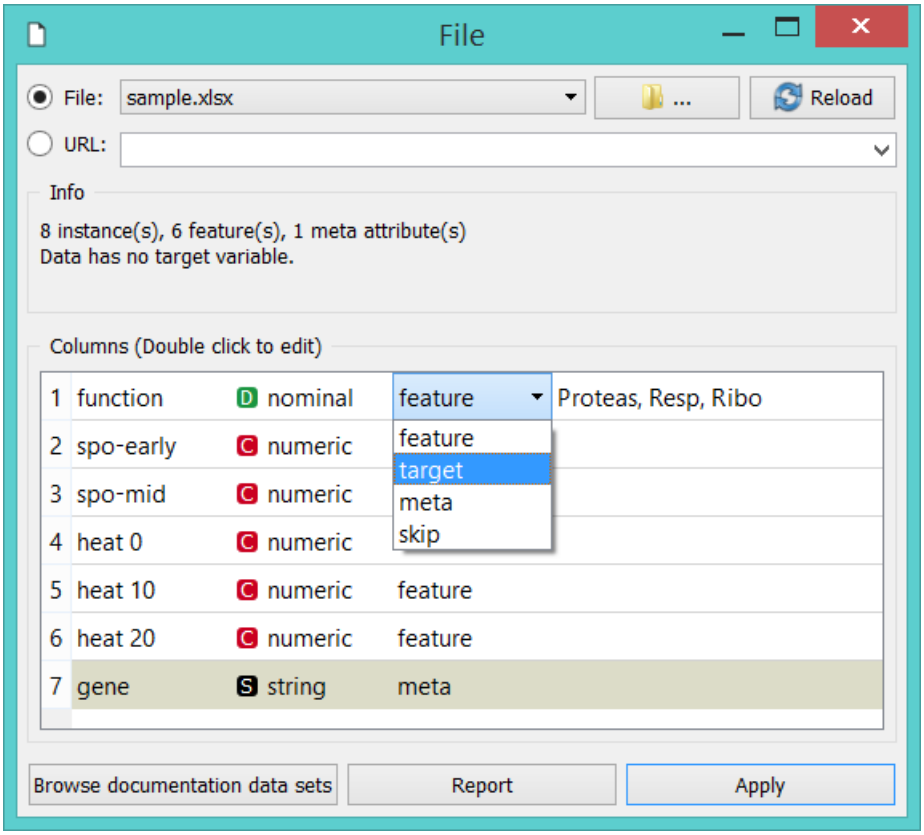


File Widget: Setting the Attribute Type and Role

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



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.



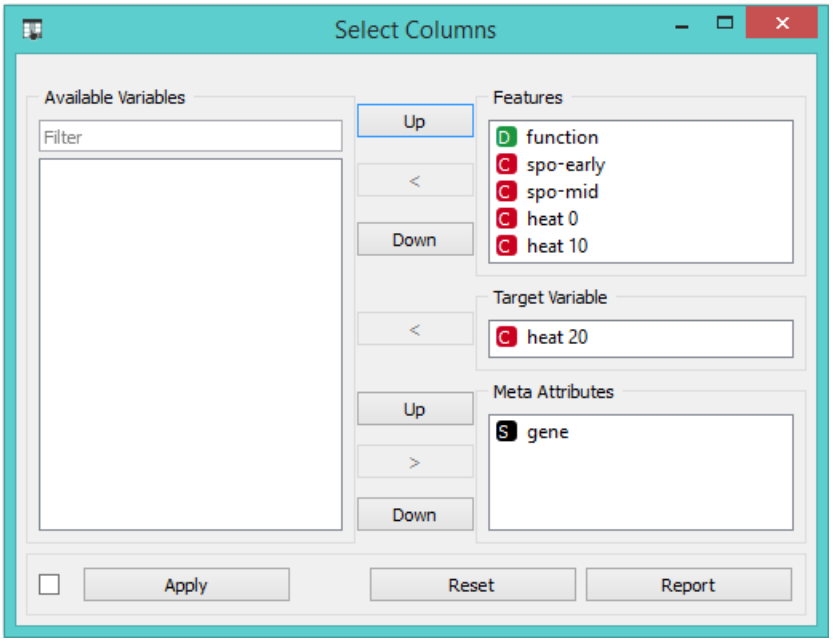
You can also change attribute type from nominal to numeric, from string to datetime, and so on. Naturally, data values have to suit the specified attribute type. Datetime accepts only values in ISO 8601 format, e.g. 2016-01-01 16:16:01. Orange would also assume the attribute is numeric if it has several 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.

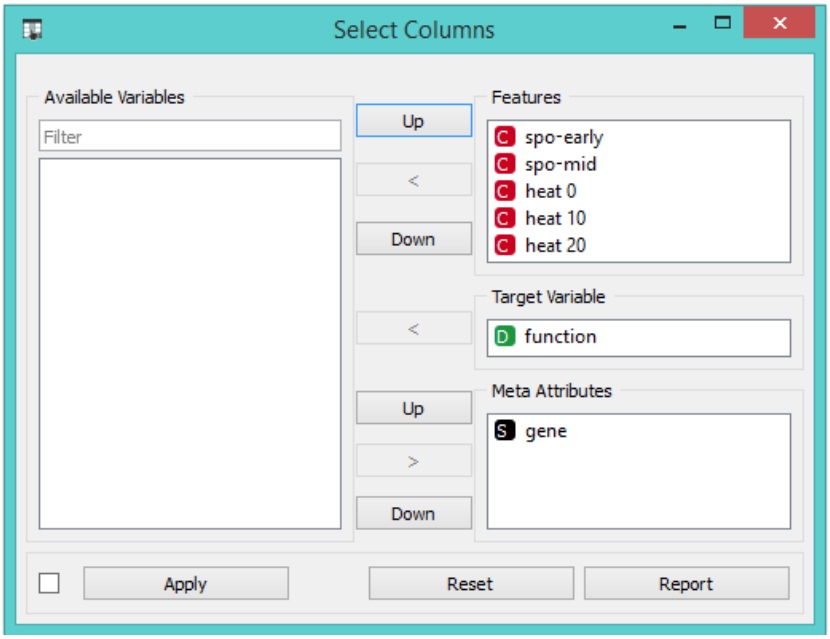
Select Columns: Setting the Attribute Role

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

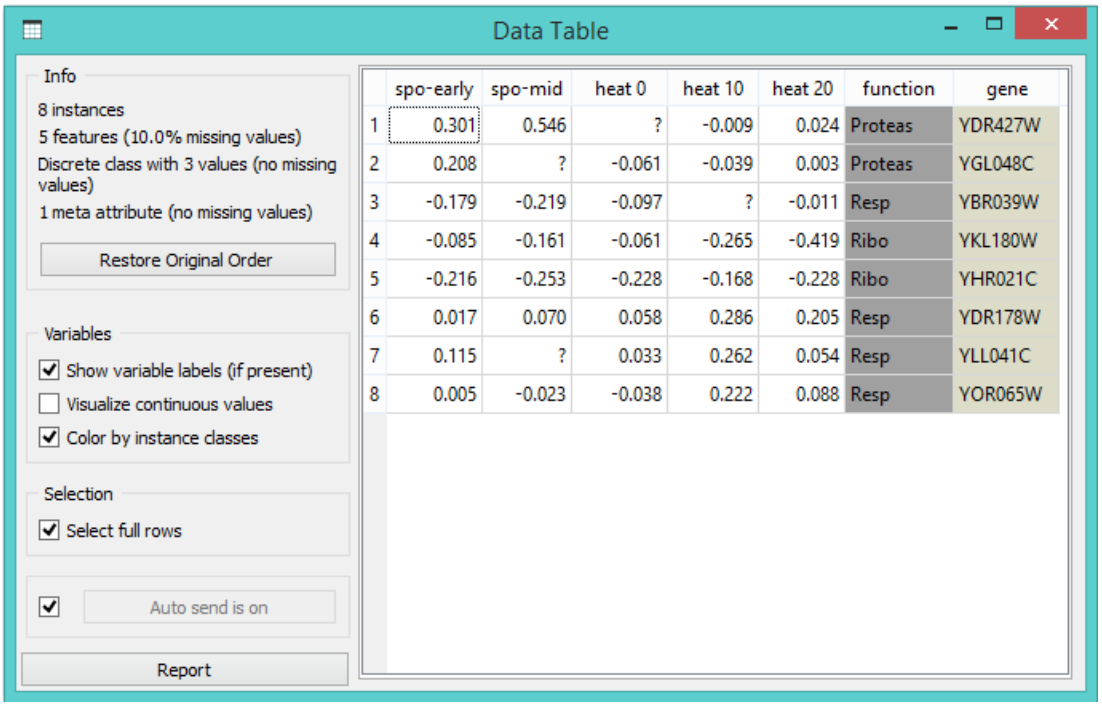
Opening *Select Columns* reveals Orange's classification of attributes. We would like all of our continuous attributes to be data features, gene function to be our target variable and gene names considered as meta attributes. We can obtain this by dragging the attribute names around the boxes in **Select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



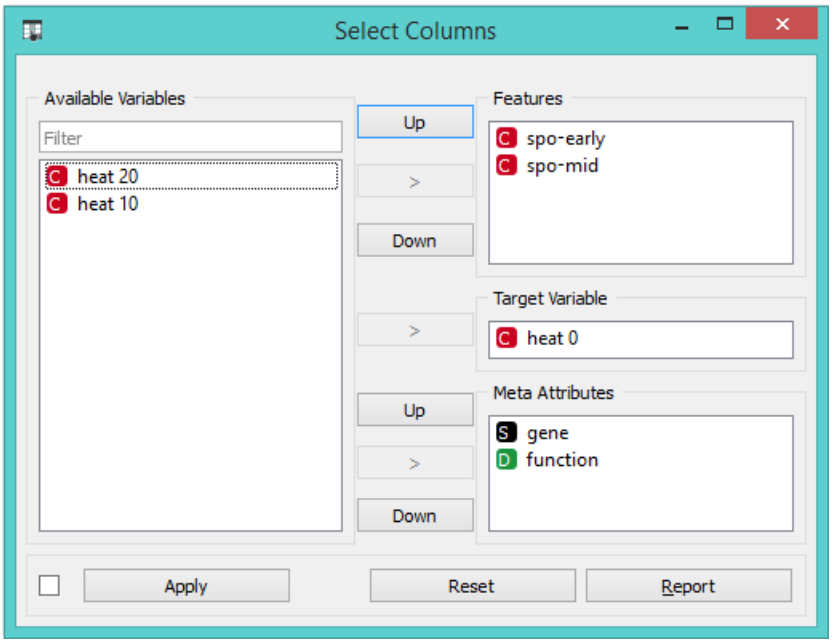
To correctly reassign attribute types, drag attribute named *function* to a **Class** box, and attribute named *gene* to a **Meta Attribute** box. The *Select Columns* widget should now look like this:



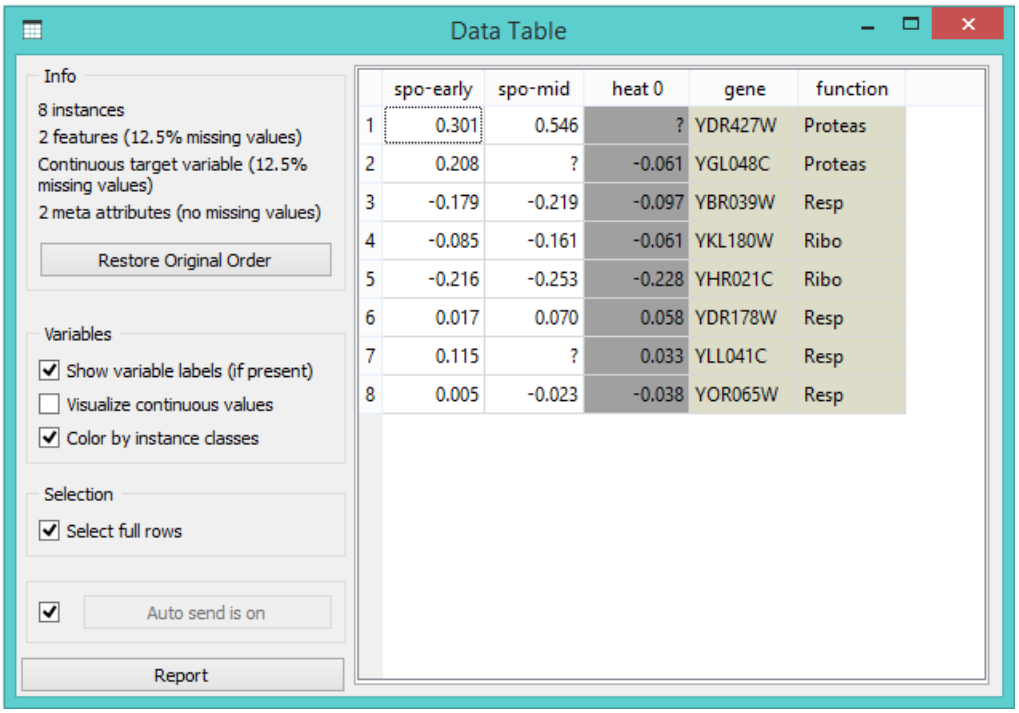
Change of attribute types in *Select Columns* widget should be confirmed by clicking the **Apply** button. The data from this widget is fed into *Data Table* that now renders the data just the way we intended:



We could also define the domain for this dataset in a different way. Say, we could make the dataset 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 dataset:



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



1.1.3 Header with Attribute Type Information

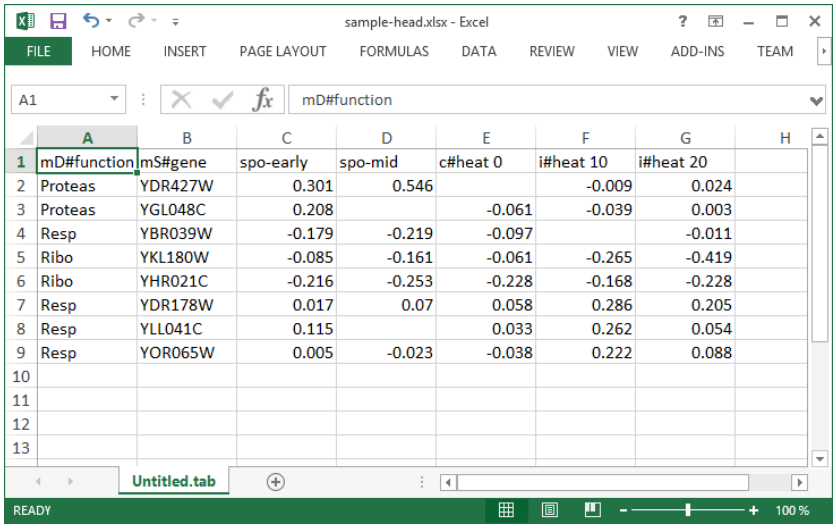
Consider again the `sample.xlsx` dataset. This time we will augment the names of the attributes with prefixes that define attribute type (continuous, discrete, time, string) and role (class or meta attribute) Prefixes are separated from the attribute name with a hash sign ("#"). Prefixes for attribute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

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

and for the type:

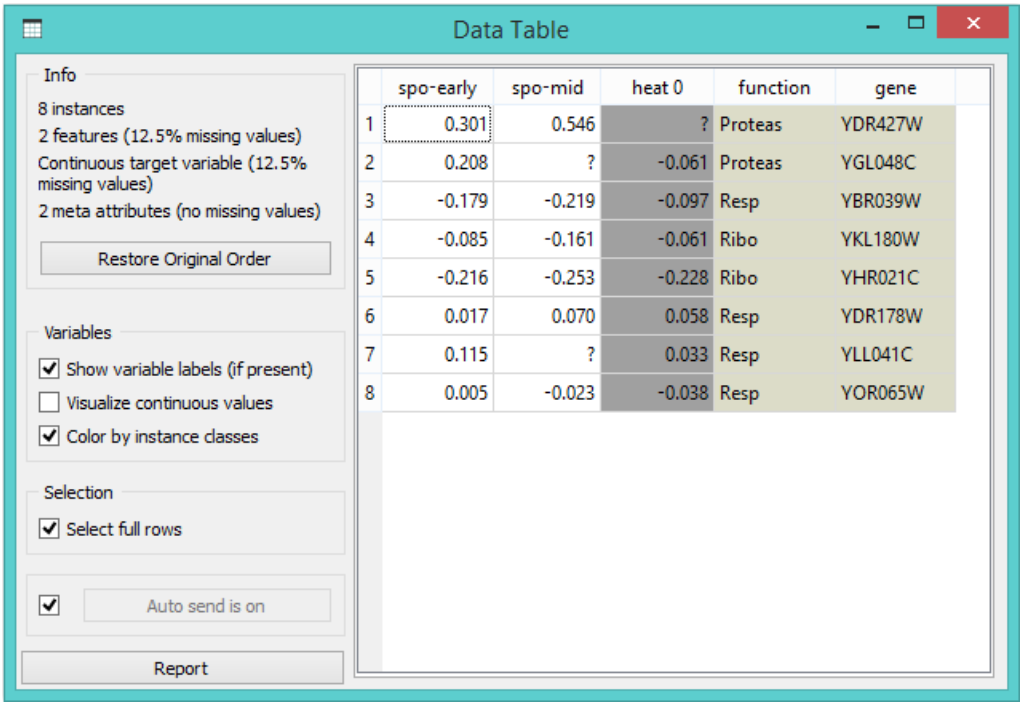
- C: Continuous
- D: Discrete
- T: Time
- S: String

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



	A	B	C	D	E	F	G	H
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								

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

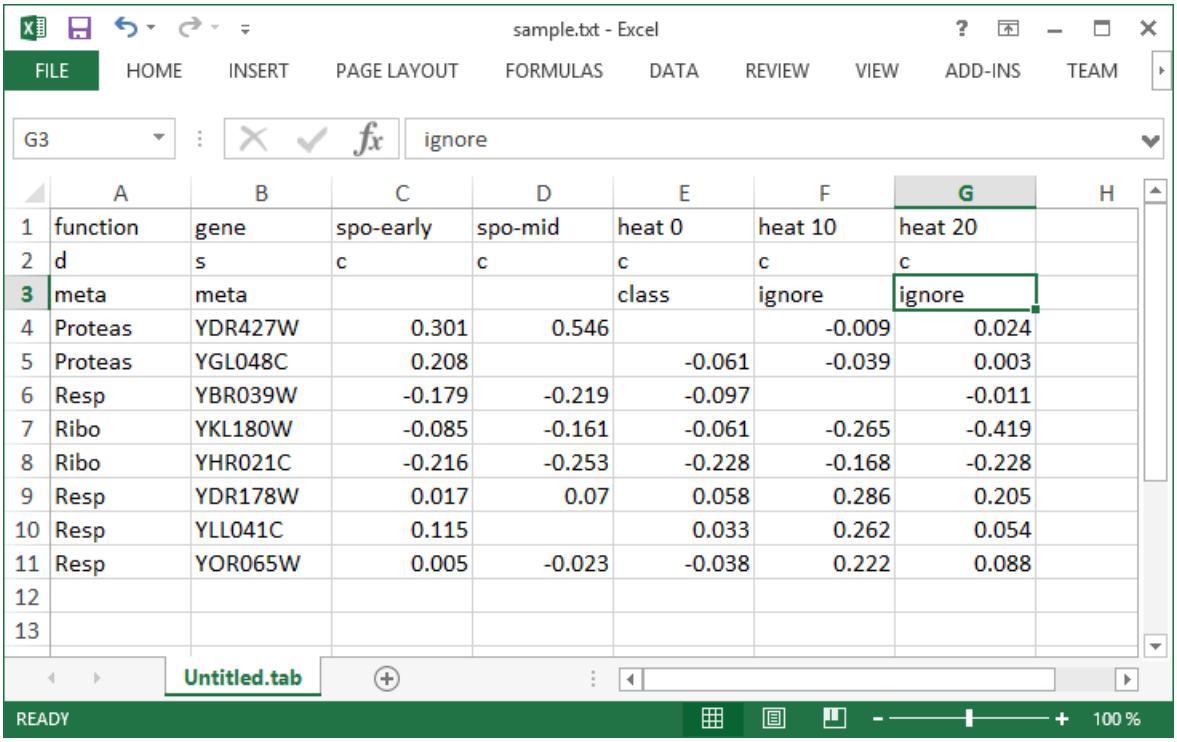


	spo-early	spo-mid	heat 0	function	gene
1	0.301	0.546	?	Proteas	YDR427W
2	0.208	?	-0.061	Proteas	YGL048C
3	-0.179	-0.219	-0.097	Resp	YBR039W
4	-0.085	-0.161	-0.061	Ribo	YKL180W
5	-0.216	-0.253	-0.228	Ribo	YHR021C
6	0.017	0.070	0.058	Resp	YDR178W
7	0.115	?	0.033	Resp	YLL041C
8	0.005	-0.023	-0.038	Resp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

Notice that the attributes we have ignored (label "i" in the attribute name) are not present in the dataset.

1.1.4 Three-Row Header Format

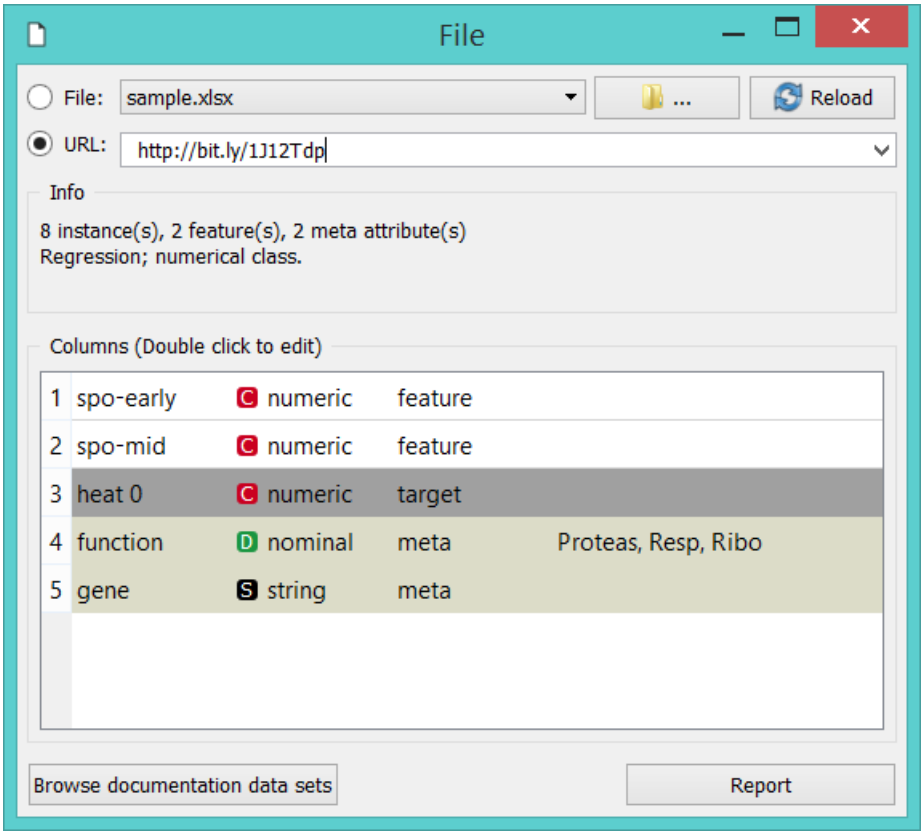
Orange's legacy native data format is a tab-delimited text file with three header rows. The first row lists the attribute names, the second row defines their type (continuous, discrete, time and string, or abbreviated c, d, t, and s), and the third row an optional role (class, meta, weight, or ignore).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



	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								

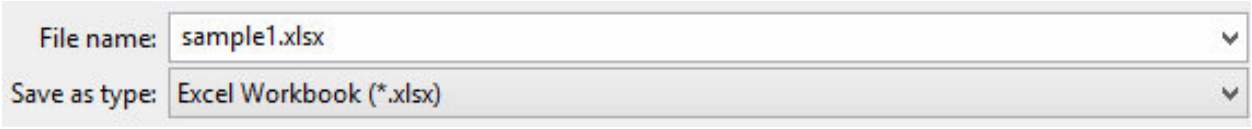
1.1.5 Data from Google Sheets

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:



1.1.6 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*).



1.1.7 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
```

CHAPTER 2

Widgets

2.1 Data

2.1.1 File

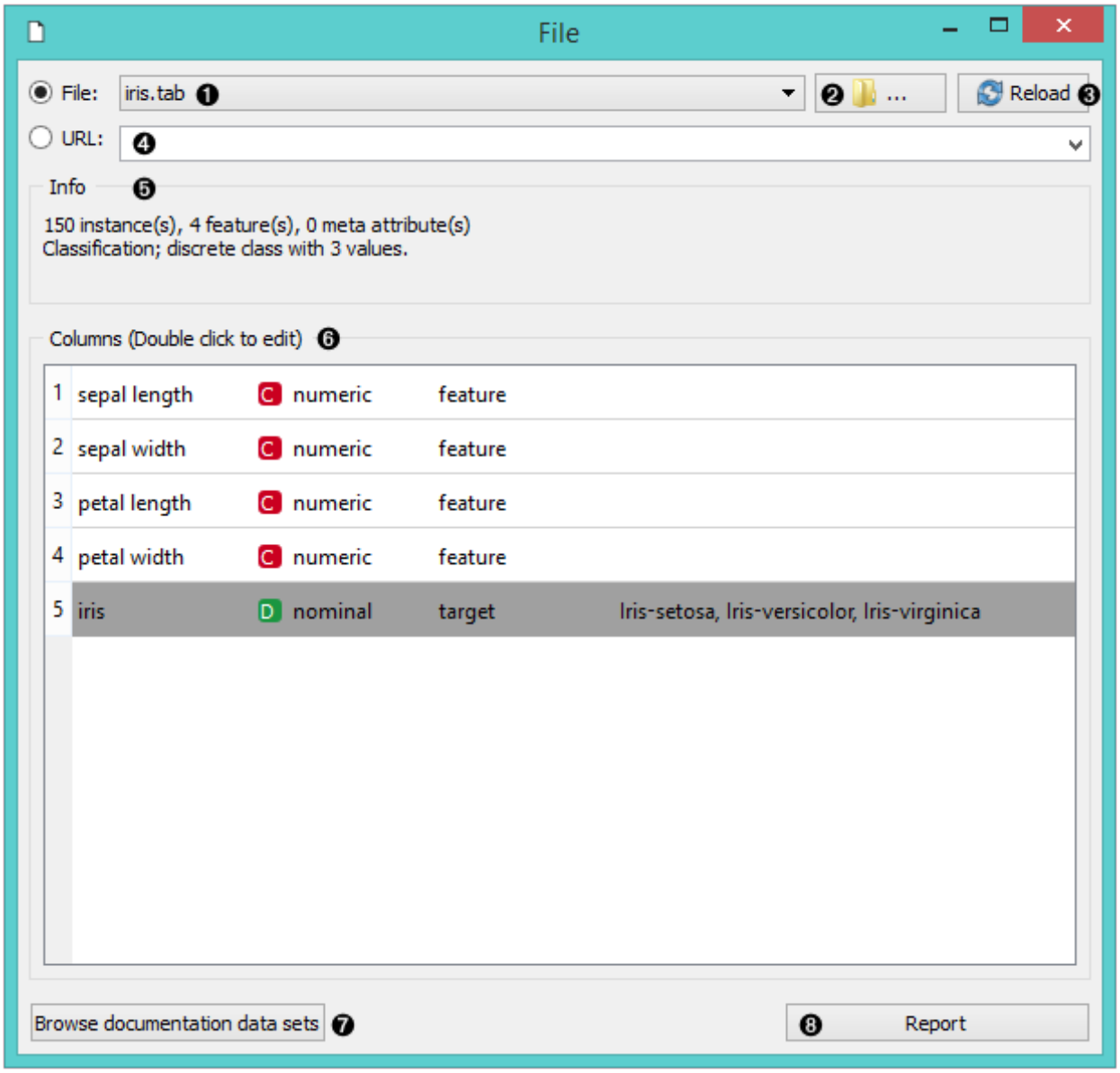
Reads attribute-value data from an input file.

Outputs

- Data: dataset from the file

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

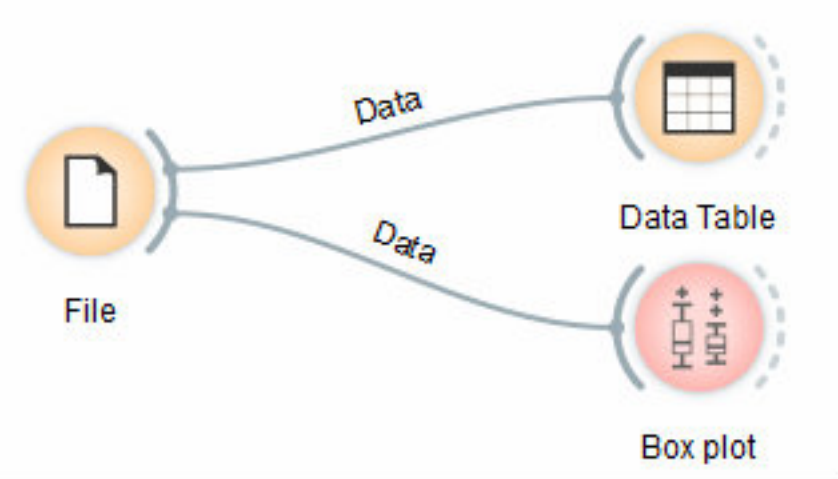
The widget reads data from Excel (**.xlsx**), simple tab-delimited (**.txt**), comma-separated files (**.csv**) or URLs.



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 dataset: dataset size, number and types of data features.
6. Additional information on the features in the dataset. 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 datasets.
8. Produce a report.

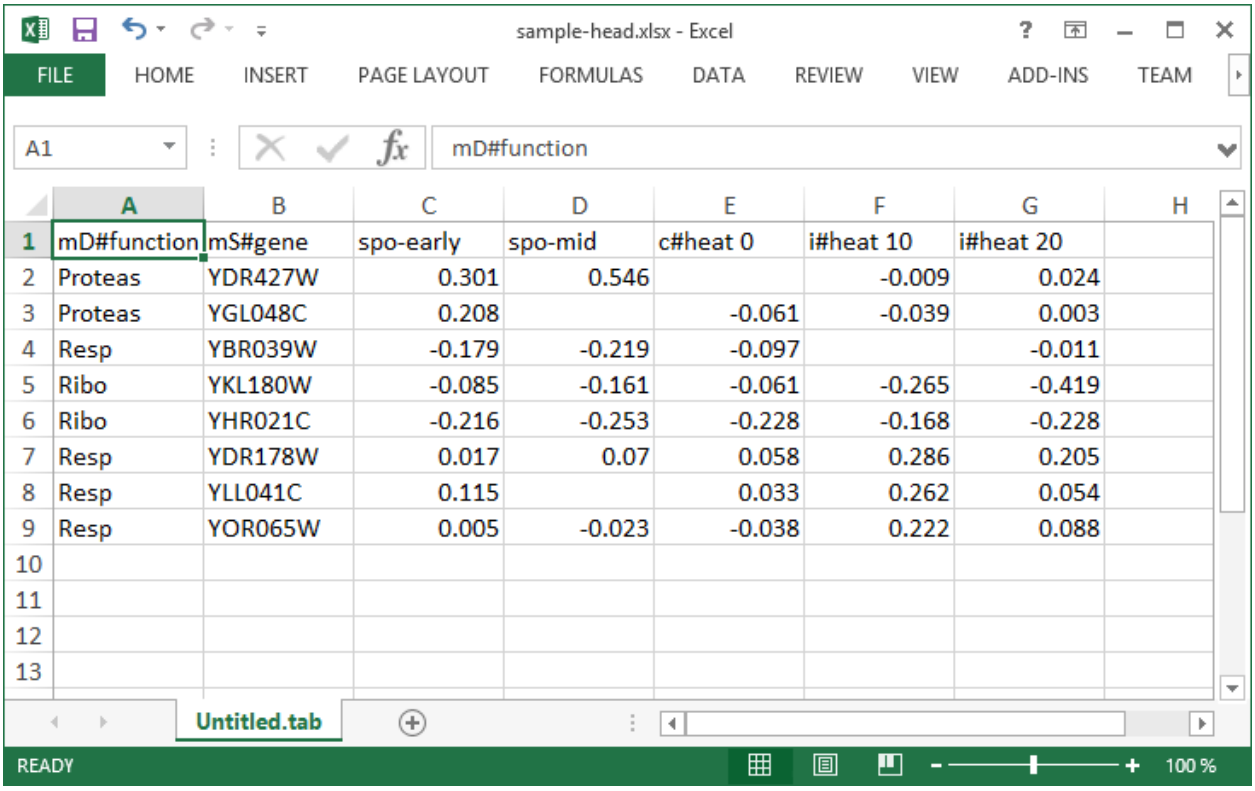
Example

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.
- 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 *time*).



	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								

Read more on loading your data *here*.

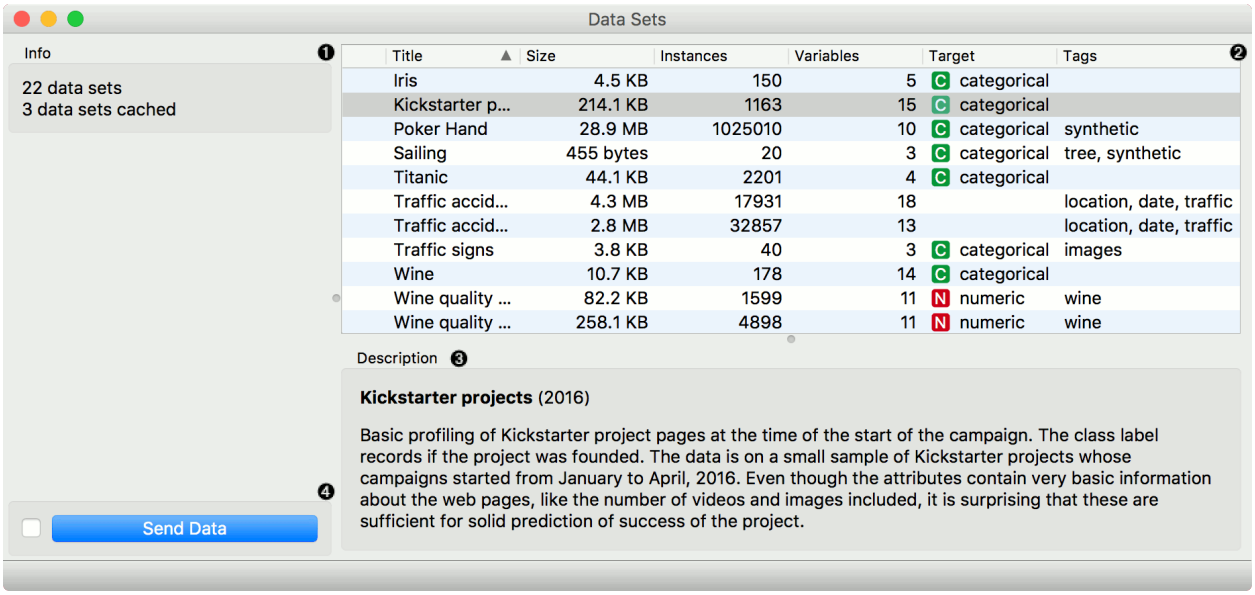
2.1.2 Datasets

Load a dataset from an online repository.

Outputs

- Data: output dataset

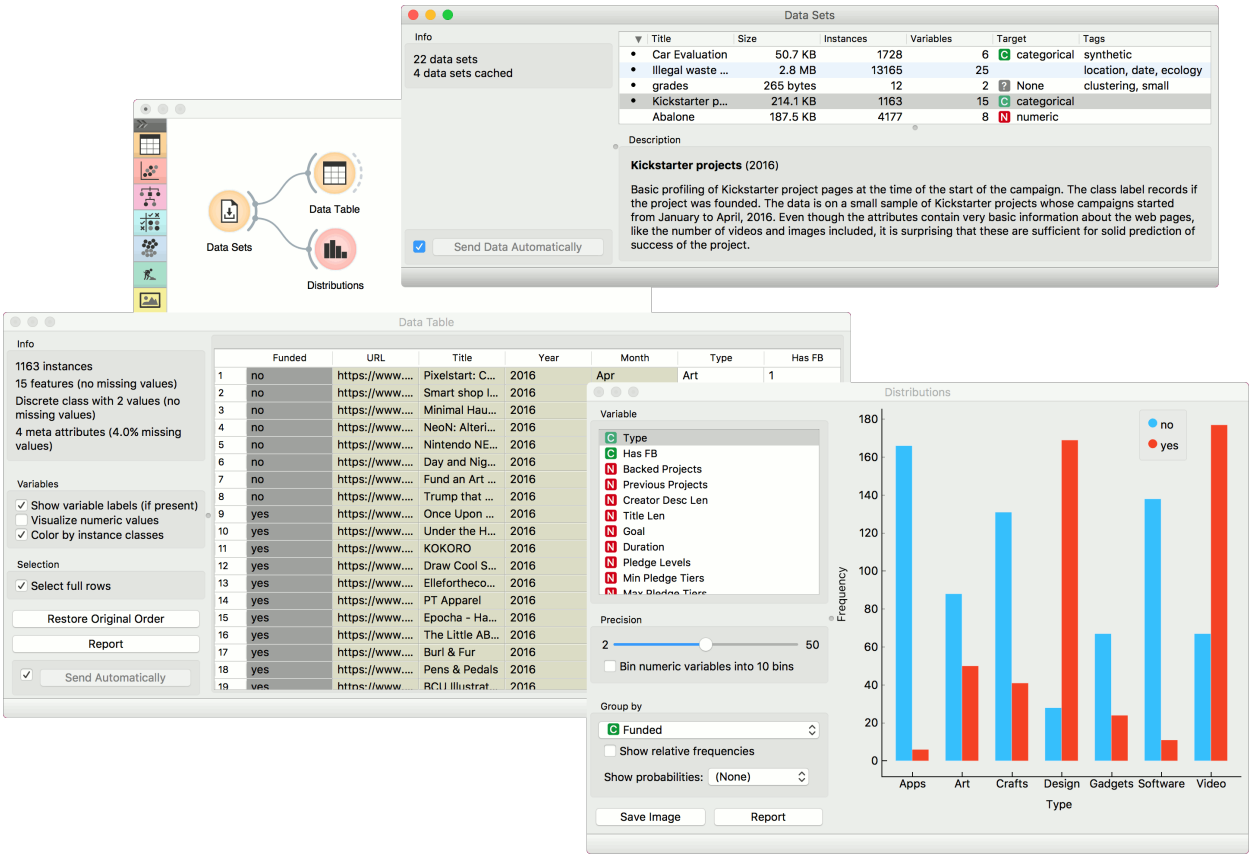
Datasets widget retrieves selected dataset from the server and sends it to the output. File is downloaded to the local memory and thus instantly available even without the internet connection. Each dataset is provided with a description and information on the data size, number of instances, number of variables, target and tags.



1. Information on the number of datasets available and the number of them downloaded to the local memory.
2. Content of available datasets. Each dataset is described with the size, number of instances and variables, type of the target variable and tags.
3. Formal description of the selected dataset.
4. If *Send Data Automatically* is ticked, selected dataset is communicated automatically. Alternatively, press *Send Data*.

Example

Orange workflows can start with **Datasets** widget instead of **File** widget. In the example below, the widget retrieves a dataset from an online repository (Kickstarter data), which is subsequently sent to both the Data Table and the Distributions.



2.1.3 SQL Table

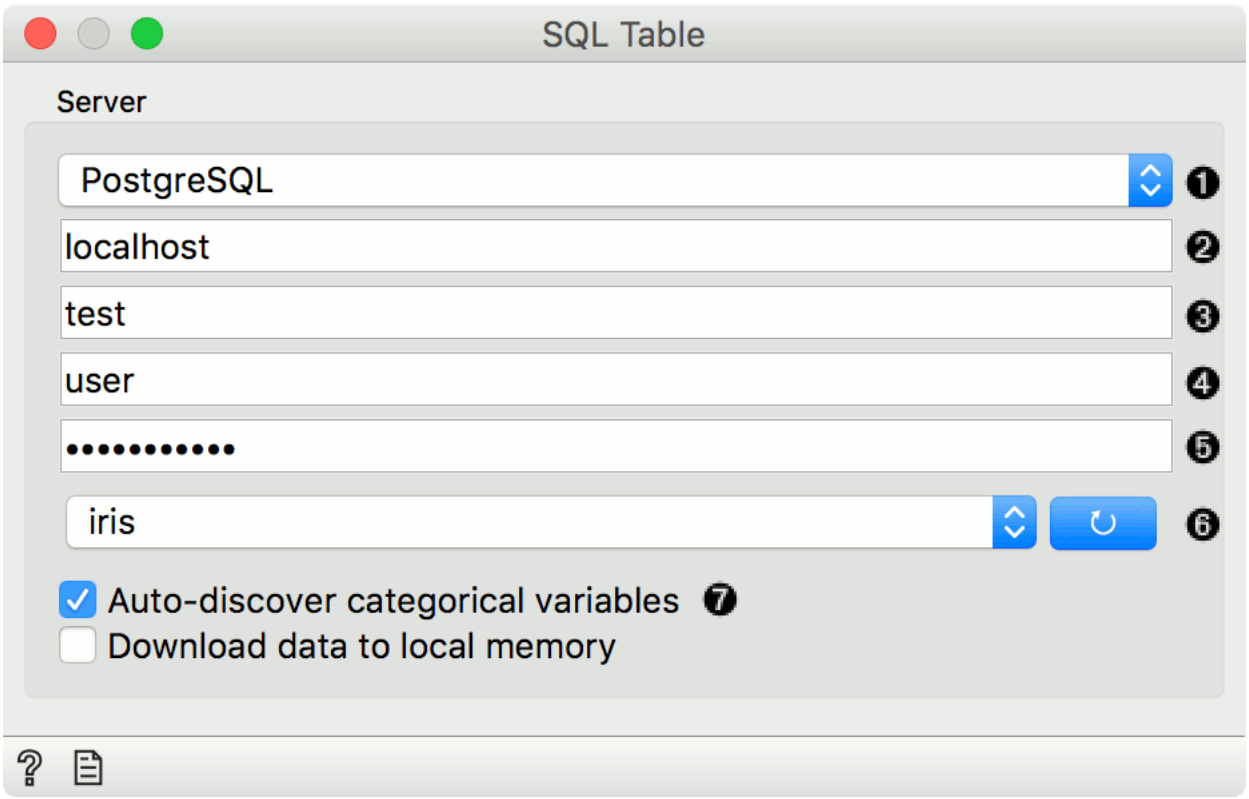
Reads data from an SQL database.

Outputs

- Data: dataset from the database

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

To handle large databases, Orange attempts to execute a part of the computation in the database itself without downloading the data. This only works with PostgreSQL database and requires quantile and tsm_system_time extensions installed on server. If these extensions are not installed, the data will be downloaded locally.



1. Database type (can be either PostgreSQL or MSSQL).
2. Host name.
3. Database name.
4. Username.
5. Password.
6. Press the blue button to connect to the database. Then select the table in the dropdown.
7. *Auto-discover categorical variables* will cast INT and CHAR columns with less than 20 distinct values as categorical variables (finding all distinct values can be slow on large tables). When not selected, INT will be treated as numeric and CHAR as text. *Download to local memory* downloads the selected table to your local machine.

##Installation Instructions

###PostgreSQL

Install the backend.

```
pip install psycopg2
```

Alternatively, you can follow these instructions for installing the backend.

Install the extensions. [optional]

###MSSQL

Install the backend.

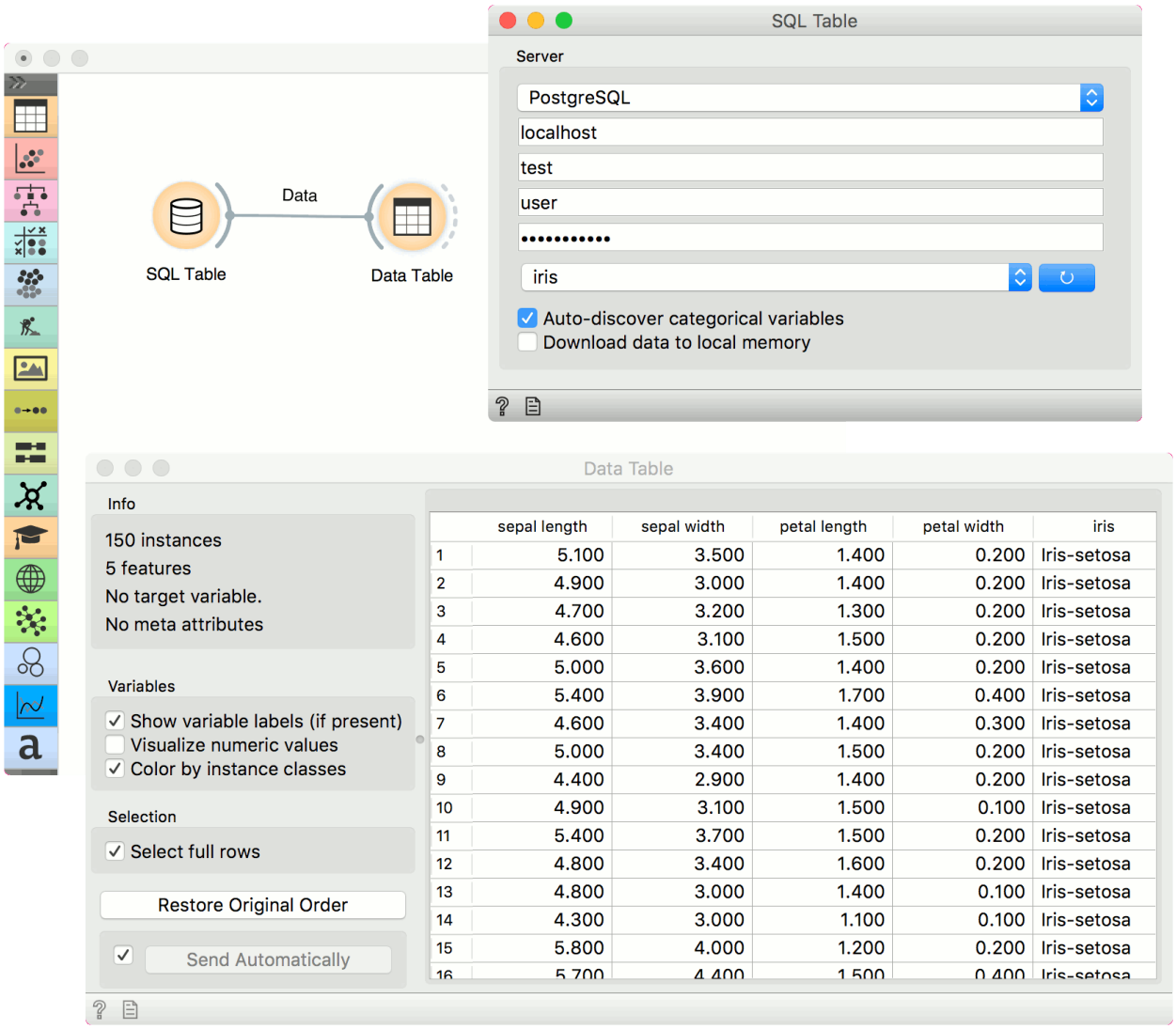
```
pip install pymssql
```

If you are encountering issues, follow these instructions.

##Example

Here is a simple example on how to use the **SQL Table** widget. Place the widget on the canvas, enter your database credentials and connect to your database. Then select the table you wish to analyse.

Connect **SQL Table** to Data Table widget to inspect the output. If the table is populated, your data has transferred correctly. Now, you can use the **SQL Table** widget in the same way as the File widget.



2.1.4 Save Data

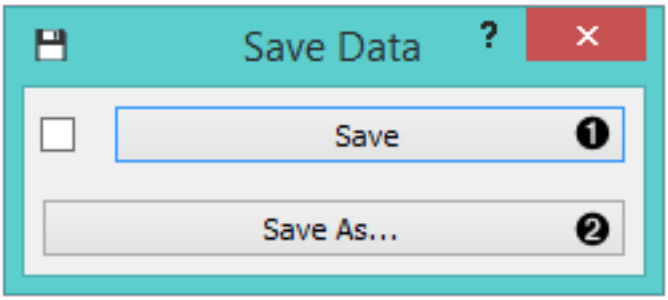
Saves data to a file.

Inputs

- Data: input dataset

The **Save Data** widget considers a dataset 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.

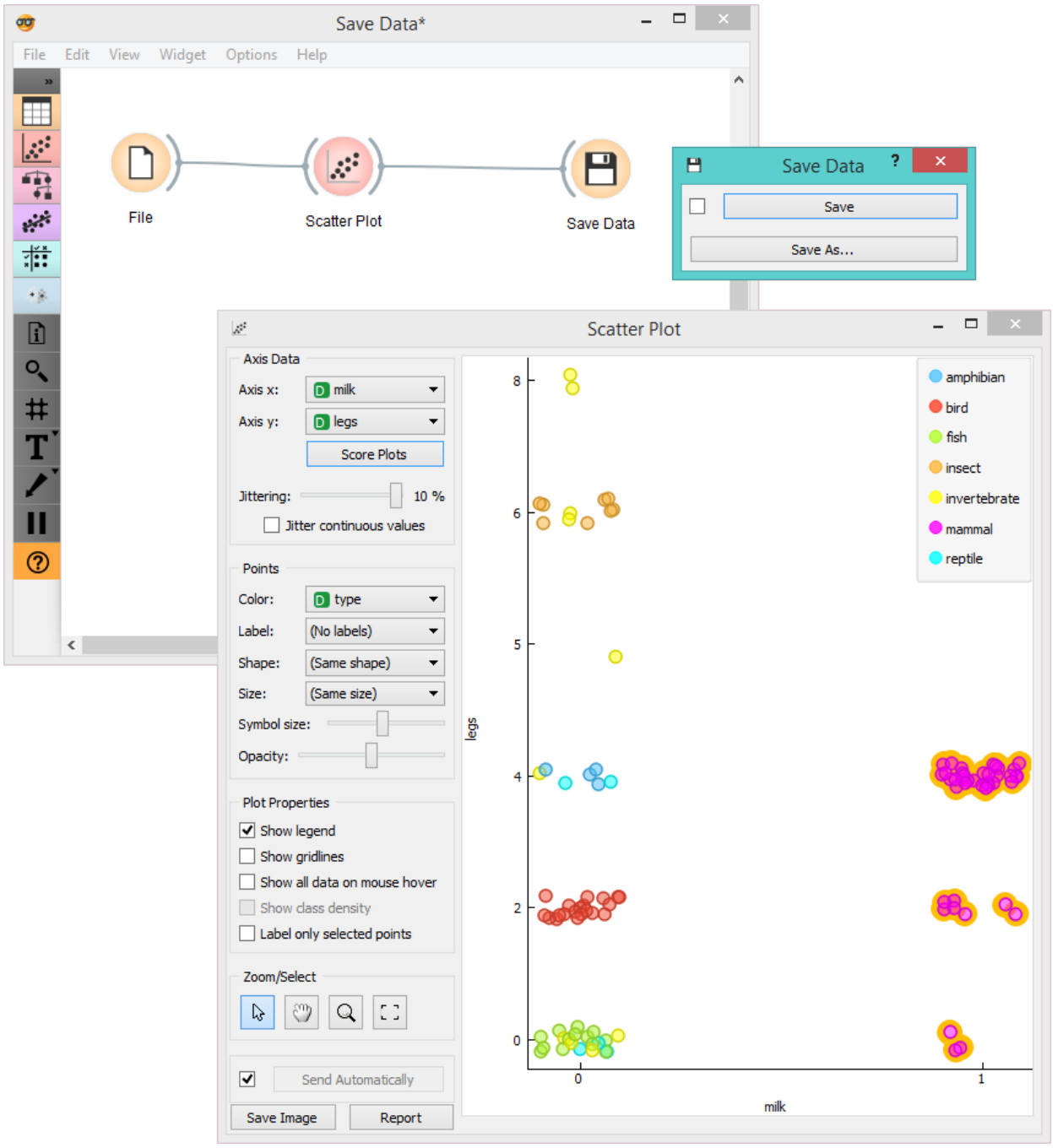
The widget does not save the data every time it receives a new signal in the input as this would constantly (and, mostly, inadvertently) overwrite the file. Instead, the data is saved only after a new file name is set or the user pushes the *Save* button.



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

Example

In the workflow below, we used the *Zoo* dataset. 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.



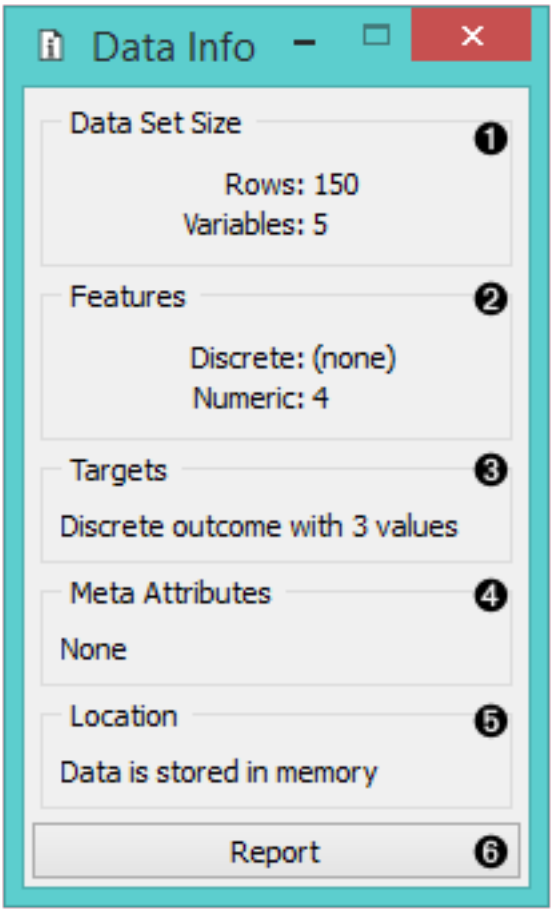
2.1.5 Data Info

Displays information on a selected dataset.

Inputs

- Data: input dataset

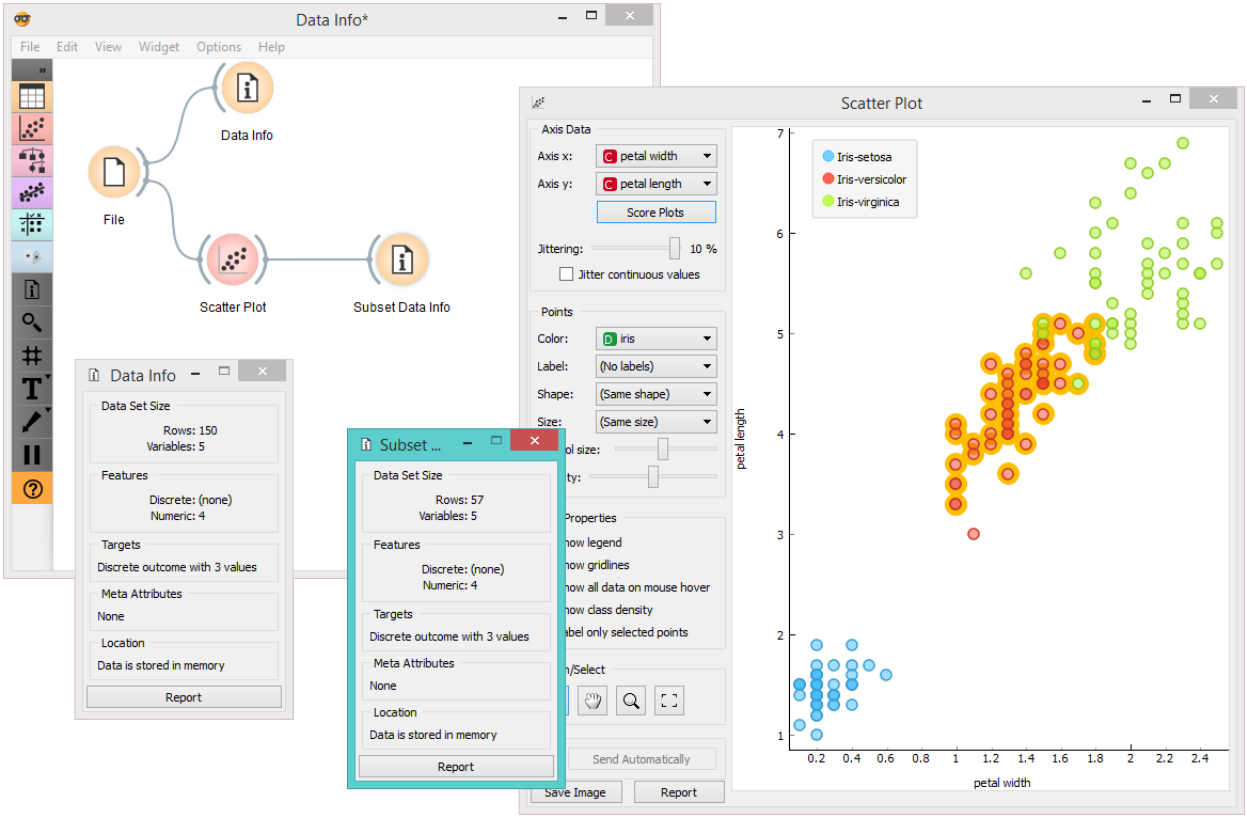
A simple widget that presents information on dataset size, features, targets, meta attributes, and location.



1. Information on dataset 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 dataset and the other with information on the (manually) selected subset from the Scatter Plot widget. We used the *Iris* dataset.



2.1.6 Data Table

Displays attribute-value data in a spreadsheet.

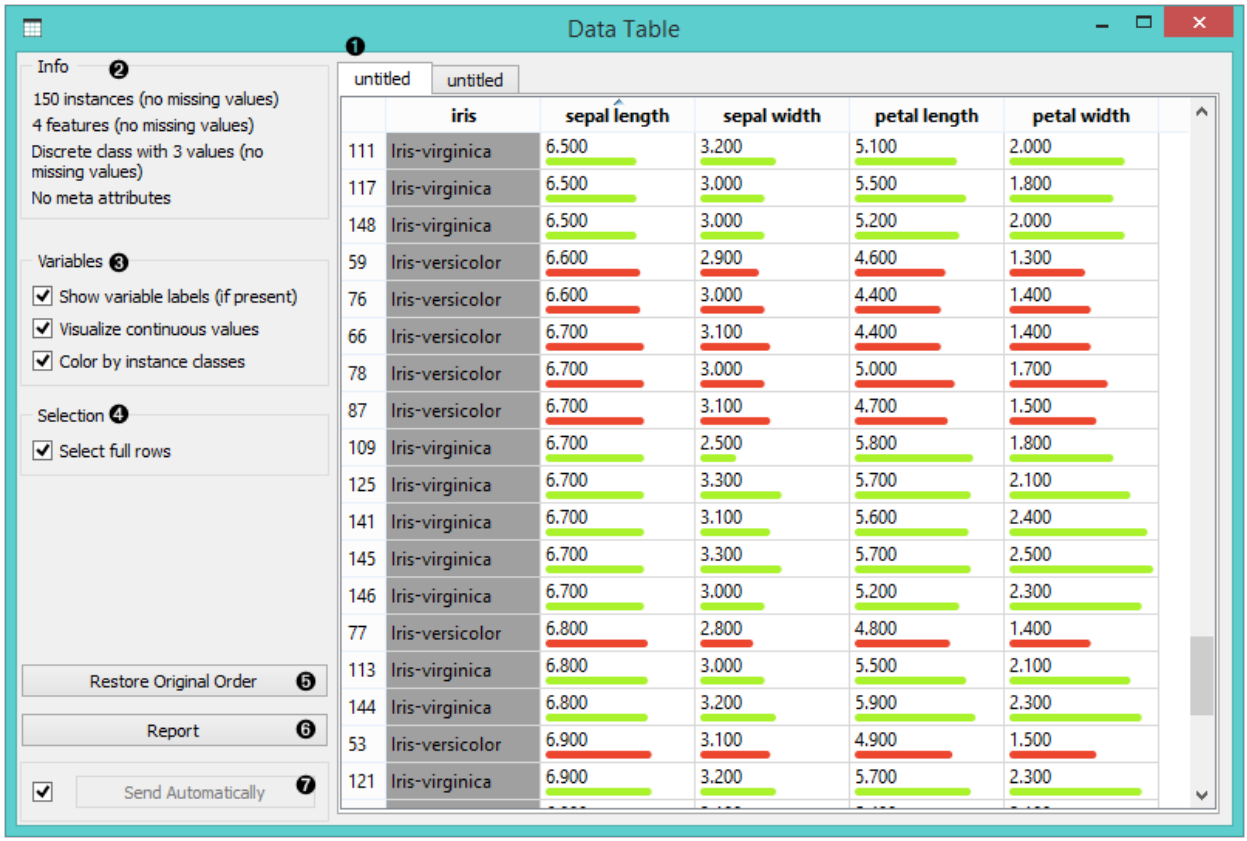
Inputs

- Data: input dataset

Outputs

- Selected Data: instances selected from the table

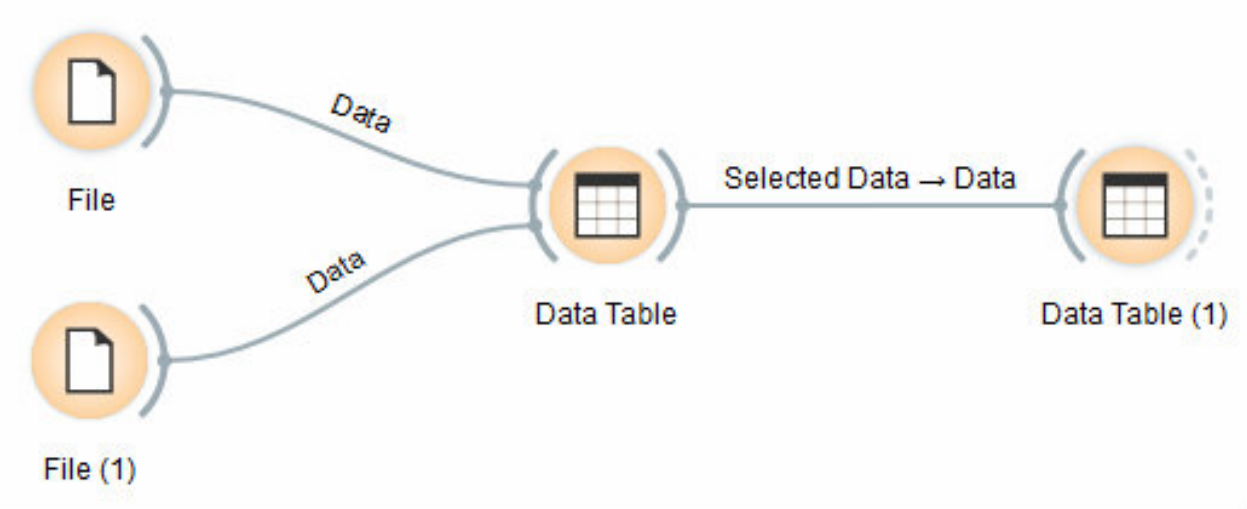
The **Data Table** widget receives one or more datasets in its input and presents them as a spreadsheet. Data instances may be sorted by attribute values. The widget also supports manual selection of data instances.



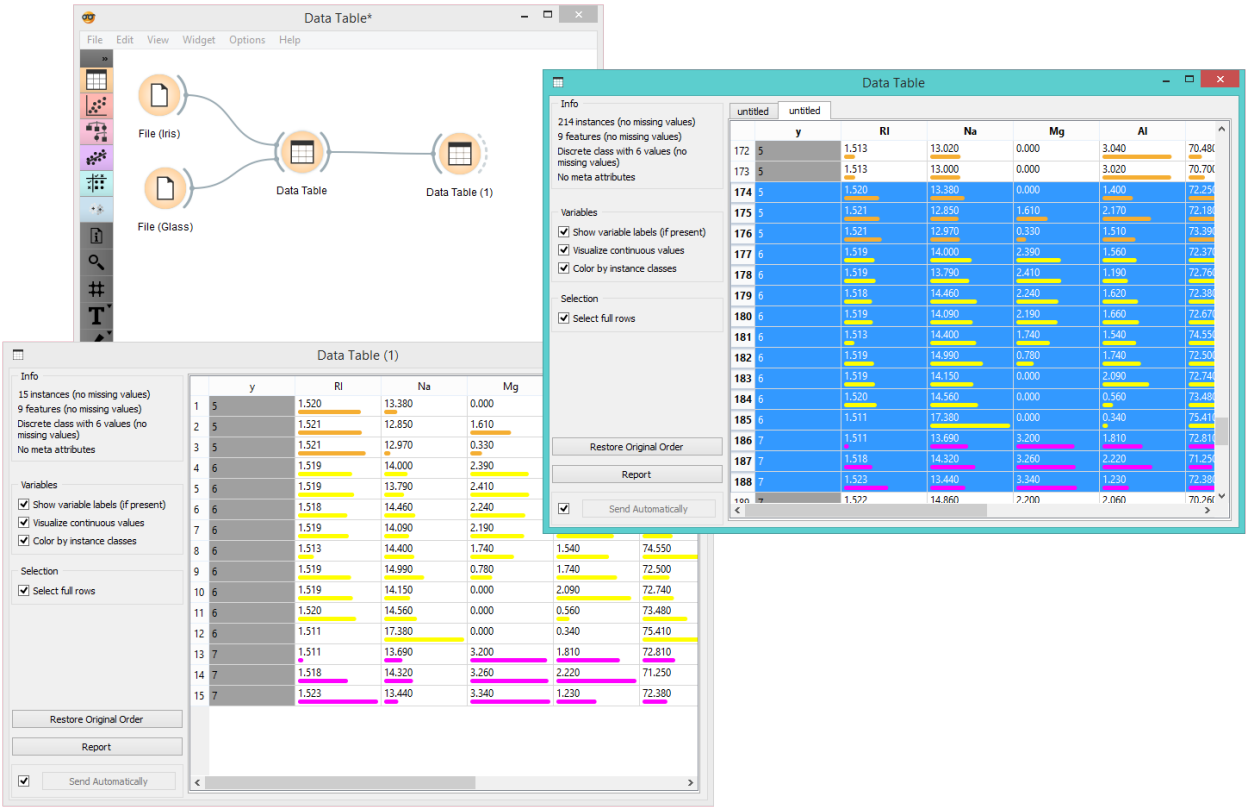
1. The name of the dataset (usually the input data file). Data instances are in rows and their attribute values in columns. In this example, the dataset is sorted by the attribute "sepal length".
2. Info on current dataset size and number and types of attributes
3. Values of continuous attributes can 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* dataset (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 dataset to view (iris or glass). Changing from one dataset to another alters the communicated selection of data instances if *Commit on any change* is selected.



2.1.7 Select Columns

Manual selection of data attributes and composition of data domain.

Inputs

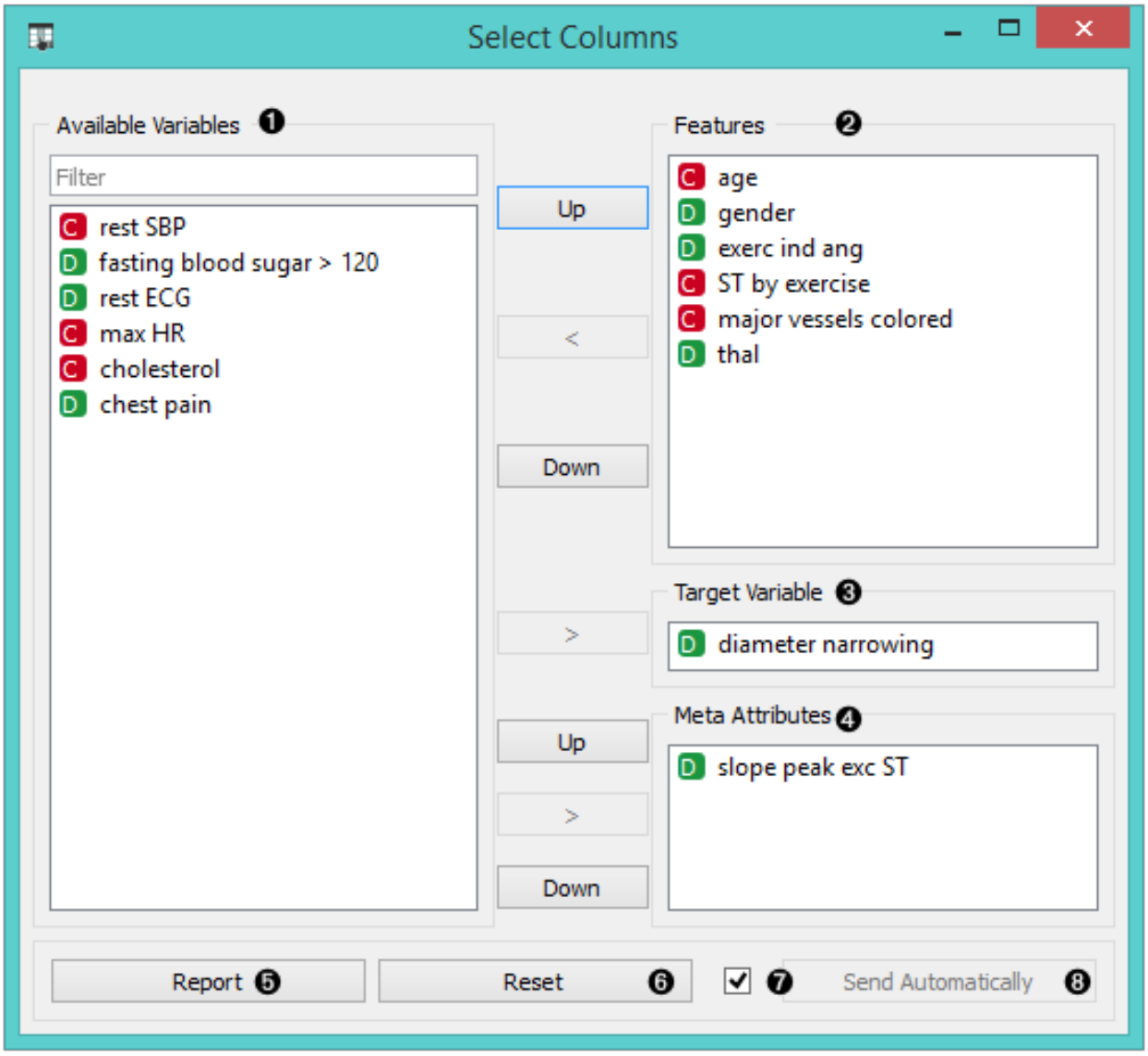
- Data: input dataset

Outputs

- Data: dataset with columns as set in the widget

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 modeling, 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).

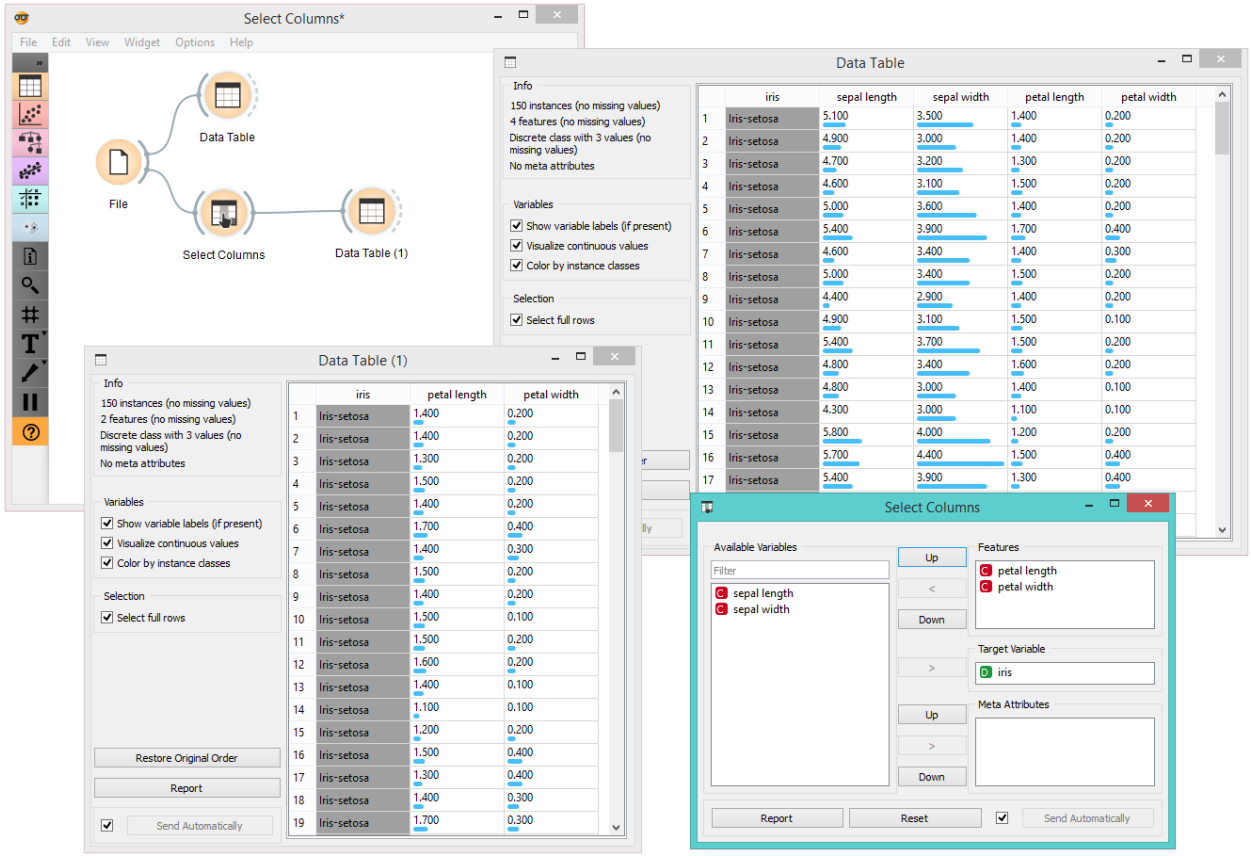


1. Left-out data attributes that will not be in the output data file
2. Data attributes in the new data file
3. Target variable. If none, the new dataset will be without a target variable.
4. Meta attributes of the new data file. These attributes are included in the dataset but are, for most methods, not considered in the analysis.

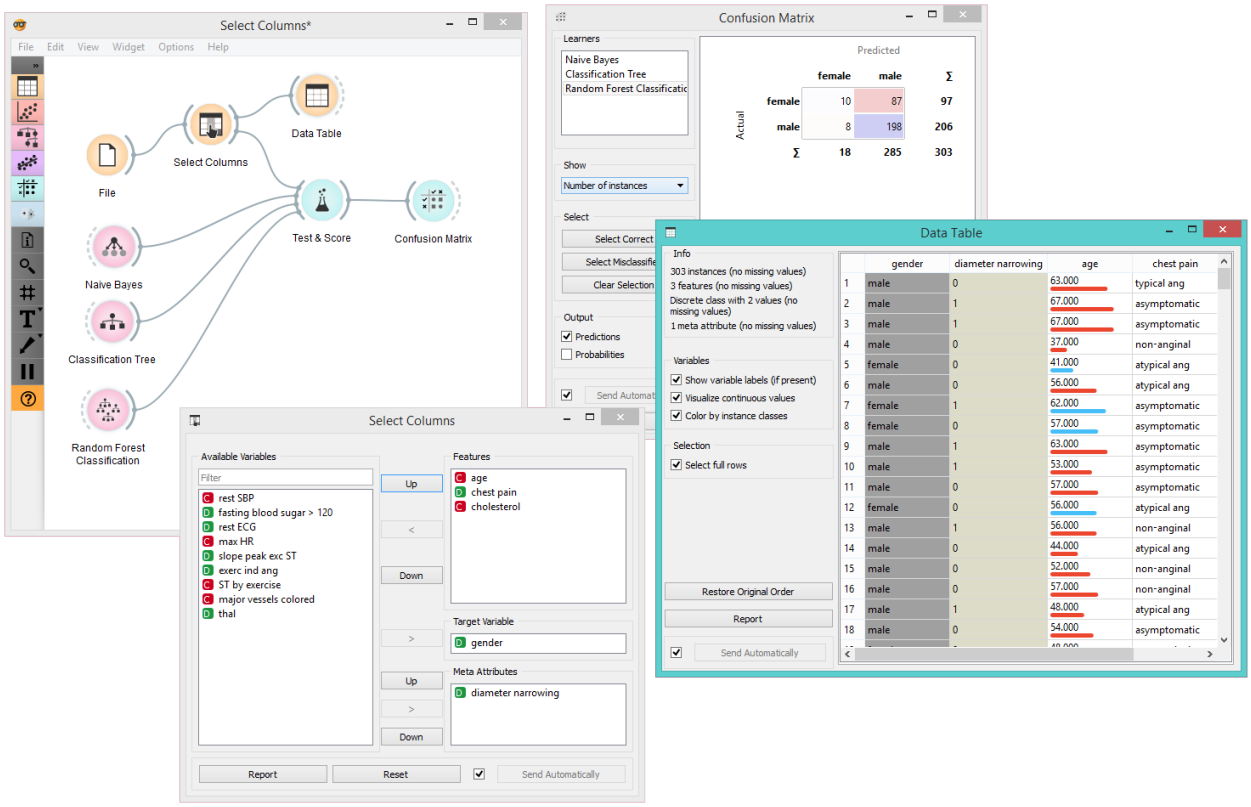
5. Produce a report.
6. Reset the domain composition to that of the input data file.
7. Tick if you wish to auto-apply changes of the data domain.
8. Apply changes of the data domain and send the new data file to the output channel of the widget.

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 dataset and the dataset with selected columns in the Data Table widget.



For a more complex use of the widget, we composed a workflow to redefine the classification problem in the *heart-disease* dataset. 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.



2.1.8 Select Rows

Selects data instances based on conditions over data features.

Inputs

- Data: input dataset

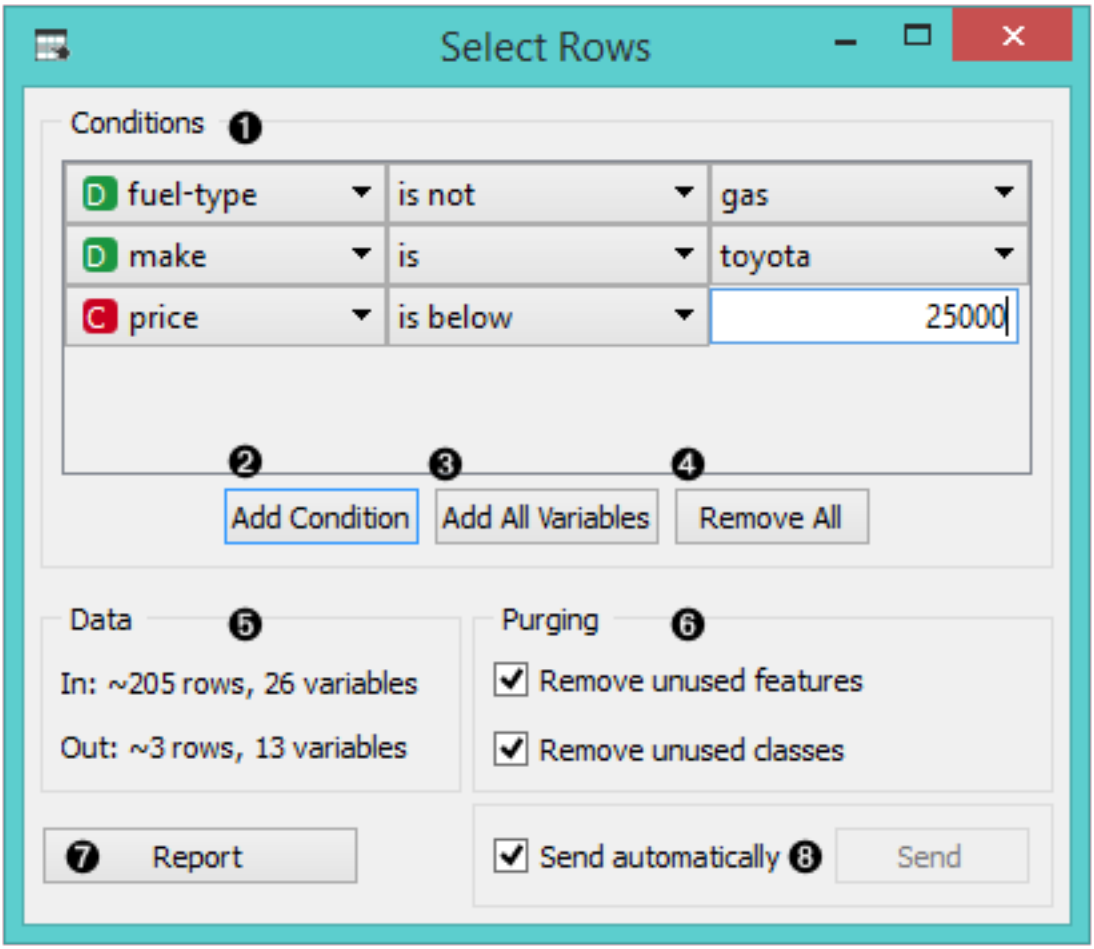
Outputs

- Matching Data: instances that match the conditions
- Non-Matching Data: instances that do not match the conditions
- Data: data with an additional column showing whether a instance is selected

This widget selects a subset from an input dataset, 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 conjunct 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.



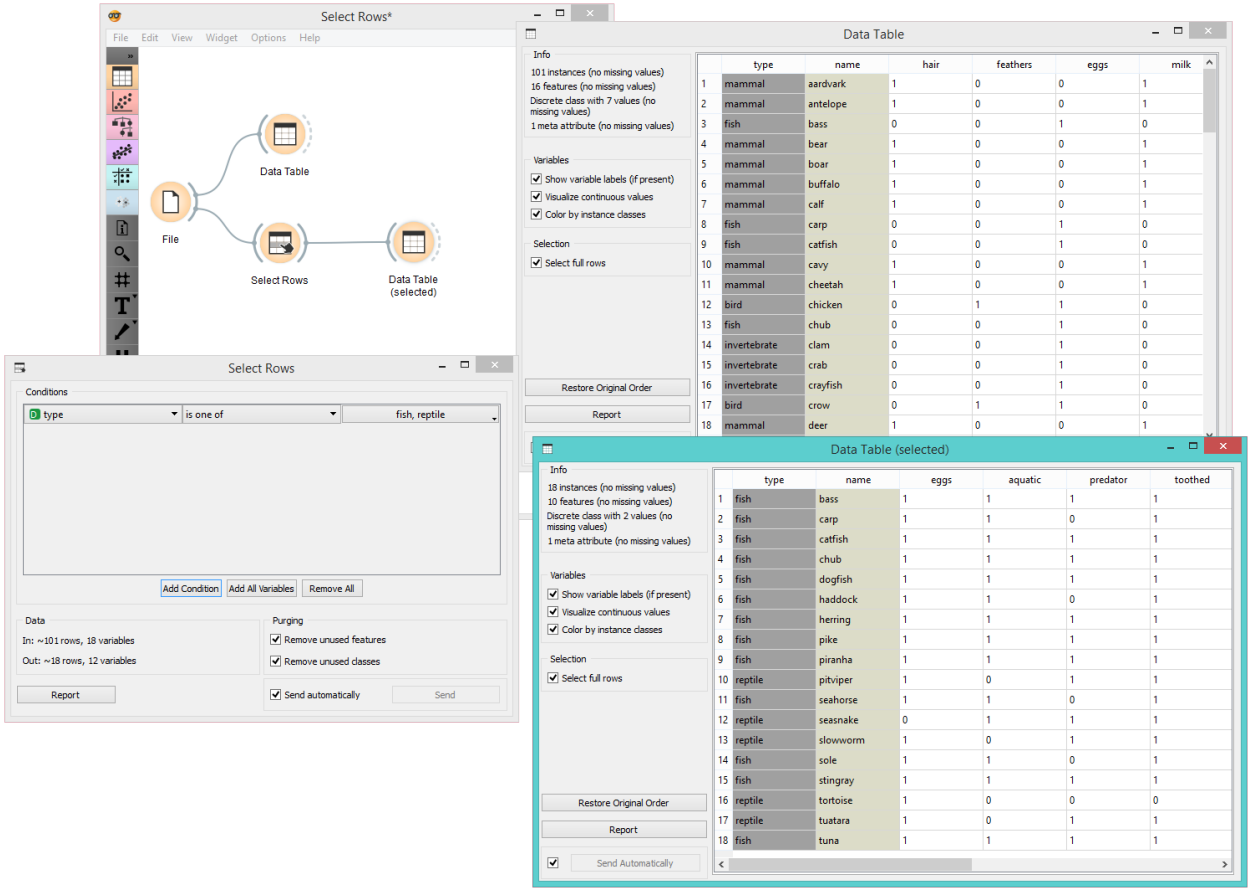
1. Conditions you want to apply, their operators and related values
2. Add a new condition to the list of conditions.
3. Add all the possible variables at once.
4. Remove all the listed variables at once.
5. Information on the input dataset and information on instances that match the condition(s)
6. 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*).

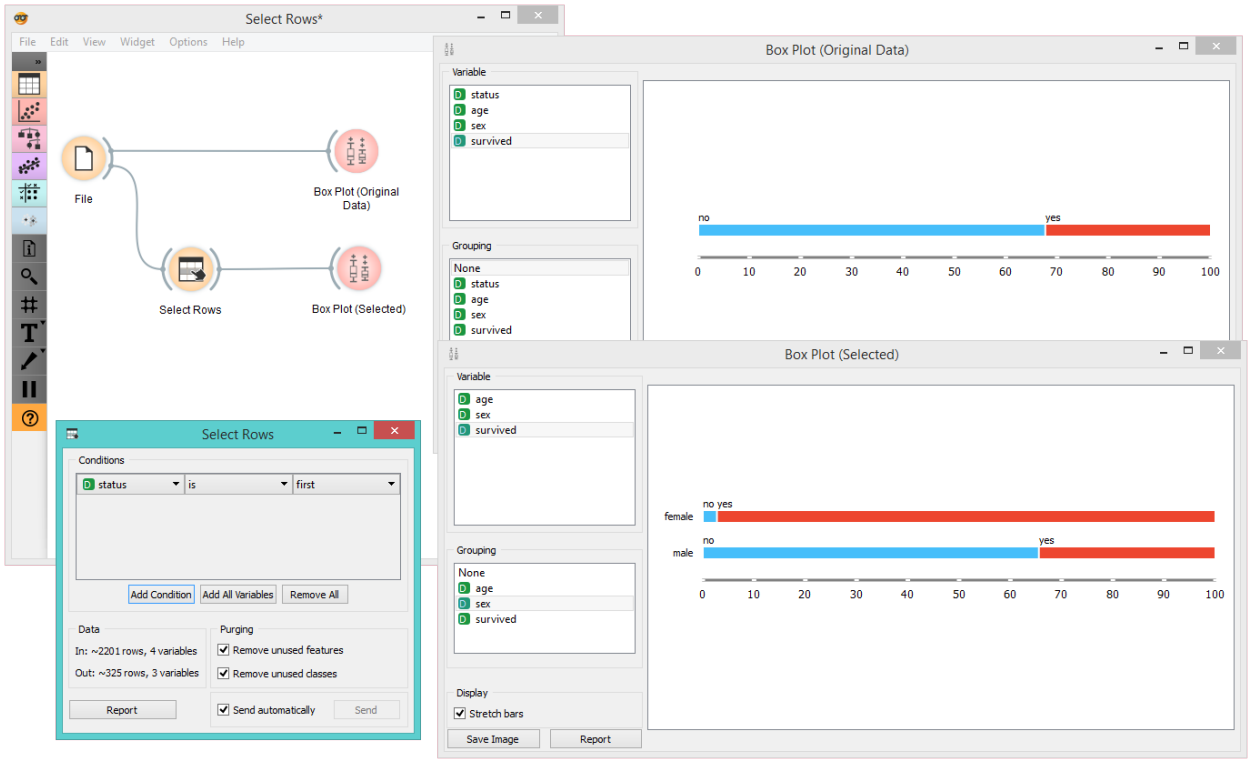
If *Send automatically* is selected, then the output is updated on any change in the composition of the condition or any of its terms.

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 dataset and the dataset with selected rows in the Data Table widget.



In the next example, we used the data from the *Titanic* dataset and similarly fed it into the Box Plot widget. We first observed the entire dataset 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.



2.1.9 Data Sampler

Selects a subset of data instances from an input dataset.

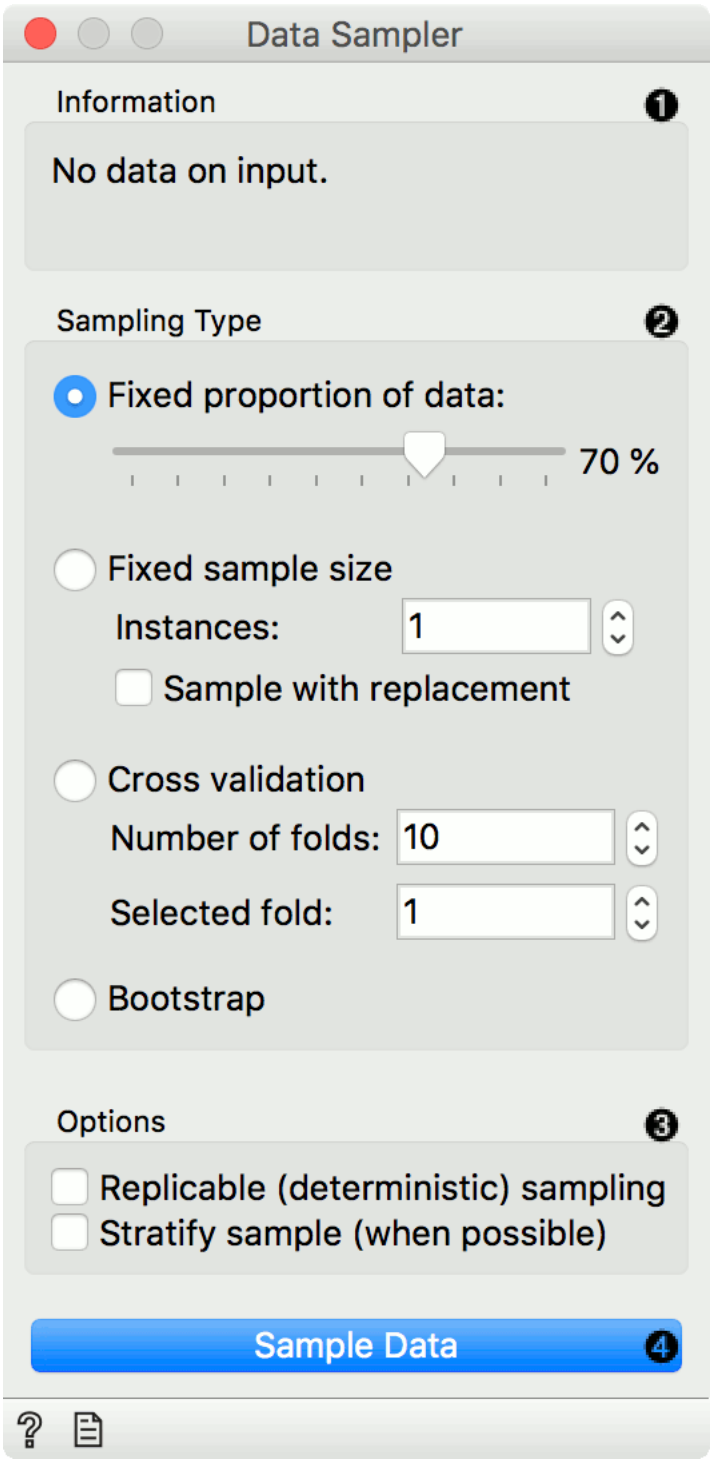
Inputs

- Data: input dataset

Outputs

- Data Sample: sampled data instances
- Remaining Data: out-of-sample data

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



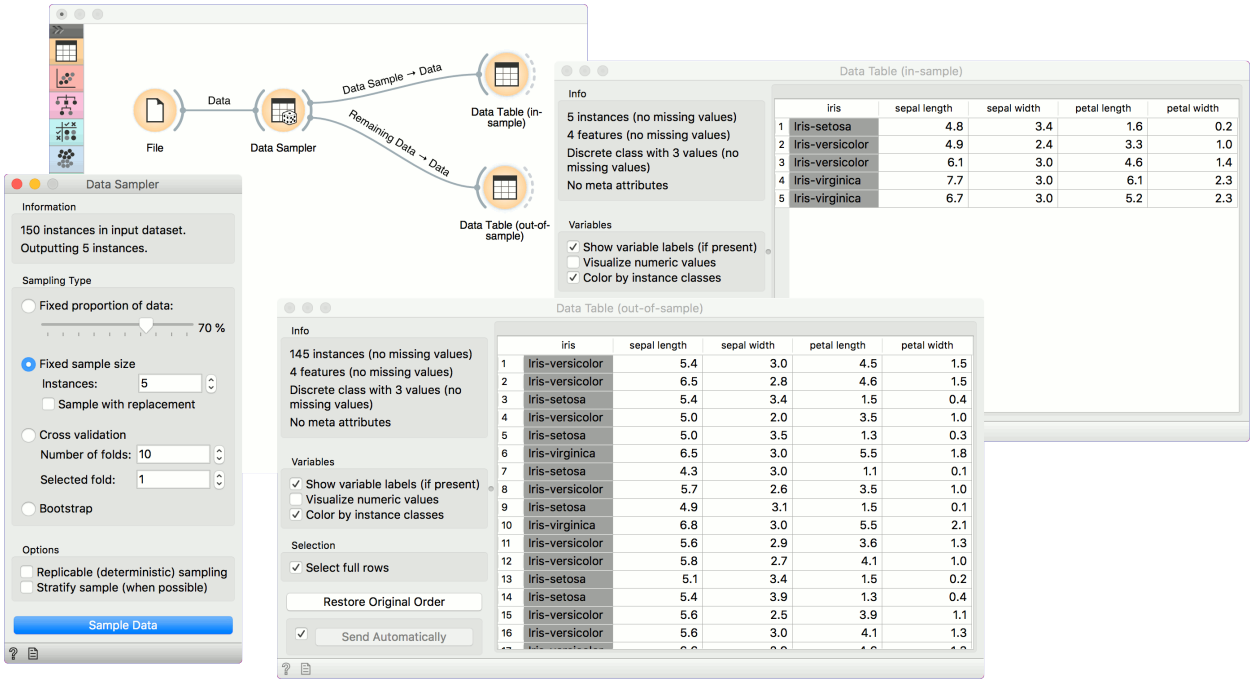
1. Information on the input and output dataset.
2. The desired sampling method:
   - **Fixed proportion of data** returns a selected percentage of the entire data (e.g. 70% of all the data)
   - **Fixed sample size** returns a selected number of data instances with a chance to set *Sample with replacement*, which always samples from the entire dataset (does not subtract instances already in the subset). With replacement, you can generate more instances than available in the input dataset.

- 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.
- Bootstrap infers the sample from the population statistic.

3. *Replicable sampling* maintains sampling patterns that can be carried across users, while *stratify sample* mimics the composition of the input dataset.
4. Press *Sample Data* to output the data sample.

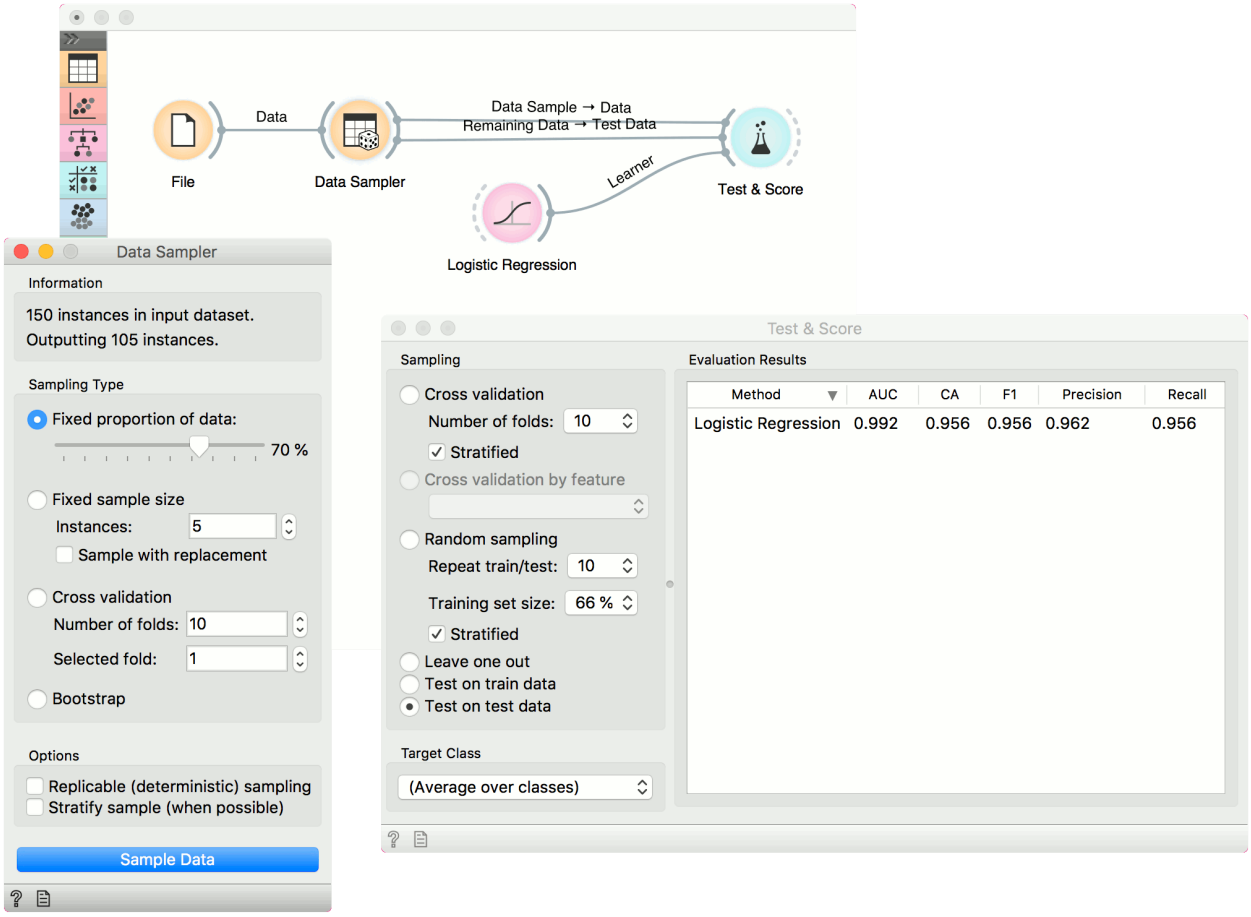
Examples

First, let's see how the **Data Sampler** works. We will use the *iris* data from the File widget. We see there are 150 instances in the data. 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 (Data Table (in-sample)). The second Data Table (Data Table (out-of-sample)) shows the remaining 145 instances that weren't in the sample. To output the out-of-sample data, double-click the connection between the widgets and rewire the output to *Remaining Data –> Data*.



Now, we will use the **Data Sampler** to split the data into training and testing part. We are using the *iris* data, which we loaded with the File widget. In **Data Sampler**, we split the data with *Fixed proportion of data*, keeping 70% of data instances in the sample.

Then we connected two outputs to the Test & Score widget, *Data Sample –> Data* and *Remaining Data –> Test Data*. Finally, we added Logistic Regression as the learner. This runs logistic regression on the Data input and evaluates the results on the Test Data.

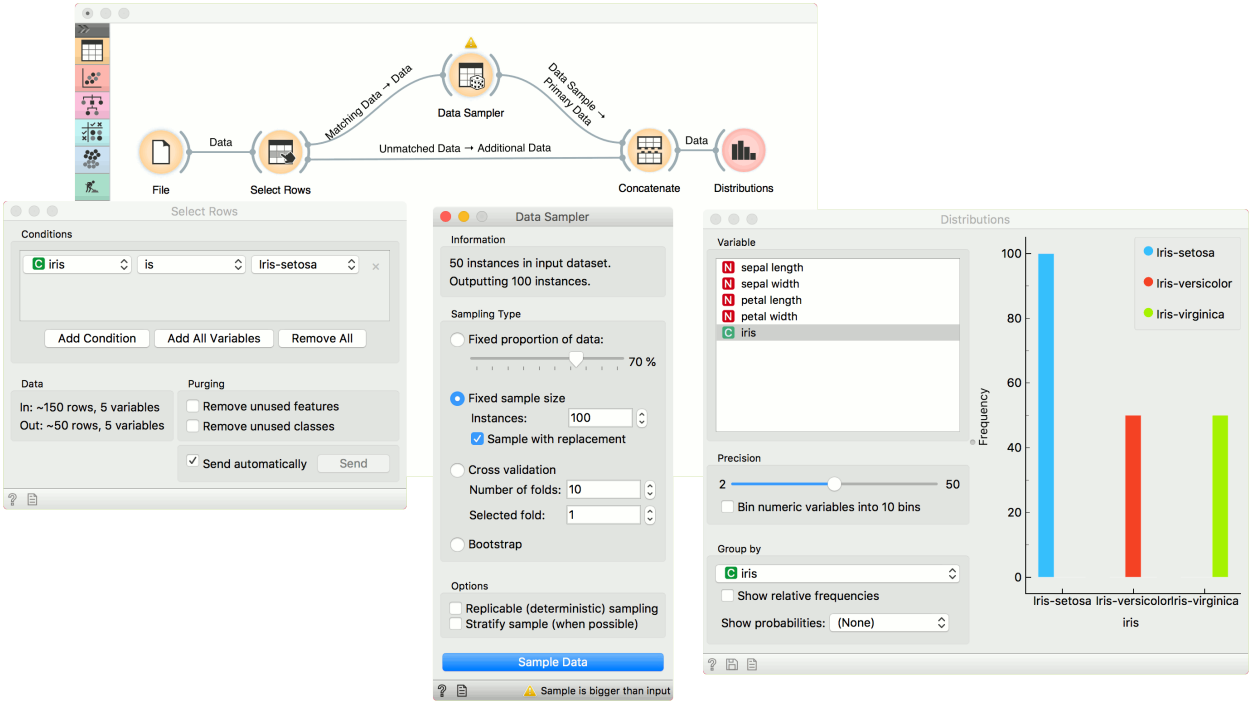


Over/Undersampling

Data Sampler can also be used to oversample a minority class or undersample majority class in the data. Let us show an example for oversampling. First, separate the minority class using a Select Rows widget. We are using the *iris* data from the File widget. The data set has 150 data instances, 50 of each class. Let us oversample, say, *iris-setosa*.

In **Select Rows**, set the condition to *iris is iris-setosa*. This will output 50 instances of the *iris-setosa* class. Now, connect *Matching Data* into the **Data Sampler**, select *Fixed sample size*, set it to, say, 100 and select *Sample with replacement*. Upon pressing *Sample Data*, the widget will output 100 instances of *iris-setosa* class, some of which will be duplicated (because we used *Sample with replacement*).

Finally, use Concatenate to join the oversampled instances and the *Unmatched Data* output of the **Select Rows** widget. This outputs a data set with 200 instances. We can observe the final results in the Distributions.



2.1.10 Transpose

Transposes a data table.

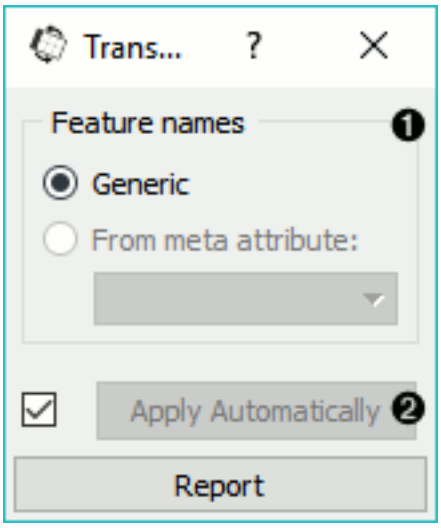
Inputs

- Data: input dataset

Outputs

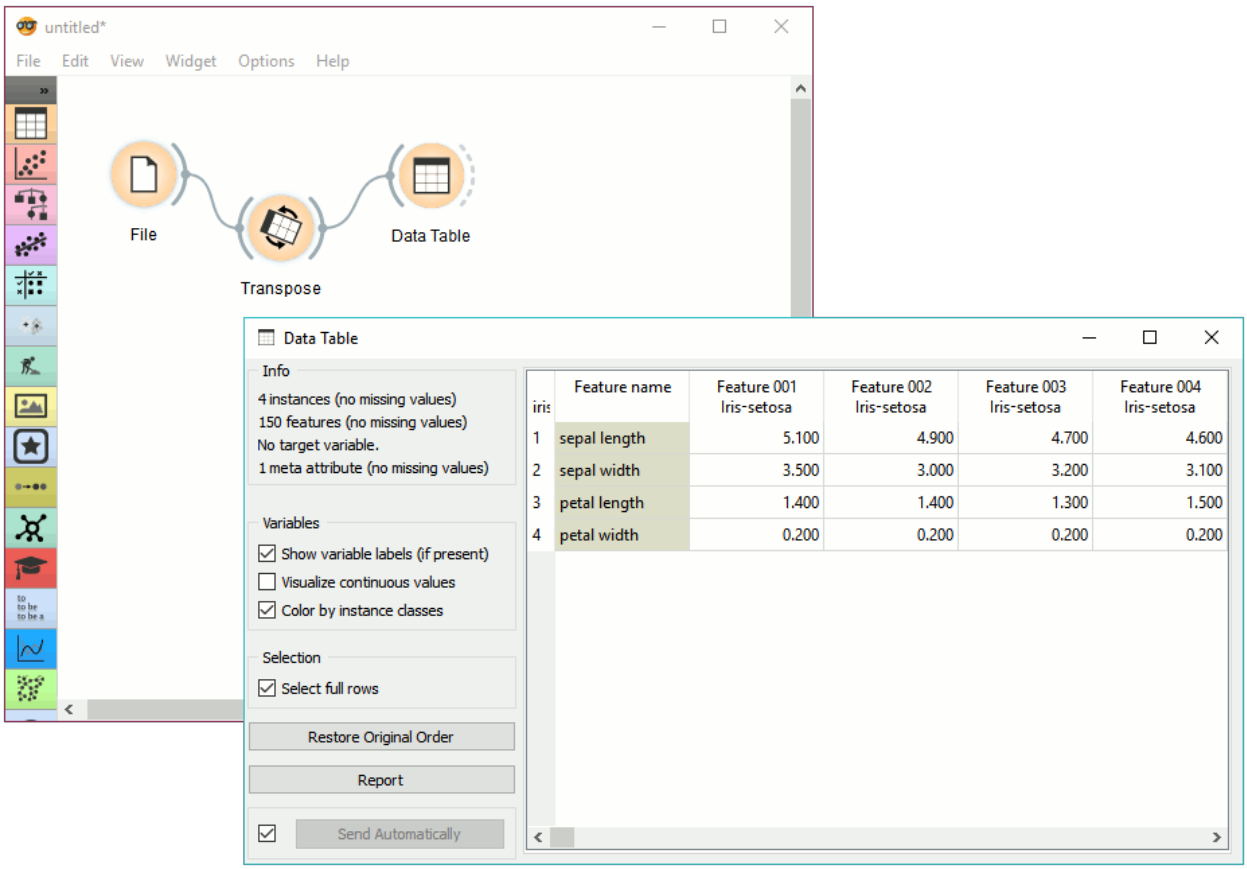
- Data: transposed dataset

Transpose widget transposes data table.



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.



2.1.11 Discretize

Discretizes continuous attributes from an input dataset.

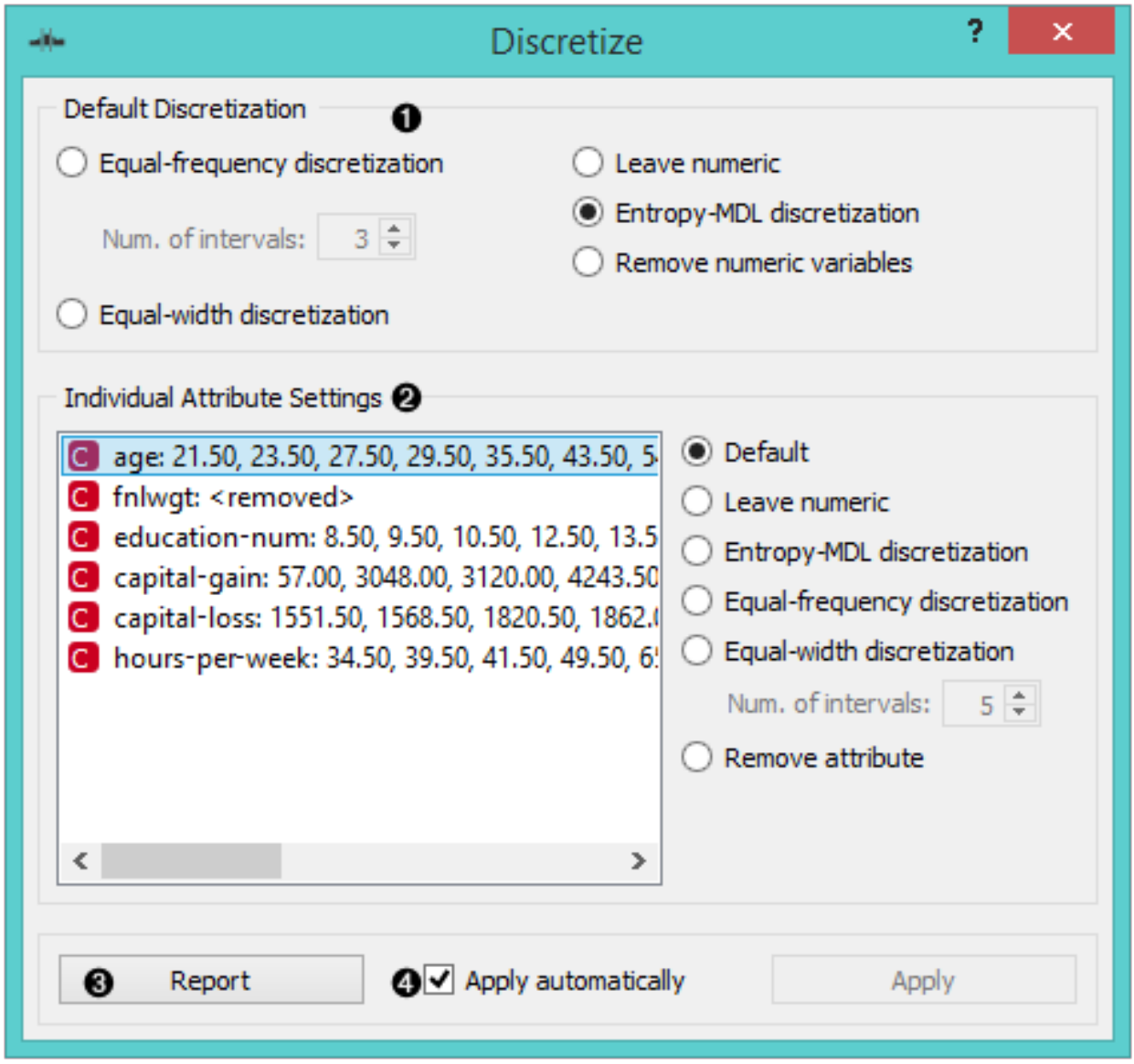
Inputs

- Data: input dataset

Outputs

- Data: dataset with discretized values

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



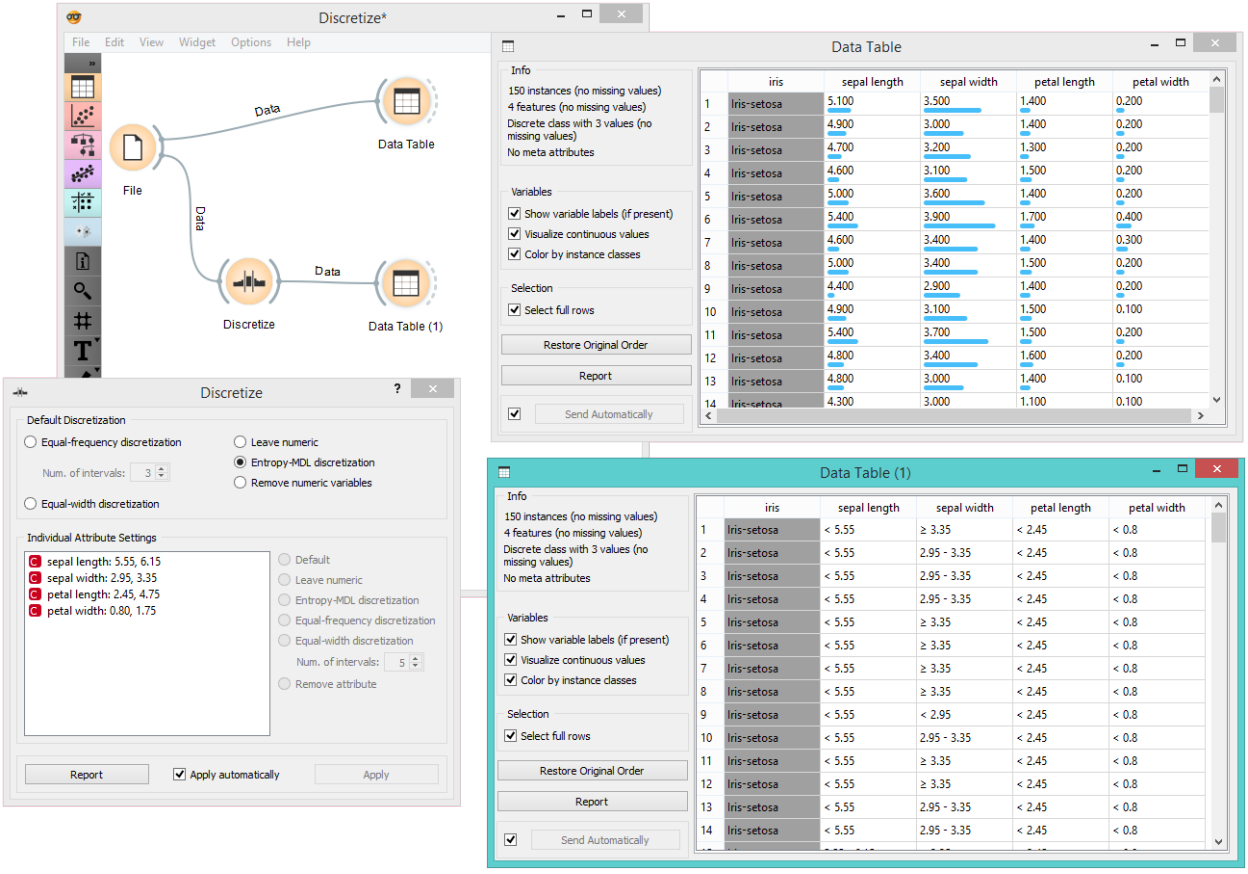
1. The basic version of the widget is rather simple. It allows choosing between three different discretizations.
   - Entropy-MDL, invented by Fayyad and Irani is a top-down discretization, which recursively splits the attribute at a cut maximizing information gain, until the gain is lower than the minimal 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.
2. 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.

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

Example

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



2.1.12 Continuize

Turns discrete attributes into continuous dummy variables.

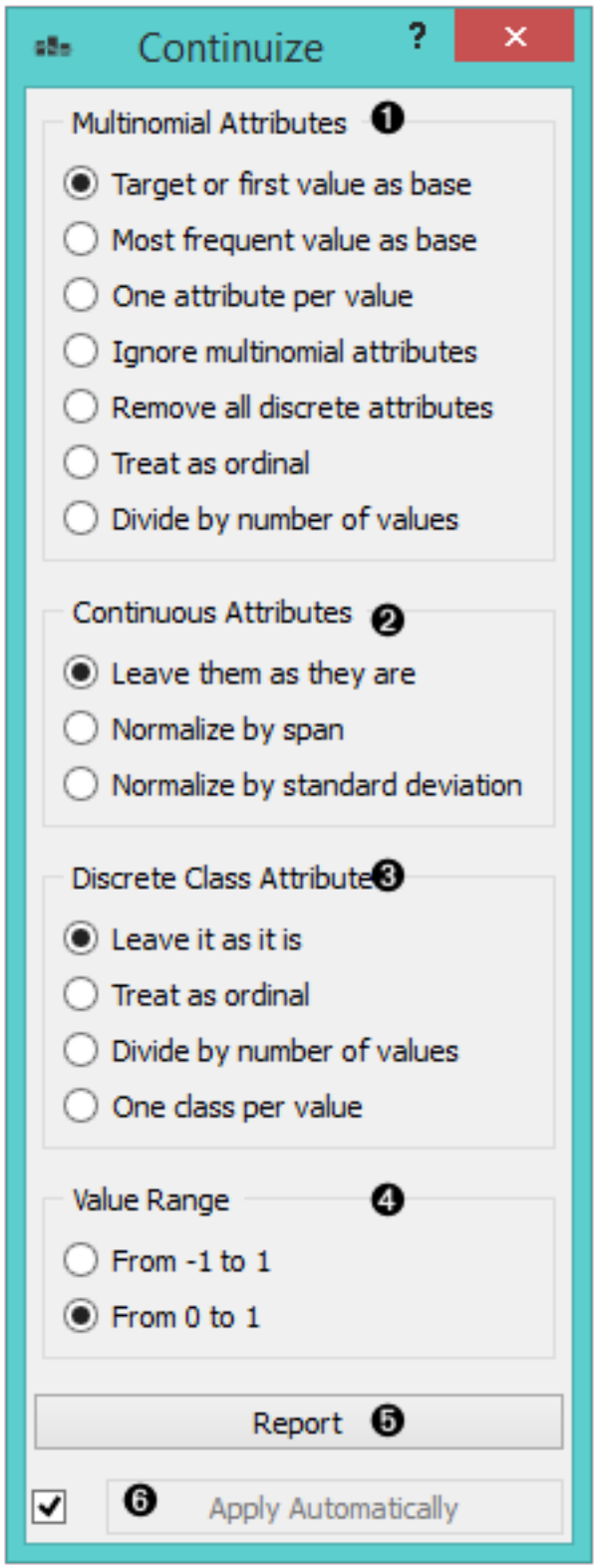
Inputs

- Data: input data set

Outputs

- Data: data set with continuized instances

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

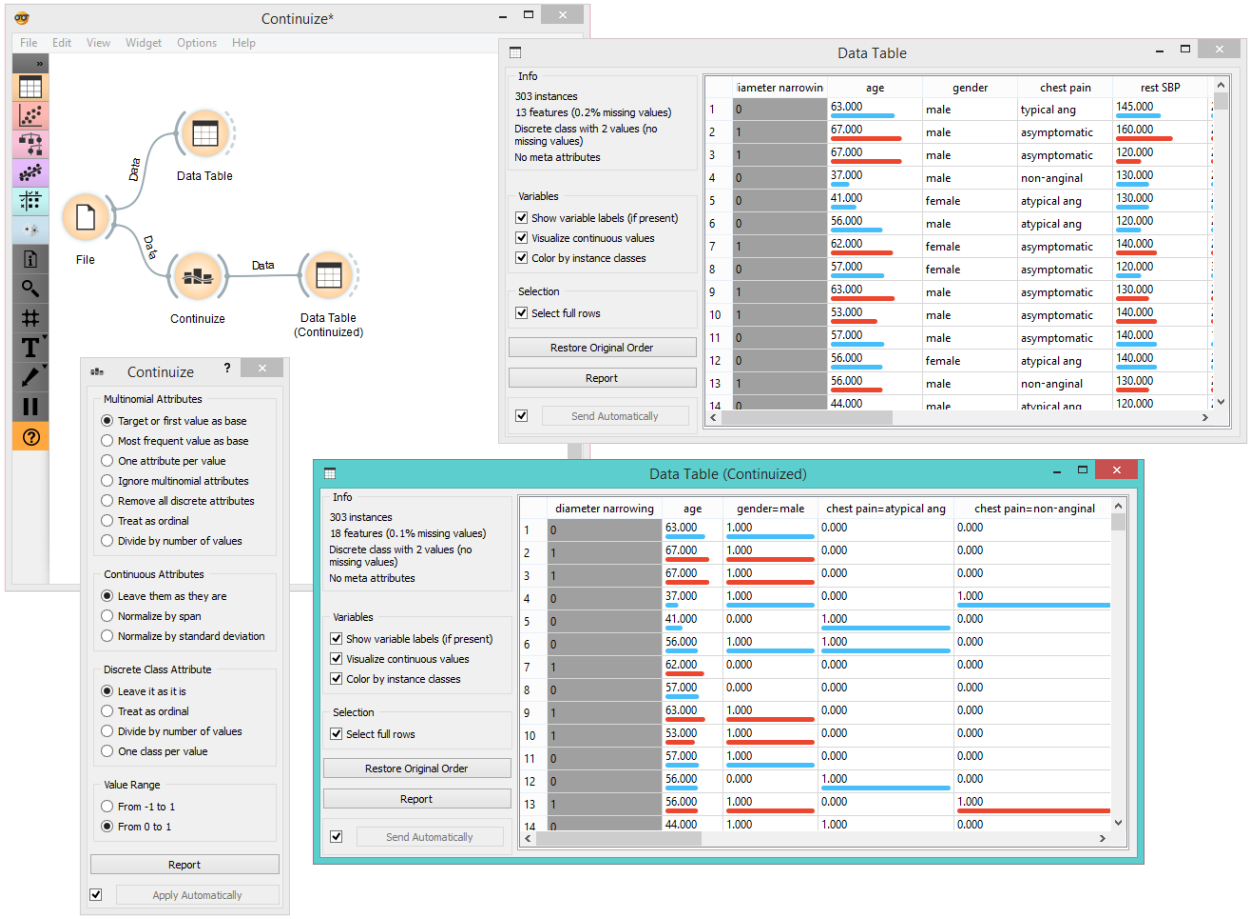
discrete attribute status with the values low, middle and high, listed in that order. Options for their transformation are:

- **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.

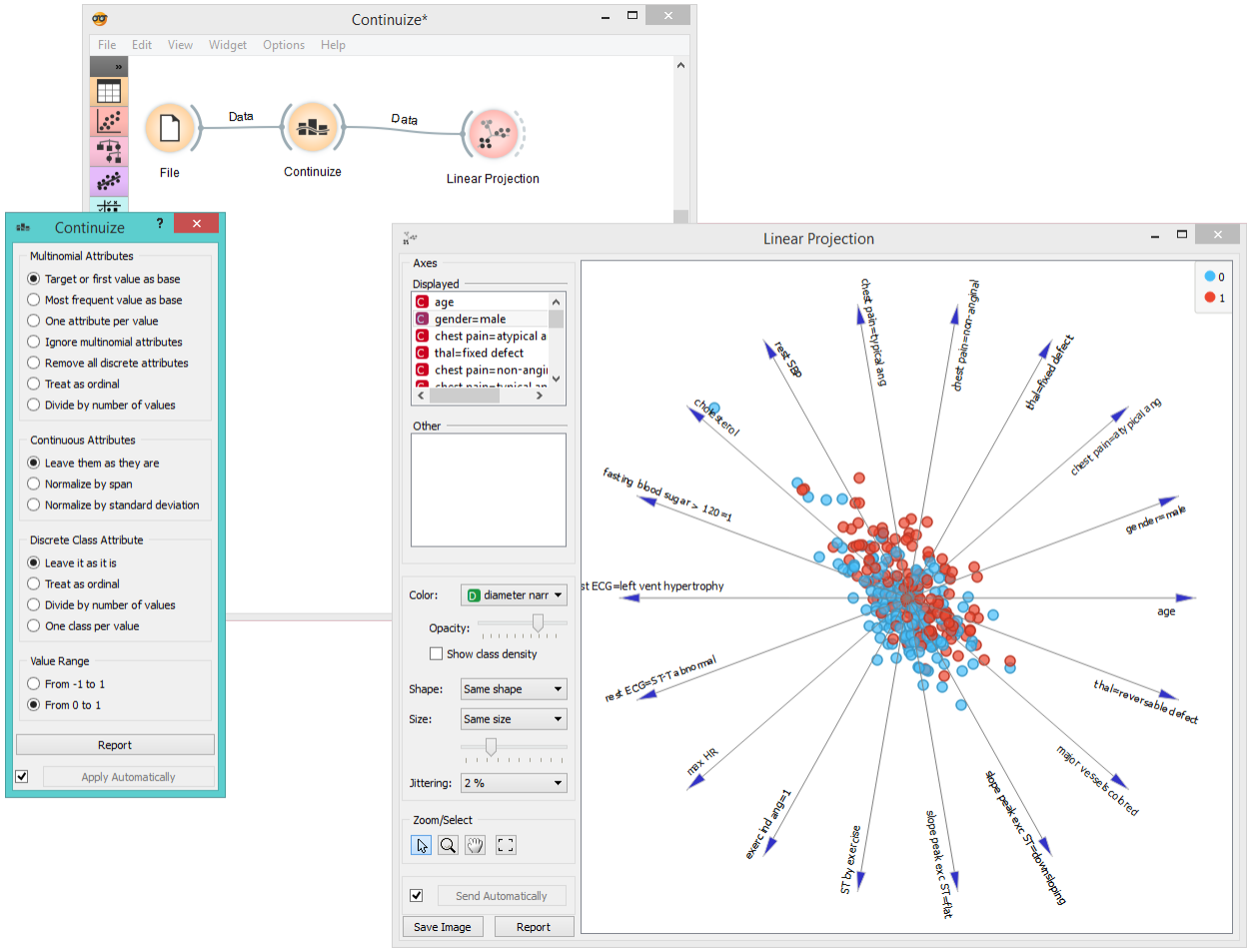
2. Define the treatment of continuous attributes. You will usually prefer the *Leave them as they are* option. The alternative is *Normalize by span*, which will subtract the lowest value found in the data and divide by the span, so all values will fit into [0, 1]. Finally,*Normalize by standard deviation* 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.
6. 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 **Continuize** widget. We feed the original data (the *Heart disease* data set) into the Data Table and see how they look like. Then we continuize the discrete values and observe them in another Data Table.



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 **Continuize** 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*".



2.1.13 Create Class

Create class attribute from a string attribute.

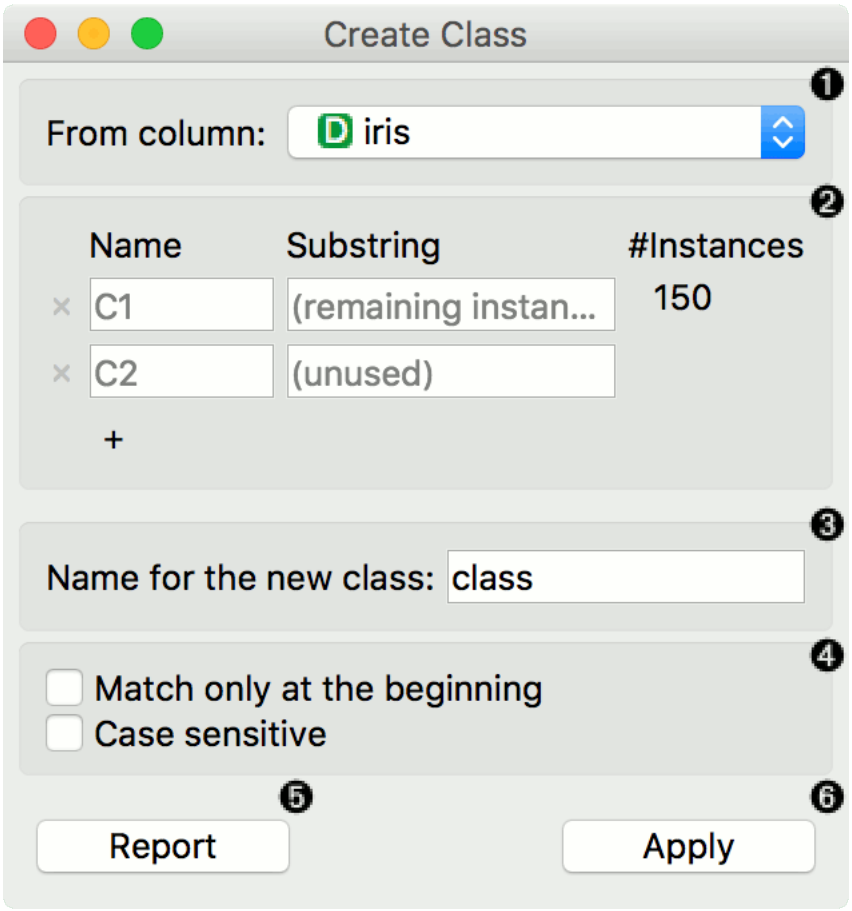
Inputs

- Data: input dataset

Outputs

- Data: dataset with a new class variable

Create Class creates a new class attribute from an existing discrete or string attribute. The widget matches the string value of the selected attribute and constructs a new user-defined value for matching instances.

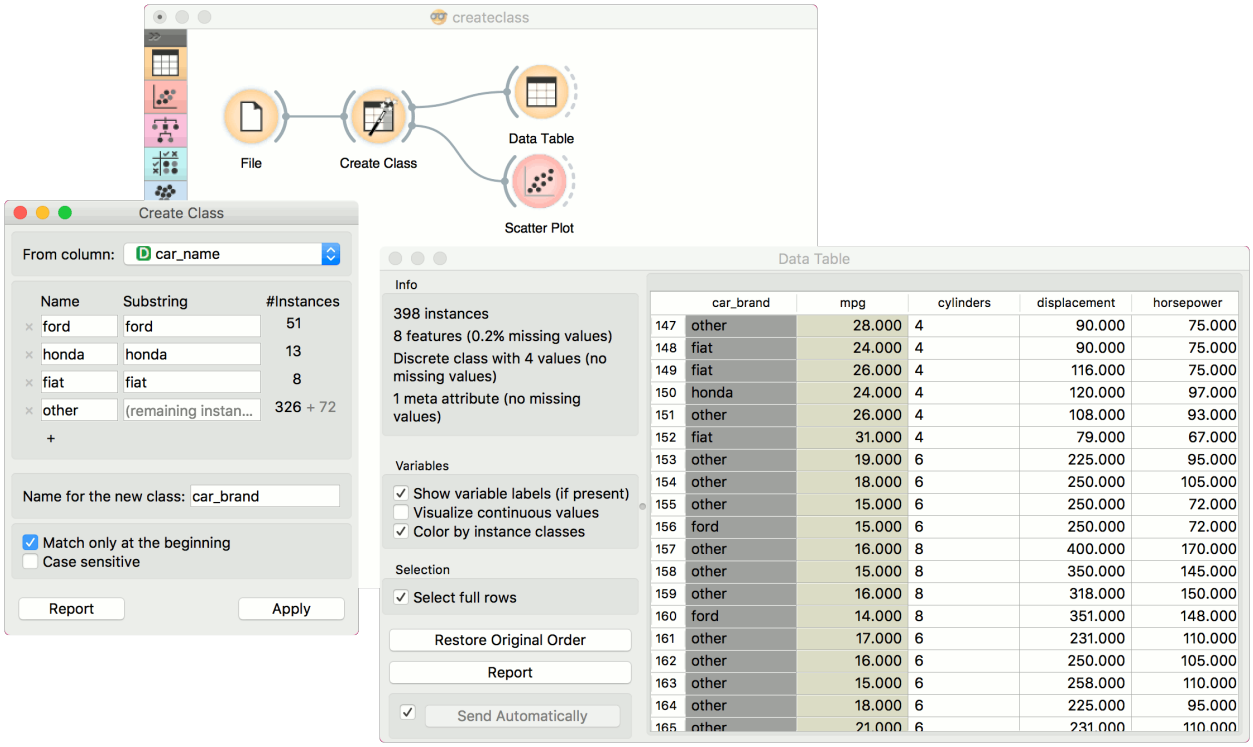


1. The attribute the new class is constructed from.
2. Matching:
   - Name: the name of the new class value
   - Substring: regex-defined substring that will match the values from the above-defined attribute
   - Instances: the number of instances matching the substring
   - Press '+' to add a new class value
3. Name of the new class column.
4. Match only at the beginning will begin matching from the beginning of the string. Case sensitive will match by case, too.
5. Produce a report.
6. Press *Apply* to commit the results.

Example

Here is a simple example with the *auto-mpg* dataset. Pass the data to **Create Class**. Select *car_name* as a column to create the new class from. Here, we wish to create new values that match the car brand. First, we type *ford* as the new value for the matching strings. Then we define the substring that will match the data instances. This means that all instances containing *ford* in their *car_name*, will now have a value *ford* in the new class column. Next, we define the same for *honda* and *fiat*. The widget will tell us how many instance are yet unmatched (remaining instances). We will name them *other*, but you can continue creating new values by adding a condition with '+'.

We named our new class column *car_brand* and we matched at the beginning of the string.



Finally, we can observe the new column in a Data Table or use the value as color in the Scatter Plot.

2.1.14 Randomize

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

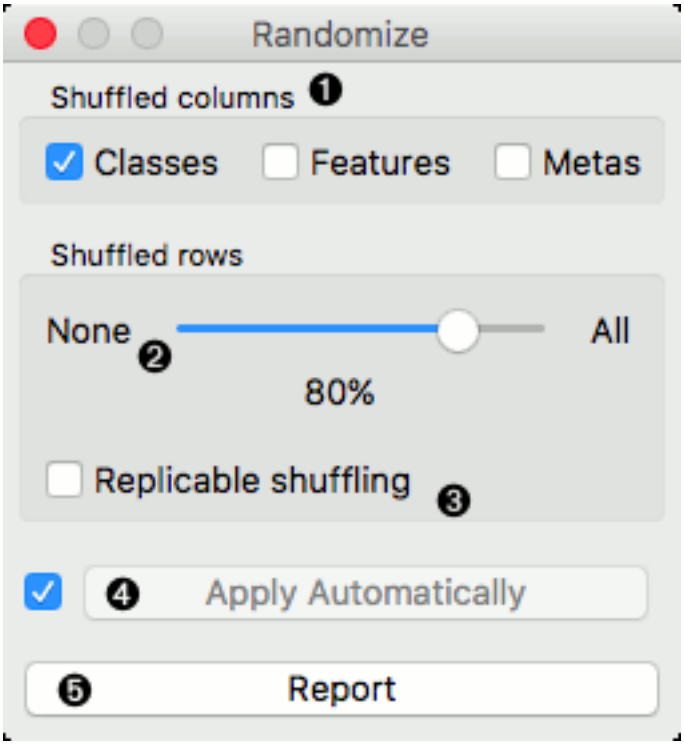
Inputs

- Data: input dataset

Outputs

- Data: randomized dataset

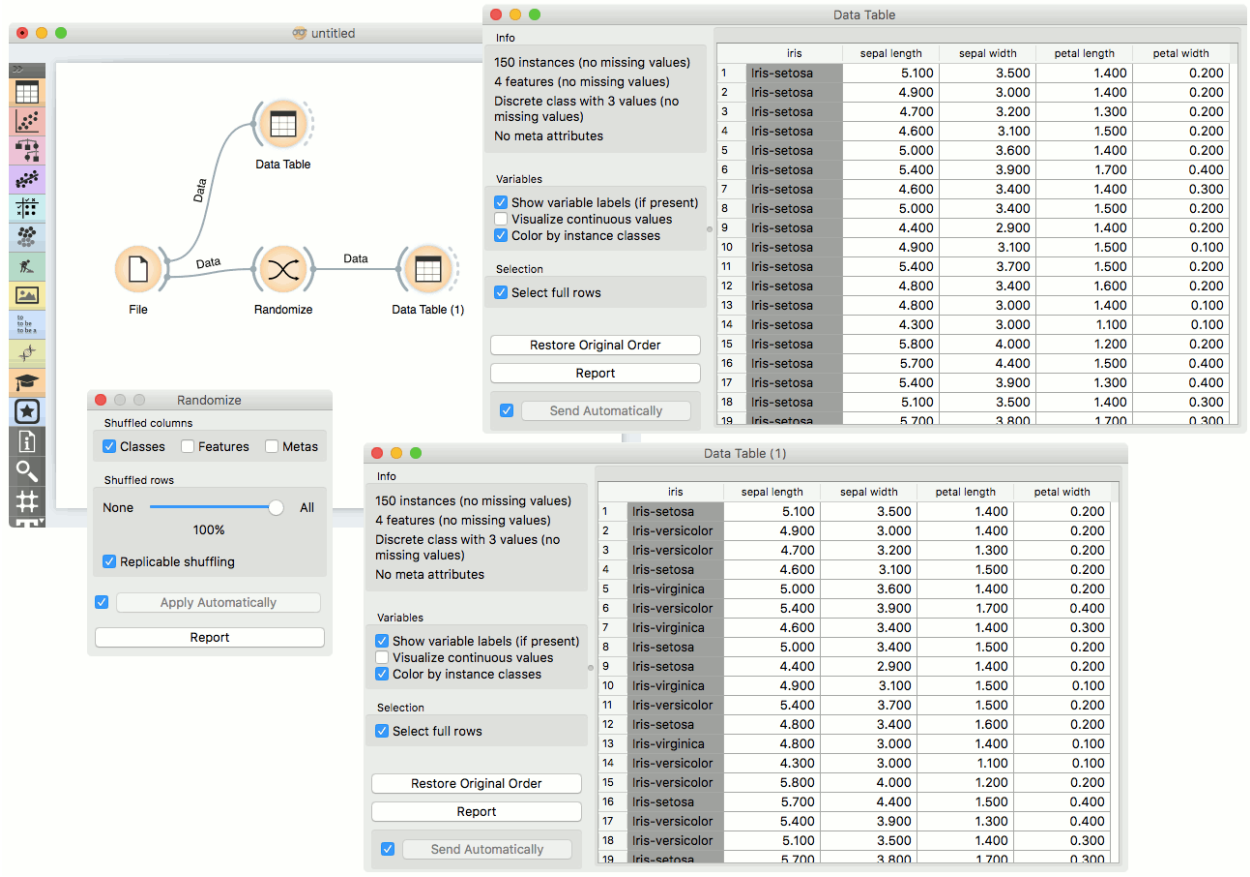
The **Randomize** widget receives a dataset in the input and outputs the same dataset in which the classes, attributes or/and metas are shuffled.



1. Select group of columns of the dataset you want to shuffle.
2. Select proportion of the dataset 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

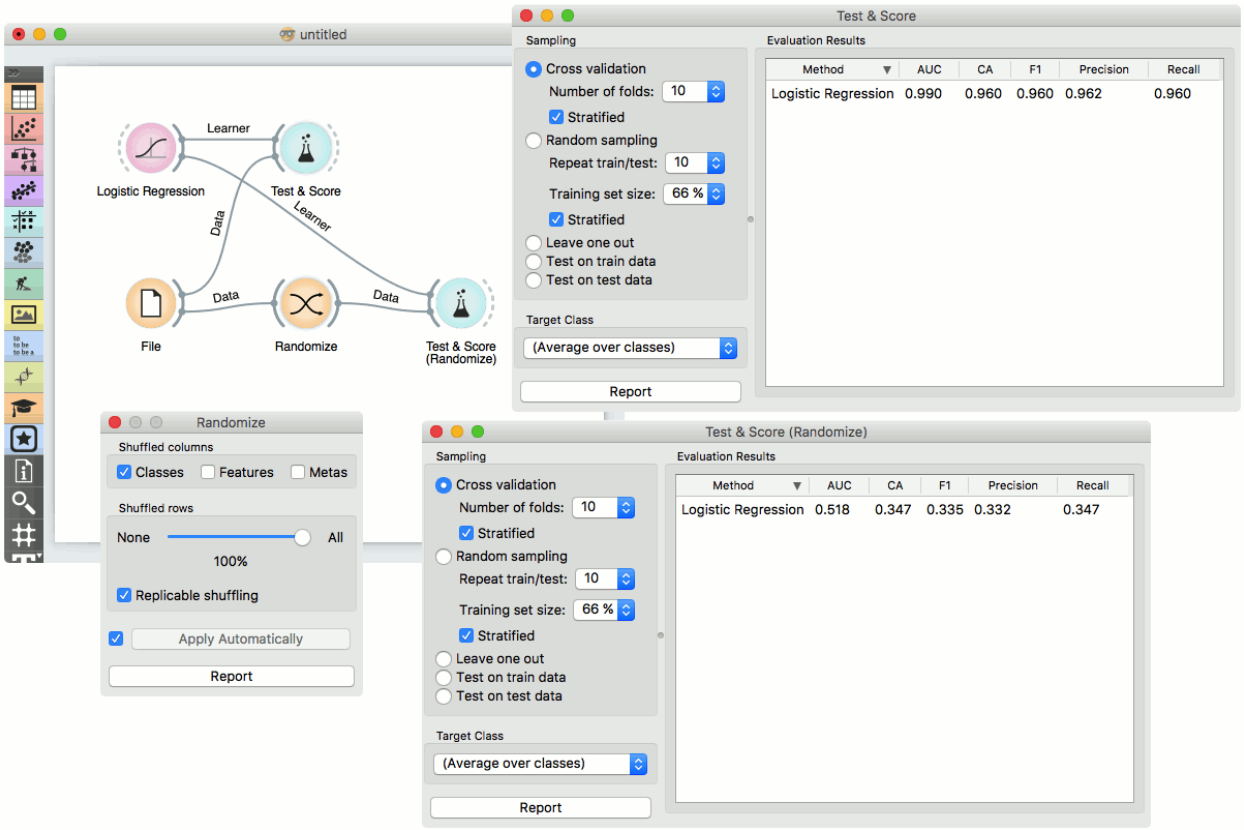
The **Randomize** widget is usually placed right after (e.g. File widget. The basic usage is shown in the following workflow, where values of class variable of Iris dataset are randomly shuffled.



	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

	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

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



2.1.15 Concatenate

Concatenates data from multiple sources.

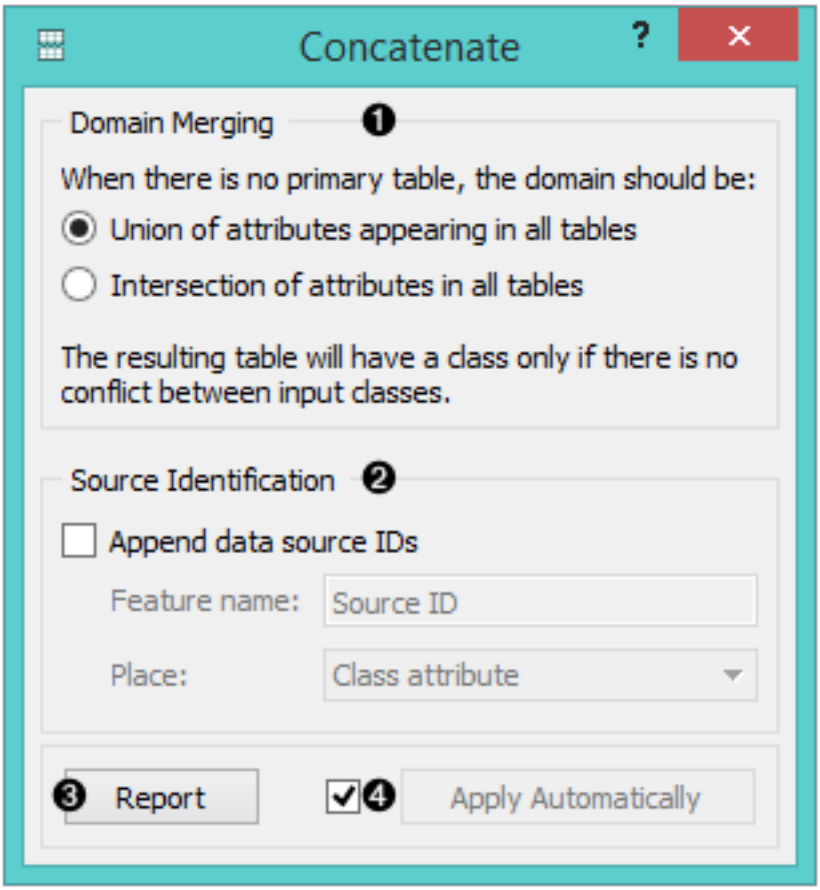
Inputs

- Primary Data: data set that defines the attribute set
- Additional Data: additional data set

Outputs

- Data: concatenated data

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.



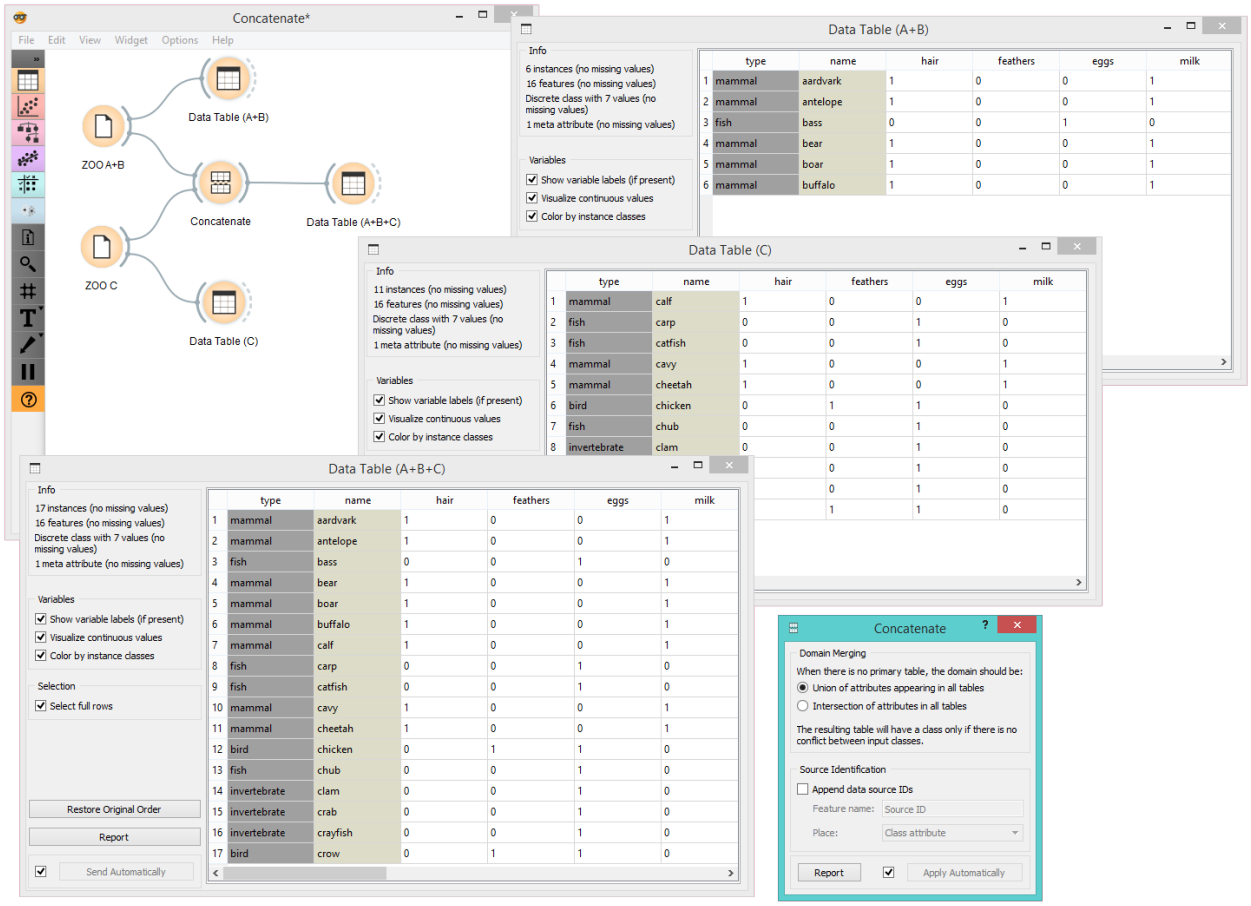
1. Set the attribute merging method.
2. Add the identification of source data sets to the output data set.
3. Produce a report.
4. If *Apply automatically* is ticked, changes are communicated automatically. Otherwise, click *Apply*.

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.



2.1.16 Select by Data Index

Match instances by index from data subset.

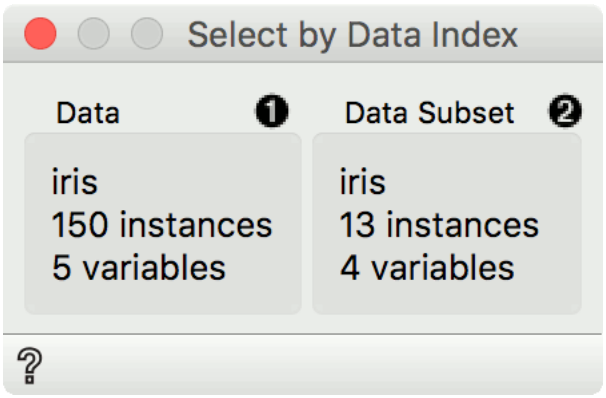
Inputs

- Data: reference data set
- Data Subset: subset to match

Outputs

- Matching data: subset from reference data set that matches indices from subset data
- Unmatched data: subset from reference data set that does not match indices from subset data
- Annotated data: reference data set with an additional column defining matches

Select by Data Index enables matching the data by indices. Each row in a data set has an index and given a subset, this widget can match these indices to indices from the reference data. Most often it is used to retrieve the original data from the transformed data (say, from PCA space).



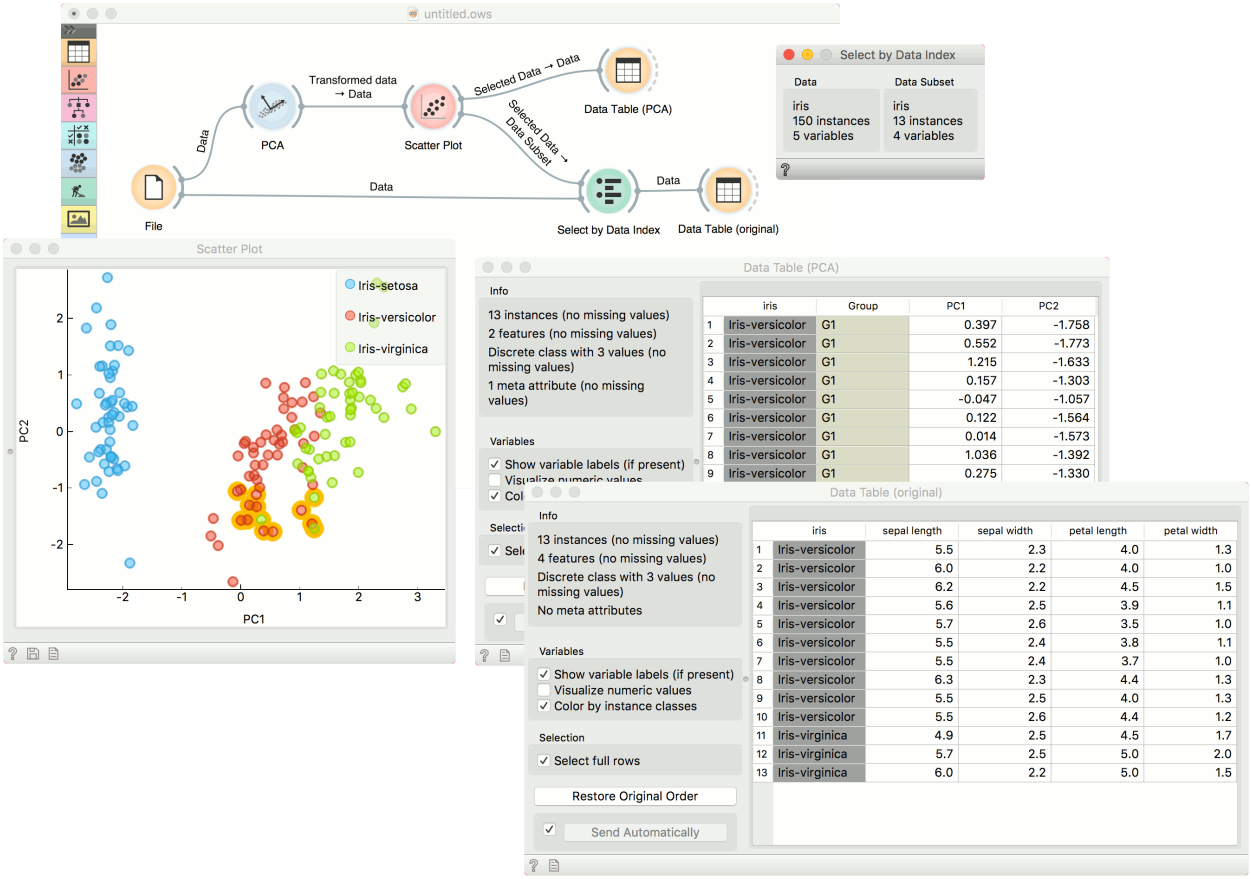
1. Information on the reference data set. This data is used as index reference.
2. Information on the data subset. The indices of this data set are used to find matching data in the reference data set. Matching data are on the output by default.

Example

A typical use of **Select by Data Index** is to retrieve the original data after a transformation. We will load *iris.tab* data in the File widget. Then we will transform this data with PCA. We can project the transformed data in a Scatter Plot, where we can only see PCA components and not the original features.

Now we will select an interesting subset (we could also select the entire data set). If we observe it in a Data Table, we can see that the data is transformed. If we would like to see this data with the original features, we will have to retrieve them with **Select by Data Index**.

Connect the original data and the subset from Scatter Plot to **Select by Data Index**. The widget will match the indices of the subset with the indices of the reference (original) data and output the matching reference data. A final inspection in another Data Table confirms the data on the output is from the original data space.



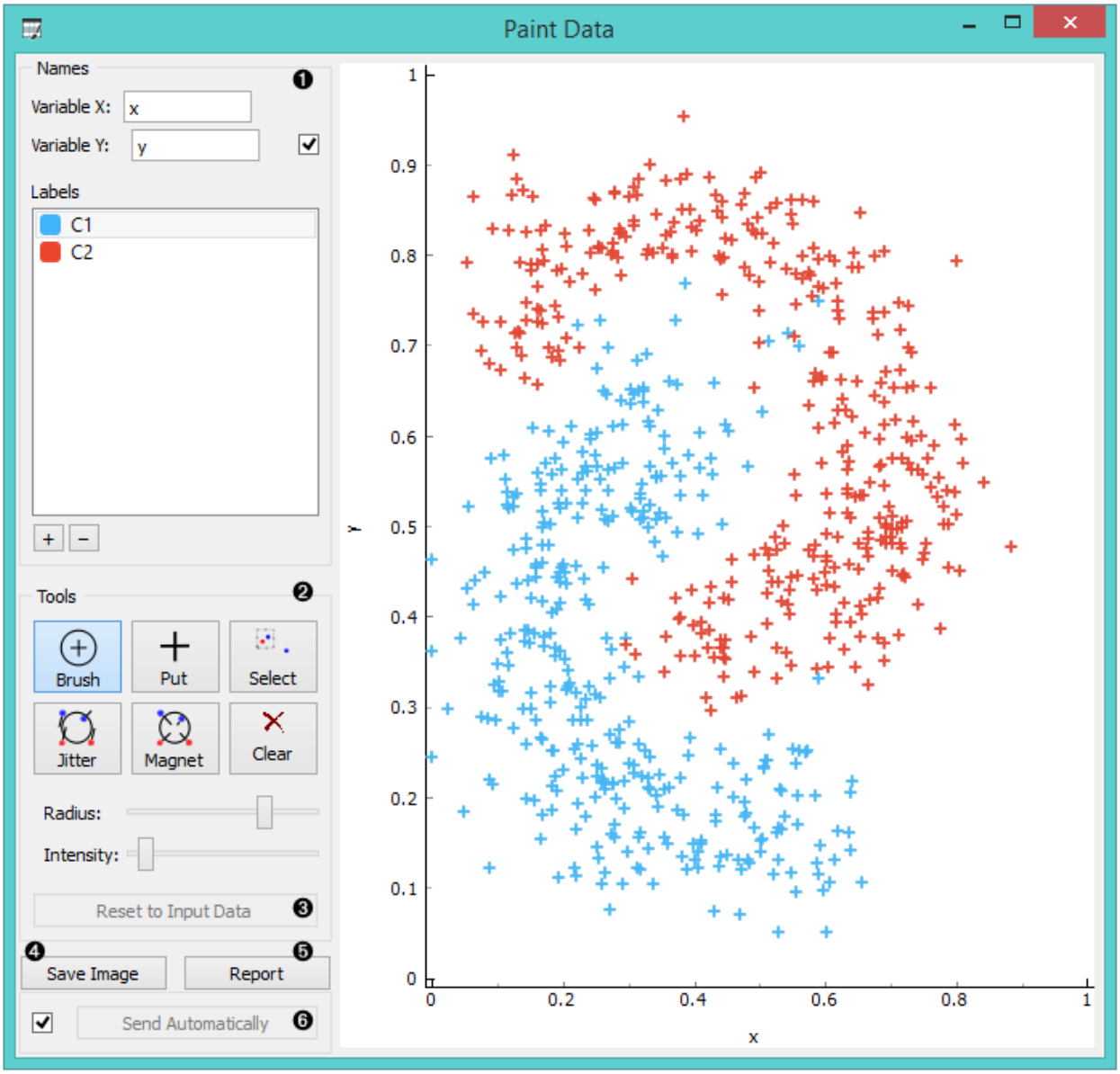
2.1.17 Paint Data

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

Outputs

- Data: dataset as painted in the plot

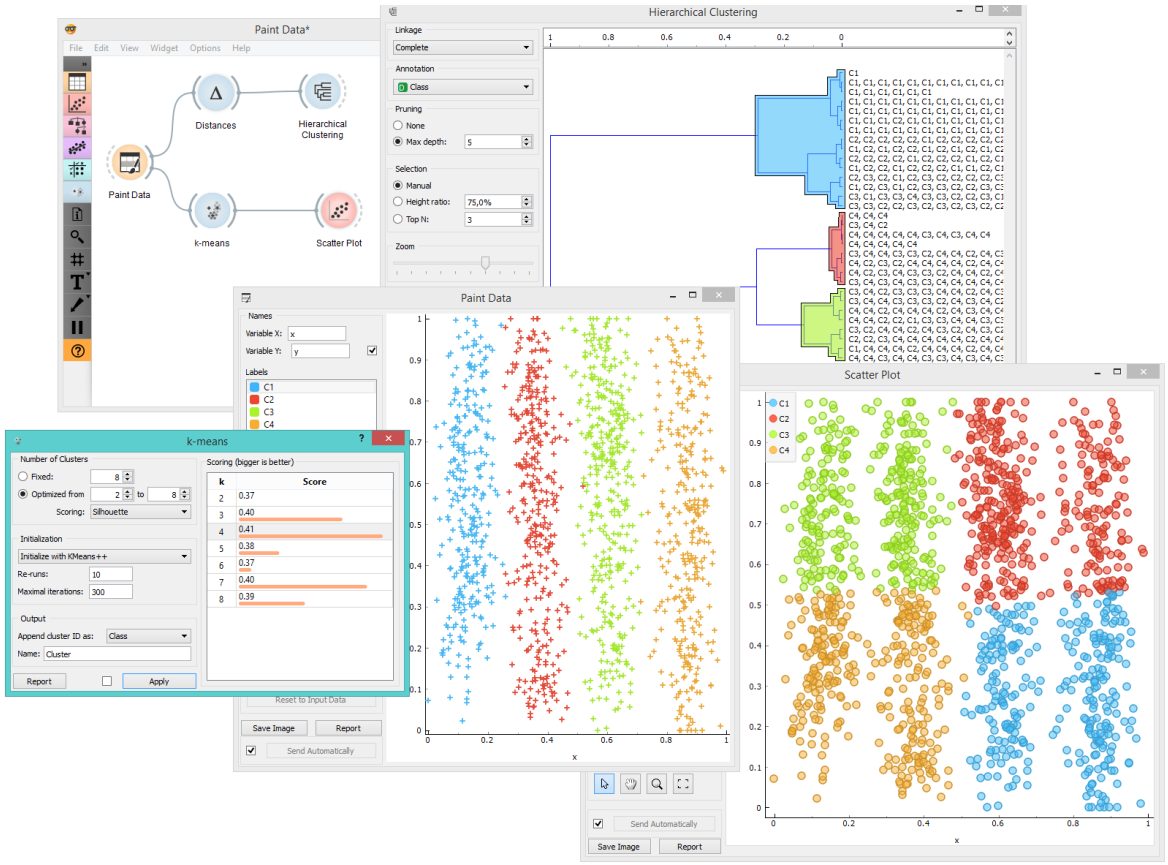
The widget supports the creation of a new dataset by visually placing data points on a two-dimension plane. Data points can be placed on the plane individually (*Put*) or in a larger number by brushing (*Brush*). Data points can belong to classes if the data is intended to be used in supervised learning.



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 datasets.
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 dataset with 4 classes. Such dataset 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.



2.1.18 Python Script

Extends functionalities through Python scripting.

Inputs

- Data (Orange.data.Table): input dataset bound to `in_data` variable
- Learner (Orange.classification.Learner): input learner bound to `in_learner` variable
- Classifier (Orange.classification.Learner): input classifier bound to `in_classifier` variable
- Object: input Python object bound to `in_object` variable

Outputs

- Data (Orange.data.Table): dataset retrieved from `out_data` variable
- Learner (Orange.classification.Learner): learner retrieved from `out_learner` variable
- Classifier (Orange.classification.Learner): classifier retrieved from `out_classifier` variable

- Object: Python object retrieved from `out_object` variable

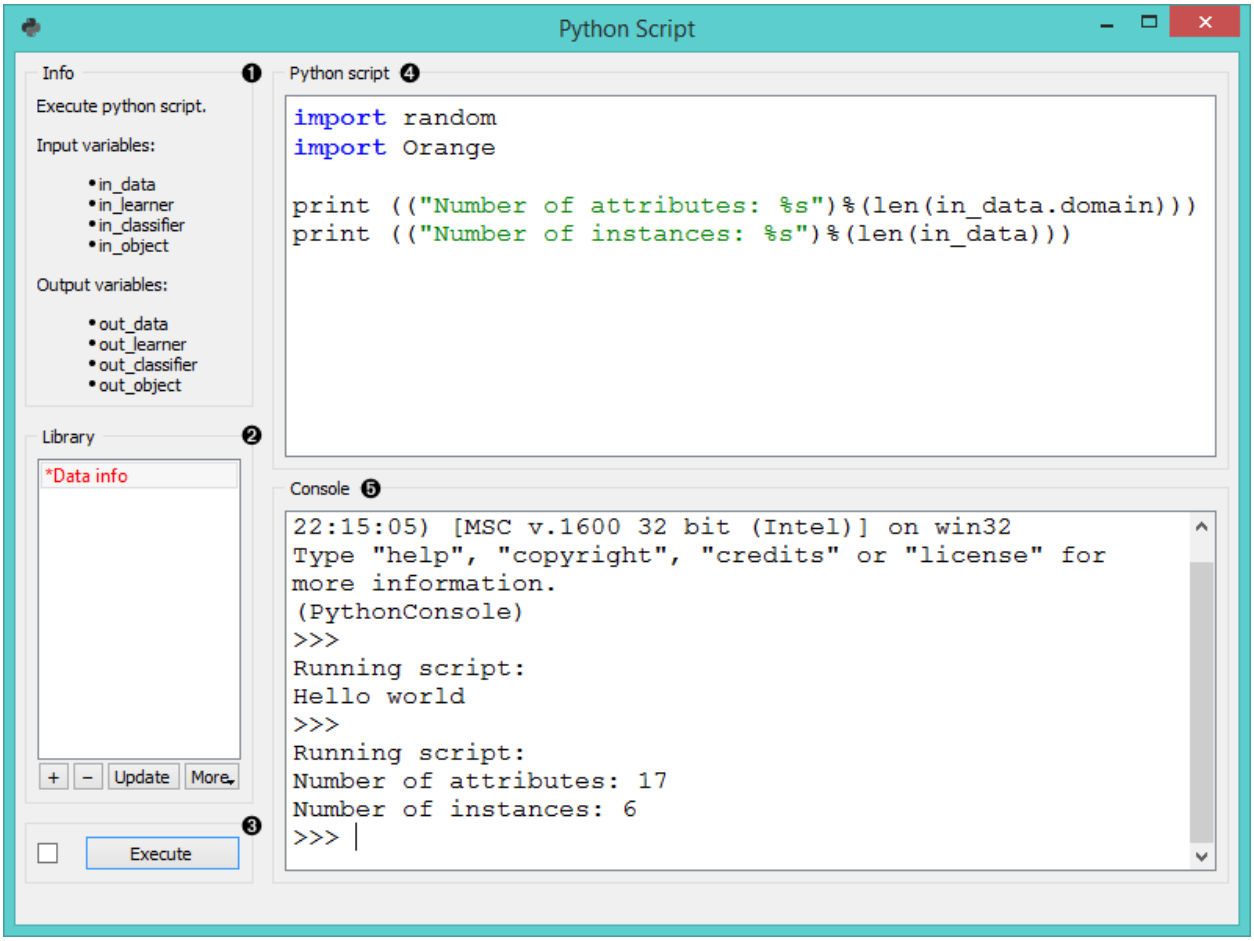
Python Script widget can be used to run a python script in the input, when a suitable functionality is not implemented in an existing widget. The script has `in_data`, `in_distance`, `in_learner`, `in_classifier` and `in_object` variables (from input signals) in its local namespace. If a signal is not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

After the script is executed 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.

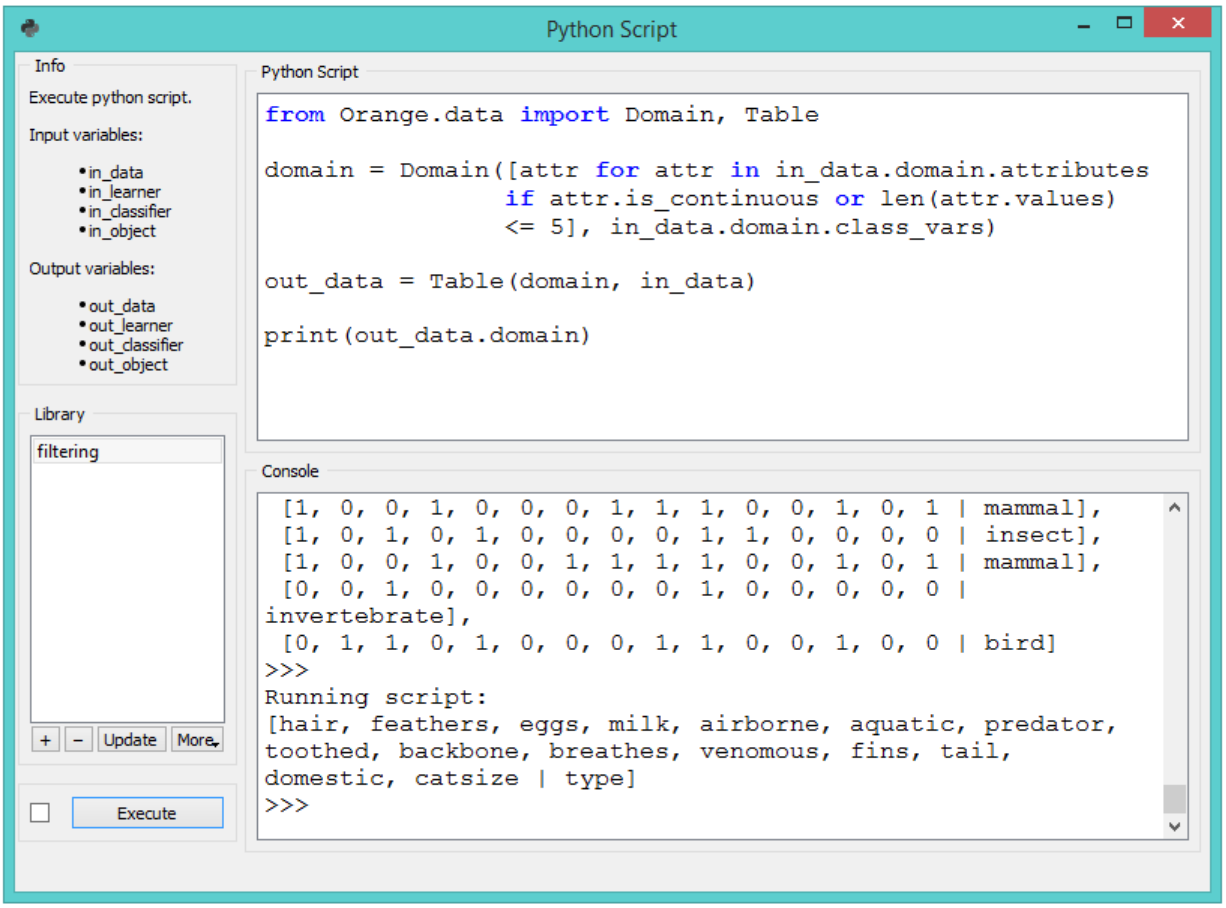
3. Pressing *Execute* in the *Run* box executes the script (keyboard shortcut "Ctrl+R"). Any script output (from `print`) is captured and displayed in the *Console* below the script.
4. The *Python script* editor on the left can be used to edit a script (it supports some rudimentary syntax highlighting).
5. Console displays the output of the script.

Examples

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

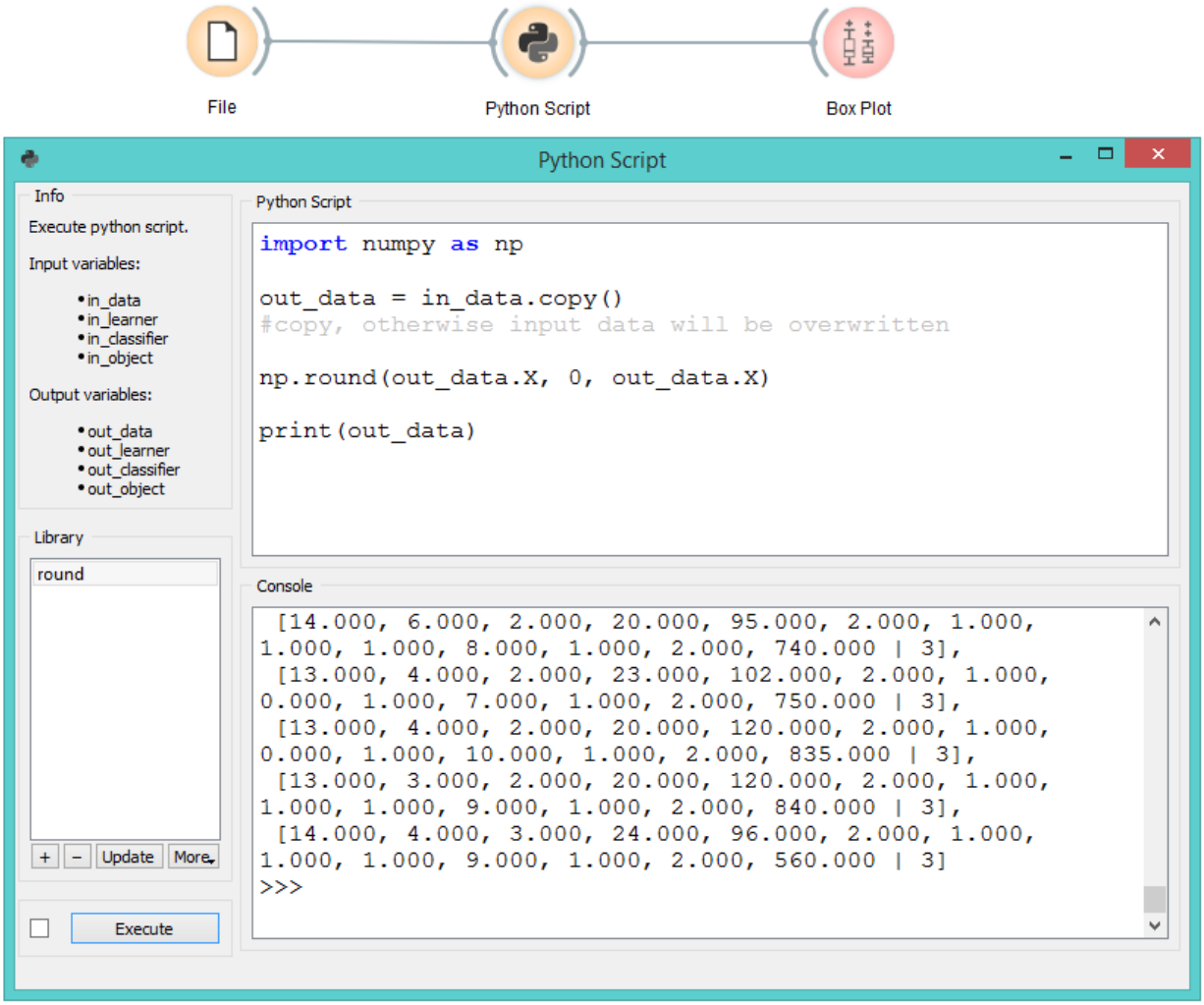
One can, for example, do batch filtering by attributes. We used zoo.tab for the example and we filtered out all the attributes that have more than 5 discrete values. This in our case removed only 'leg' attribute, but imagine an example where one would have many such attributes.

```
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)
```



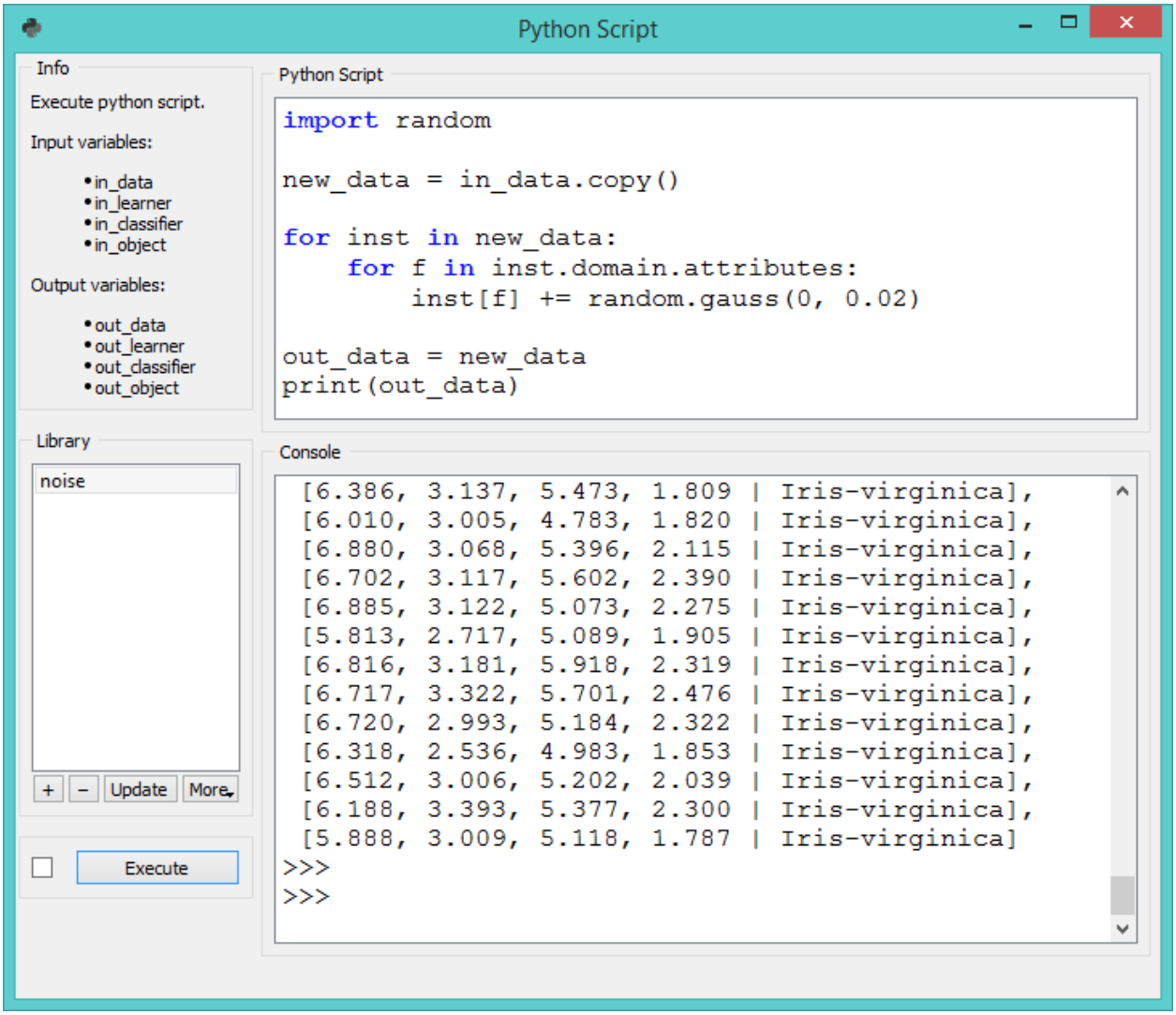
The second example shows how to round all the values in a few lines of code. This time we used wine.tab and rounded all the values to whole numbers.

```
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.

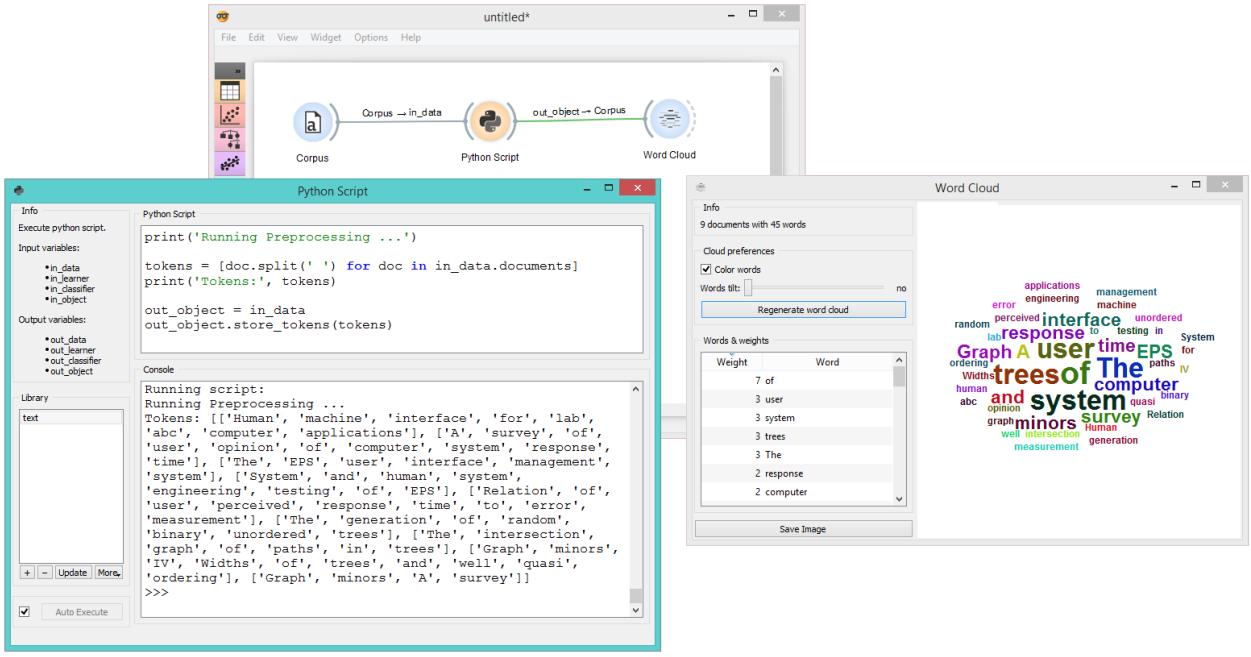
```
import random
from Orange.data import Domain, Table
new_data = in_data.copy()
for inst in new_data:
  for f in inst.domain.attributes:
    inst[f] += random.gauss(0, 0.02)
out_data = new_data
```



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.



2.1.19 Feature Constructor

Add new features to your dataset.

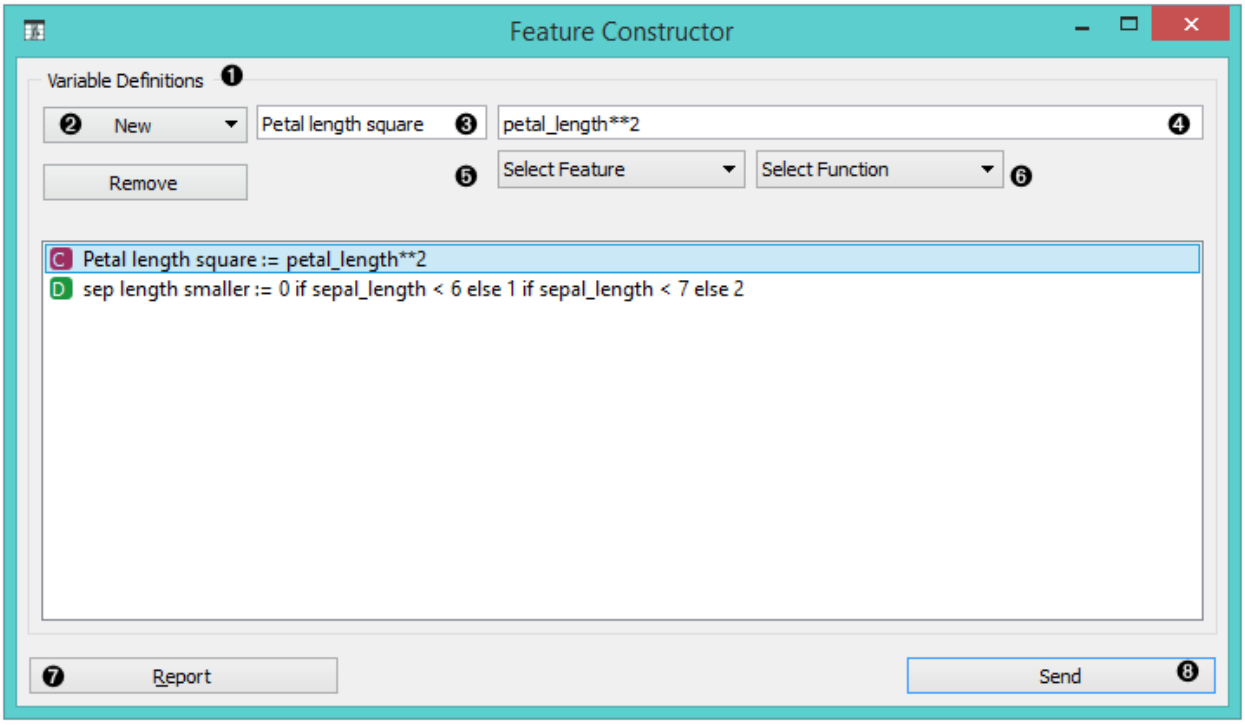
Inputs

- Data: input dataset

Outputs

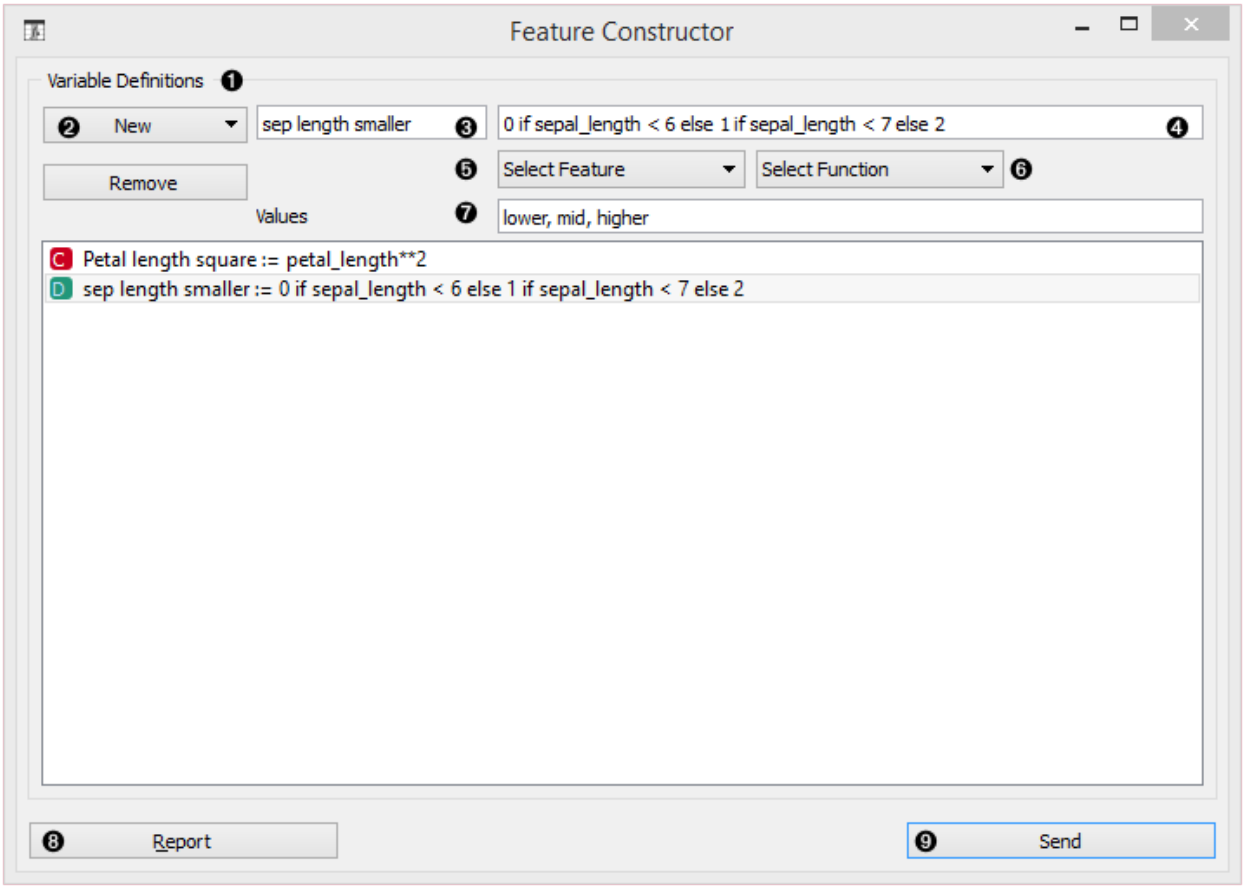
- Data: dataset with additional features

The **Feature Constructor** allows you to manually add features (columns) into your dataset. 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

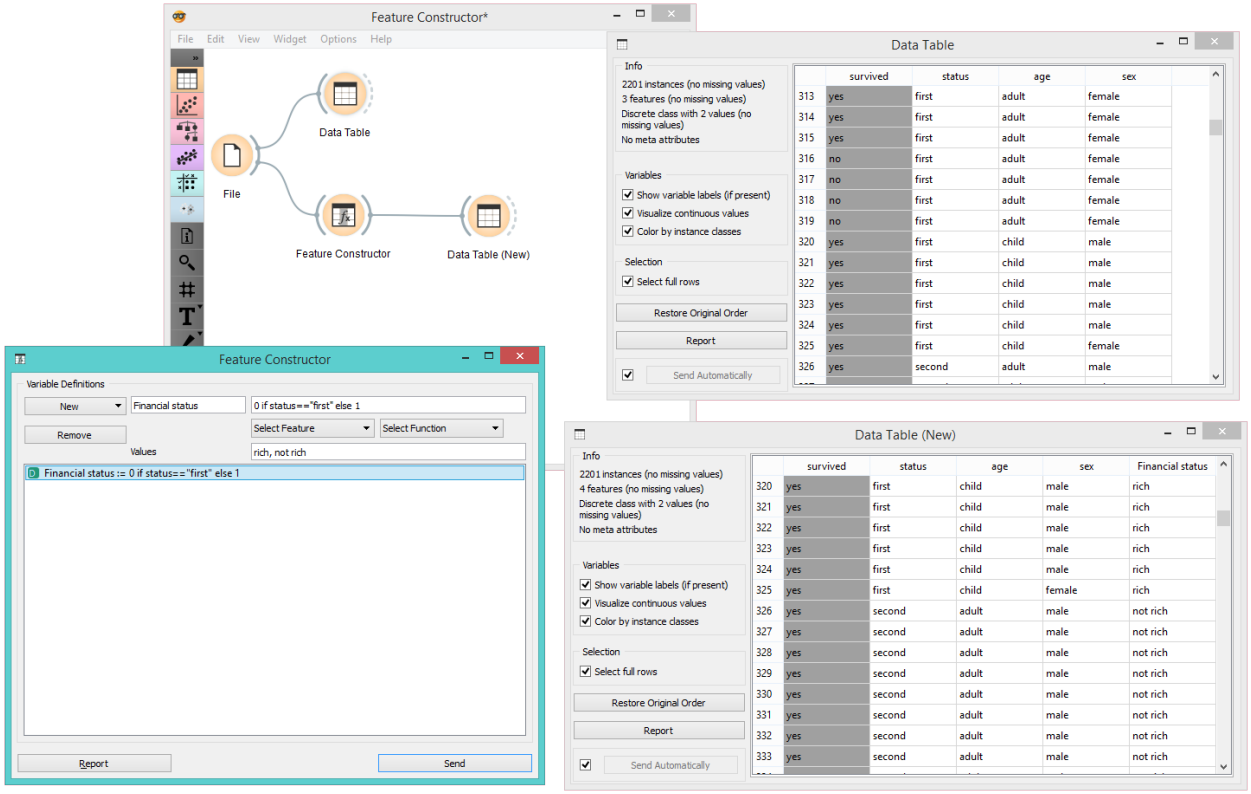
For discrete variables, however, there's a bit more work. First add or remove the values you want for the new feature. Then select the base value and the expression. In the example below, we have constructed an expression with 'if lower than' and defined three conditions; the program ascribes 0 (which 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* dataset. 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 dataset with Data Table widget.



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.

2.1.20 Edit Domain

Rename features and their values.

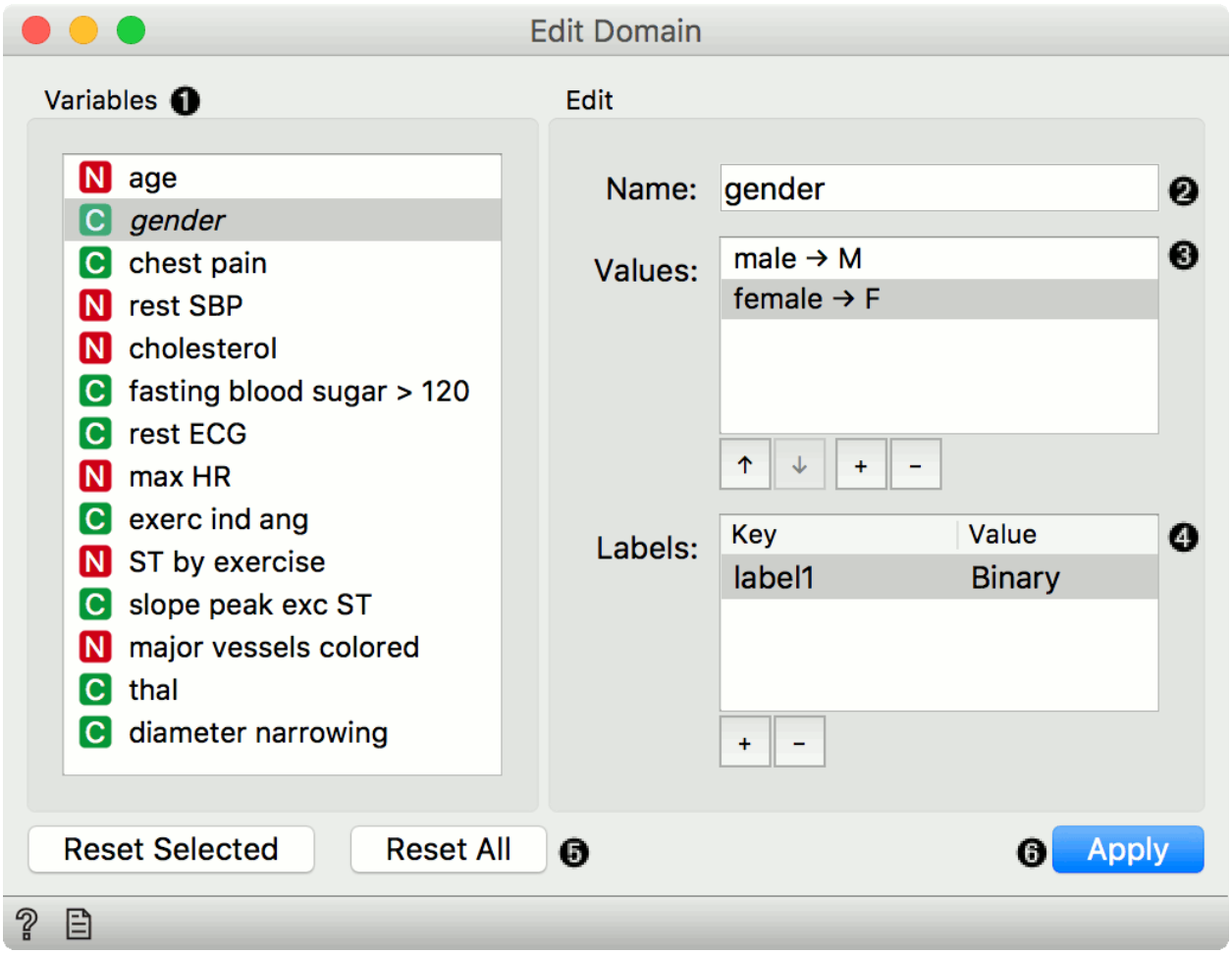
Inputs

- Data: input dataset

Outputs

- Data: dataset with edited domain

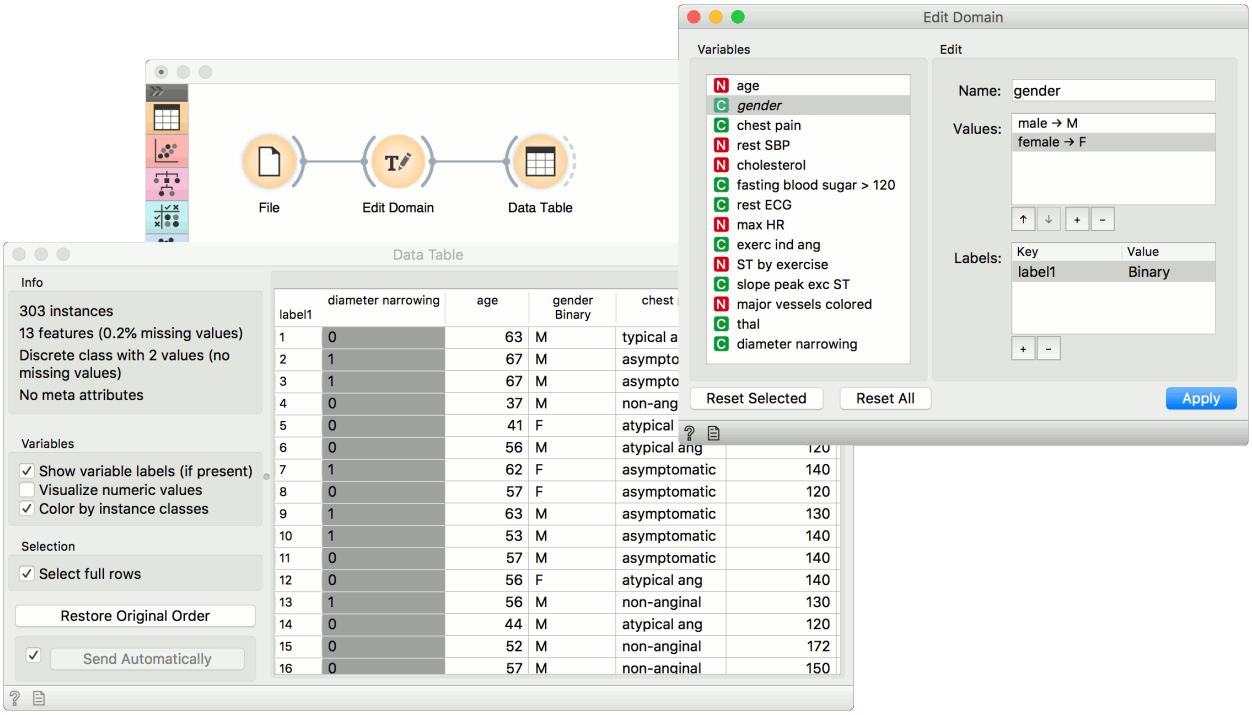
This widget can be used to edit/change a dataset's domain.



1. All features (including meta attributes) from the input dataset are listed in the *Variables* list. Selecting one feature displays an editor on the right.
2. Change the name of the feature.
3. Change the value names for discrete features in the *Values* list box. Double-click to edit the name. To reorder the values (for example to display them in Distributions, use the up and down keys at the bottom of the box. To add or remove a value, use + and - buttons.
4. Additional feature annotations can be added, removed or edited in the *Labels* box. Add a new label with the + button and add the *Key* and *Value* for the new entry. Key will be displayed in the top left corner of the Data Table, while values will appear below the specified column. Remove an existing label with the - button.
5. To revert the changes made to the selected feature, press the *Reset Selected* button while the feature is selected in the *Variables* list. Pressing *Reset All* will remove all the changes to the domain.
6. Press *Apply* to send the new domain to the output.

Example

Below, we demonstrate how to simply edit an existing domain. We selected the *heart_disease.tab* dataset and edited the *gender* attribute. Where in the original we had the values *female* and *male*, we changed it into *F* for female and *M* for male. Then we used the down key to switch the order of the variables. Finally, we added a label to mark that the attribute is binary. We can observe the edited data in the Data Table widget.



2.1.21 Impute

Replaces unknown values in the data.

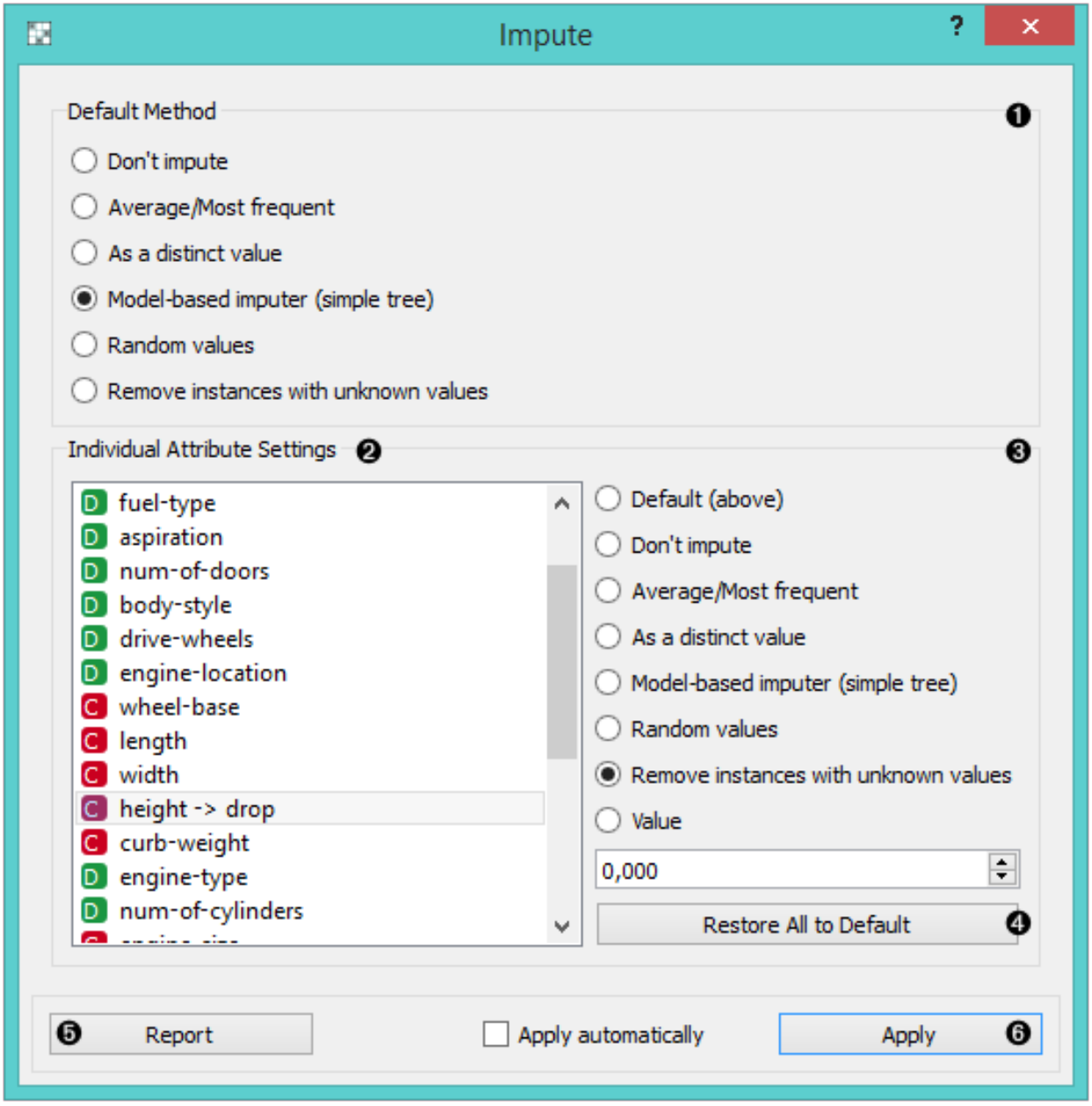
Inputs

- Data: input dataset
- Learner: learning algorithm for imputation

Outputs

- Data: dataset with imputed values

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. The default imputation is (1-NN).



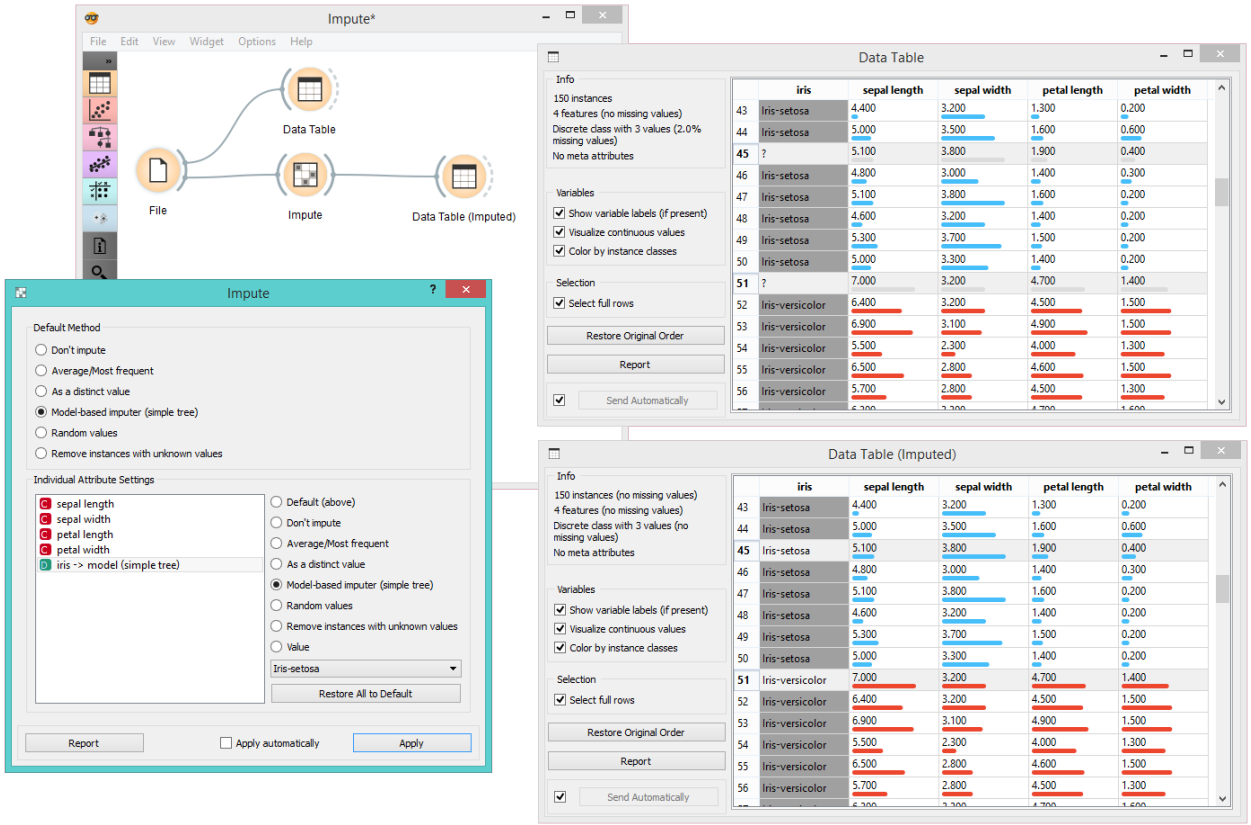
1. In the top-most box, *Default method*, the user can specify a general imputation technique for all attributes.
   - **Don't Impute** does nothing with the missing values.
   - **Average/Most-frequent** uses the average value (for continuous attributes) or the most common value (for discrete attributes).
   - **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* dataset 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").



2.1.22 Merge Data

Merges two datasets, based on values of selected attributes.

Inputs

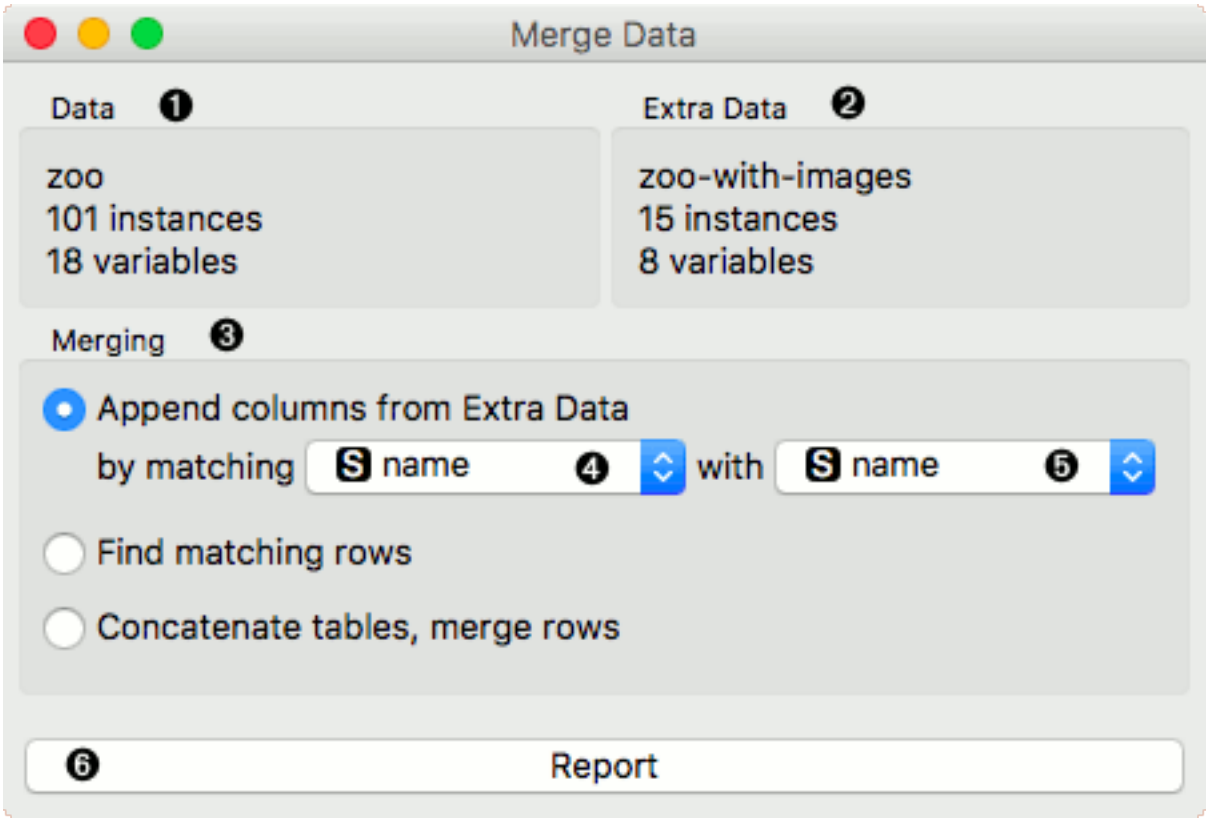
- Data: input dataset
- Extra Data: additional dataset

Outputs

- Data: dataset with features added from extra data

The **Merge Data** widget is used to horizontally merge two datasets, based on values of selected attributes. In the input, two datasets are required, data and extra data. 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 to which attributes from input extra data are appended.

Merging is done by values of selected (merging) attributes. First, the value of the merging attribute from Data is taken and instances from Extra Data are searched for matching values. If more than a single instance from Extra Data was to be found, the attribute is removed from available merging attributes.

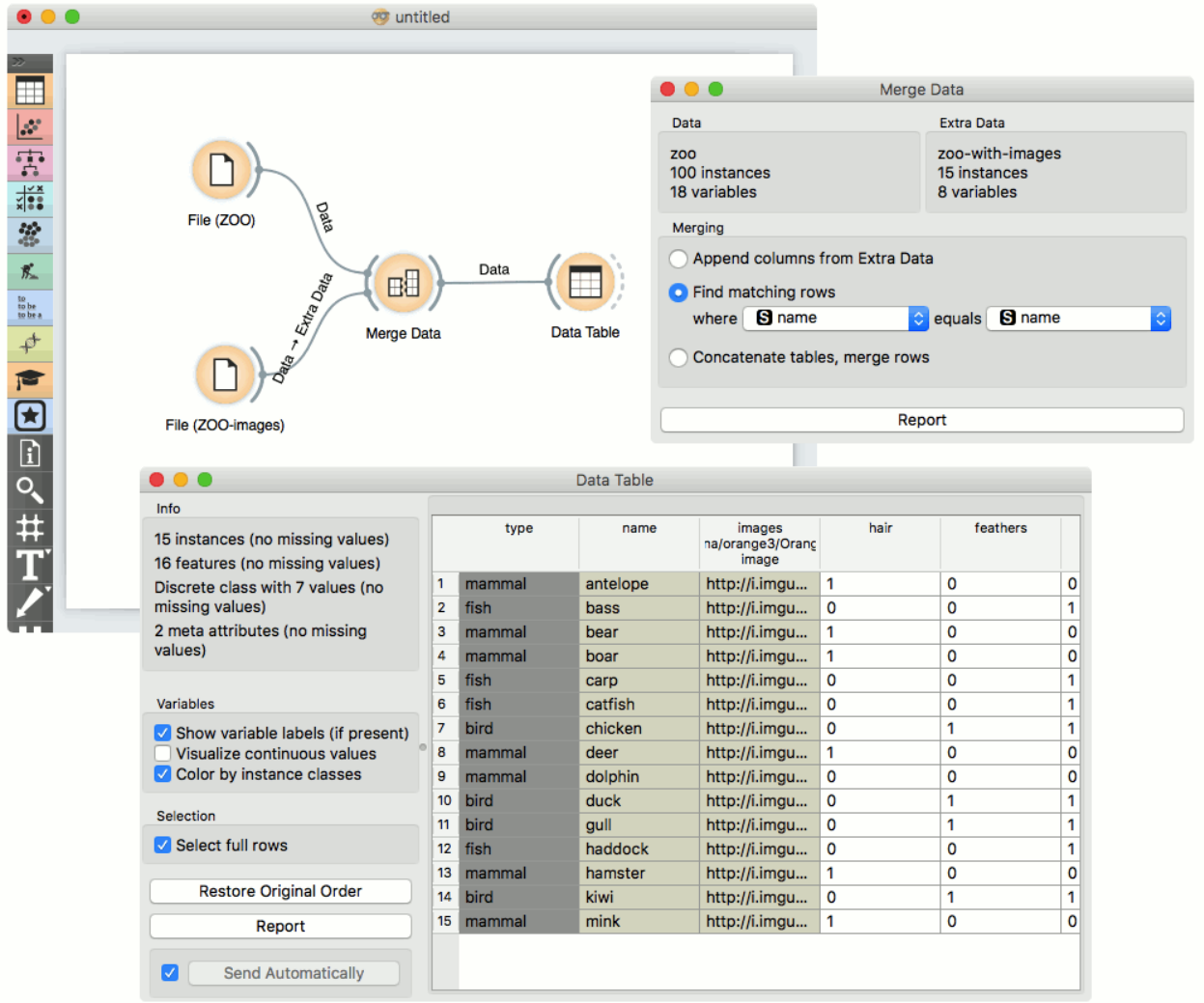


1. Information on Data
2. Information on Extra Data
3. Merging type. **Append columns from Extra Data** outputs all instances from Data appended by matching instances from Extra Data. When no match is found,unknown values are appended. **Find matching rows** outputs only matching instances. **Concatenate tables, merge rows** outputs all instances from both inputs, even though the match may not be found. In that case unknown values are assigned.
4. List of comparable attributes from Data

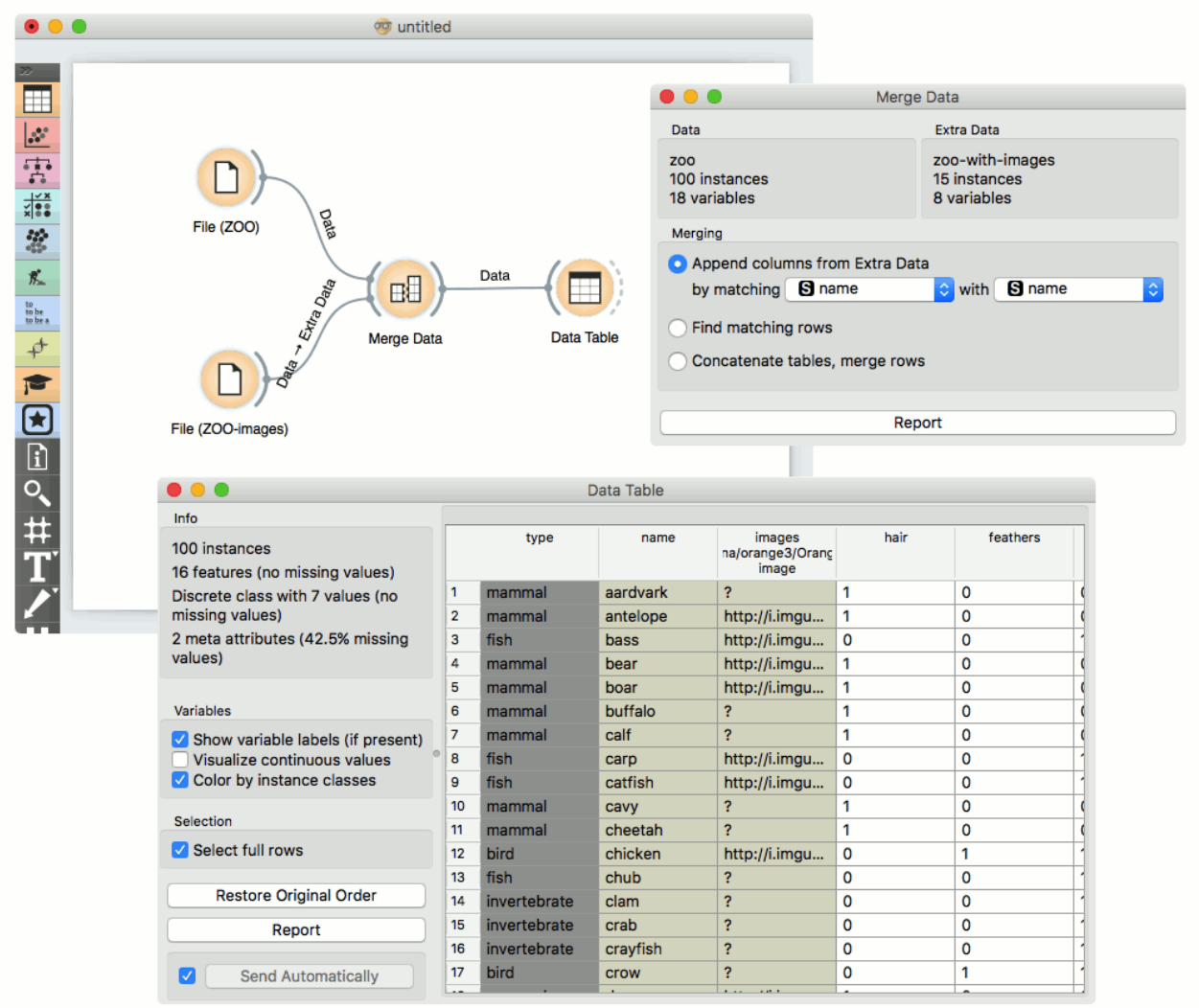
5. List of comparable attributes from Extra Data
6. Produce a report.

Example

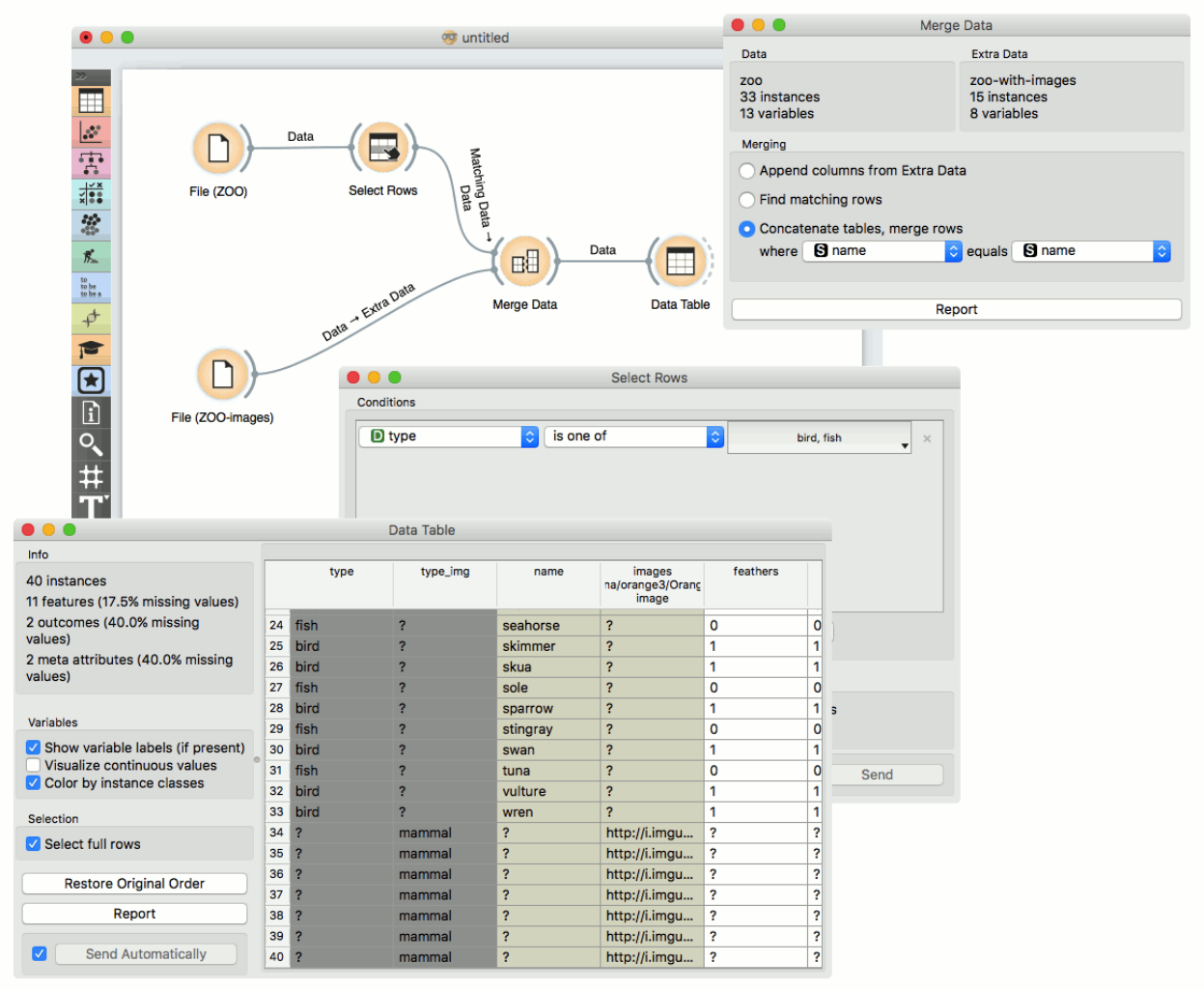
Merging two datasets results in appending new attributes to the original file, based on a selected common attribute. In the example below, we wanted to merge the **zoo.tab** file containing only factual data with zoo-with-images.tab containing images. Both files share a common string attribute *names*. Now, we create a workflow connecting the two files. The *zoo.tab* data is connected to **Data** input of the **Merge Data** widget, and the *zoo-with-images.tab* data to the **Extra Data** 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 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.



The third type of merging is shown in the next workflow. The output consist of both inputs, with unknown values assigned where no match was found.



2.1.23 Outliers

Simple outlier detection by comparing distances between instances.

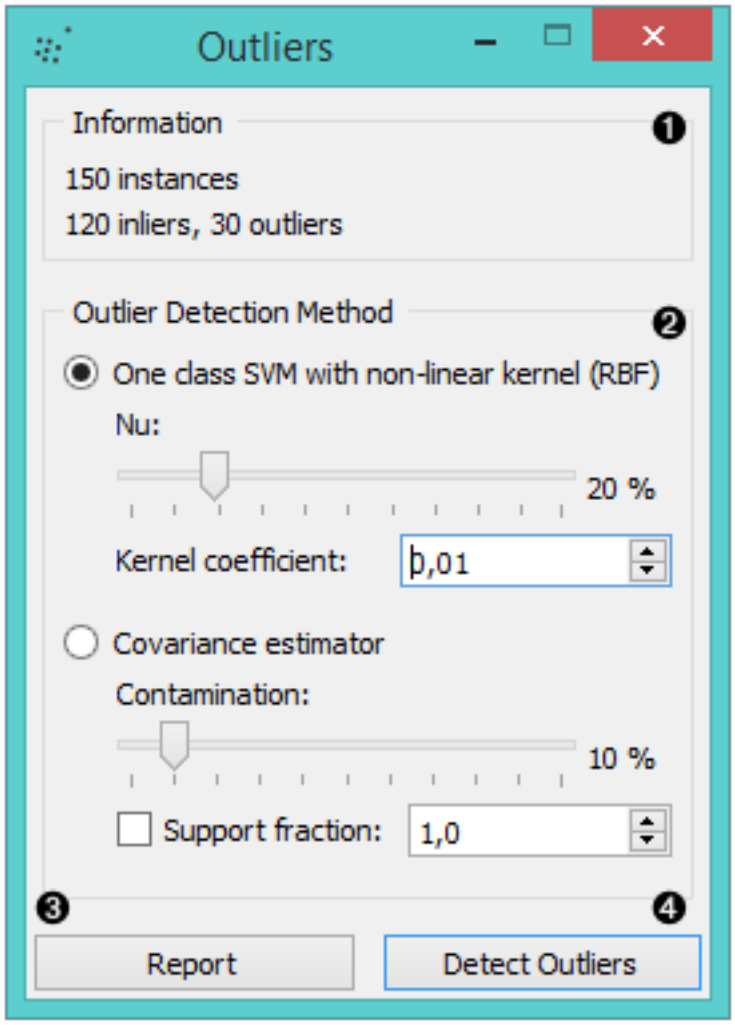
Inputs

- Data: input dataset
- Distances: distance matrix

Outputs

- Outliers: instances scored as outliers
- Inliers: instances not scored as outliers

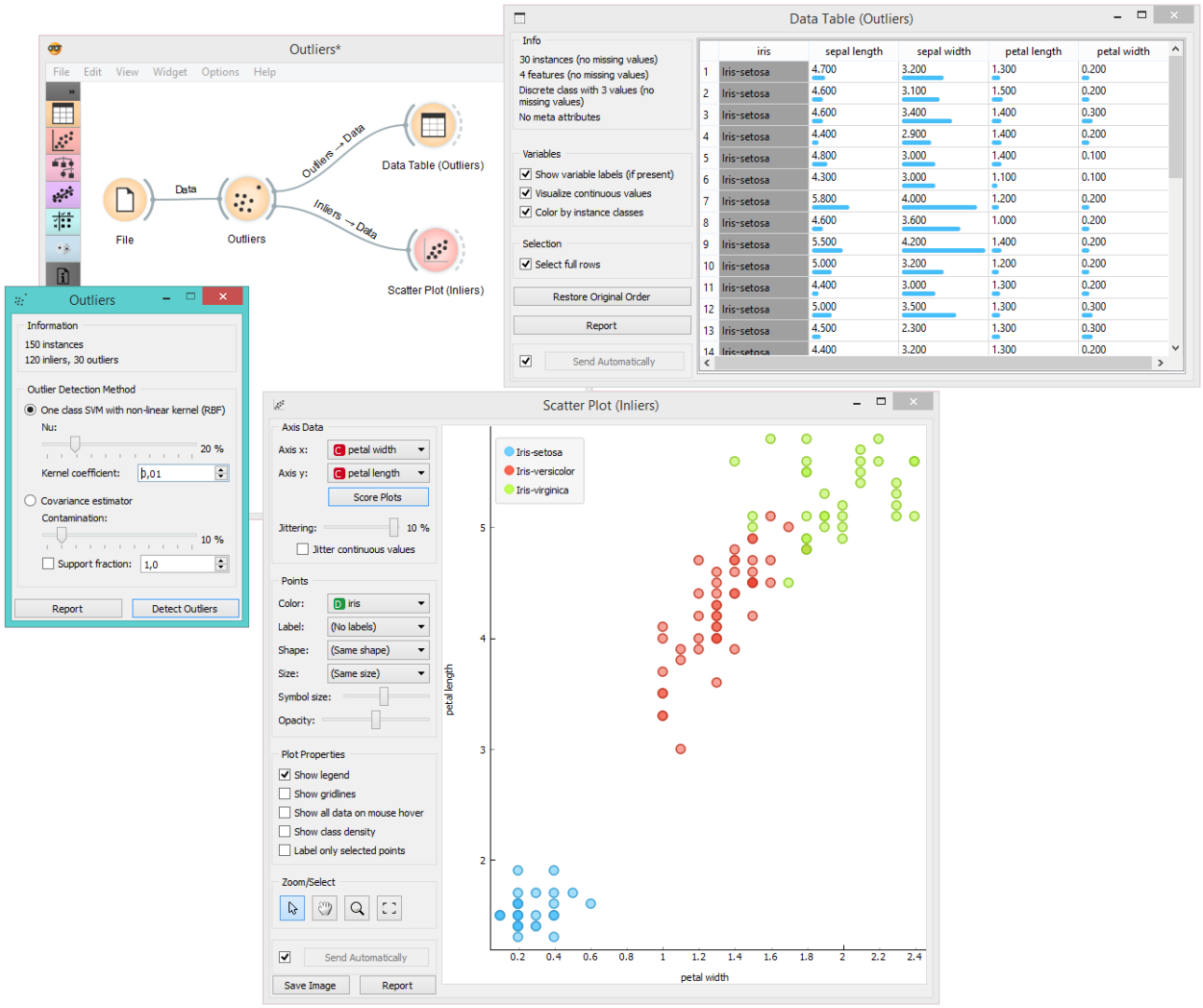
The **Outliers** widget applies one of the two methods for outlier detection. Both methods apply classification to the dataset, one with SVM (multiple kernels) and the other with elliptical envelope. *One-class SVM with non-linear kernels (RBF)* performs well with non-Gaussian distributions, while *Covariance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



2.1.24 Preprocess

Preprocesses data with selected methods.

Inputs

- Data: input dataset

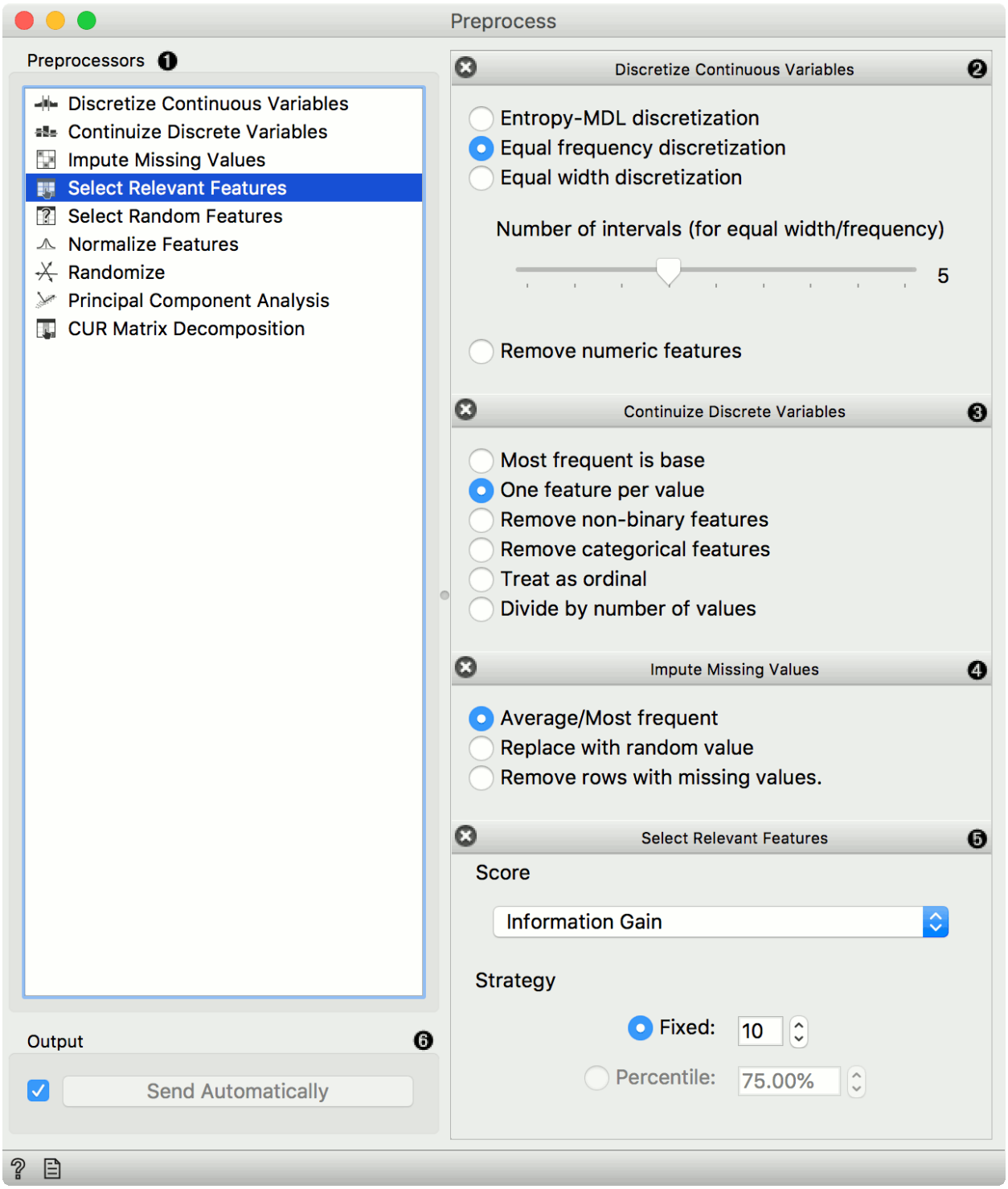
Outputs

- Preprocessor: preprocessing method
- Preprocessed Data: data preprocessed with selected methods

Preprocessing is crucial for achieving better-quality analysis results. The **Preprocess** widget offers several preprocessing methods that can be combined in a single preprocessing pipeline. Some methods are available as separate widgets, which offer advanced techniques and greater parameter tuning.

1. List of preprocessors. Double click the preprocessors you wish to use and shuffle their order by dragging them up or down. You can also add preprocessors by dragging them from the left menu to the right.
2. Preprocessing pipeline.
3. When the box is ticked (*Send Automatically*), the widget will communicate changes automatically. Alternatively, click *Send*.

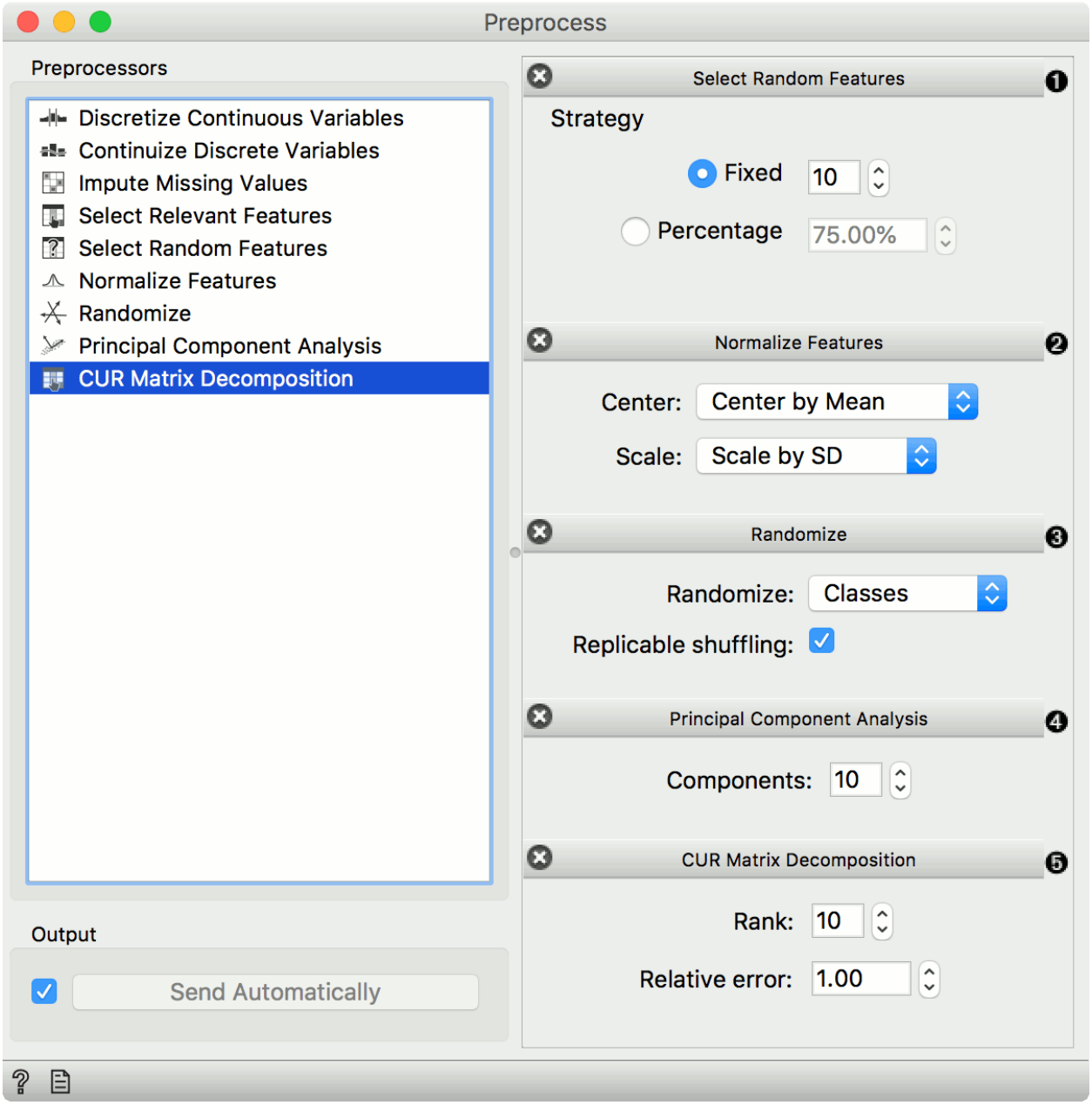
Preprocessors



1. Discretization of continuous values:
   - Entropy-MDL discretization by Fayyad and Irani that uses expected information to determine bins.
   - *Equal frequency discretization* splits by frequency (same number of instances in each bin.

- *Equal width discretization* creates bins of equal width (span of each bin is the same).
- *Remove numeric features* altogether.

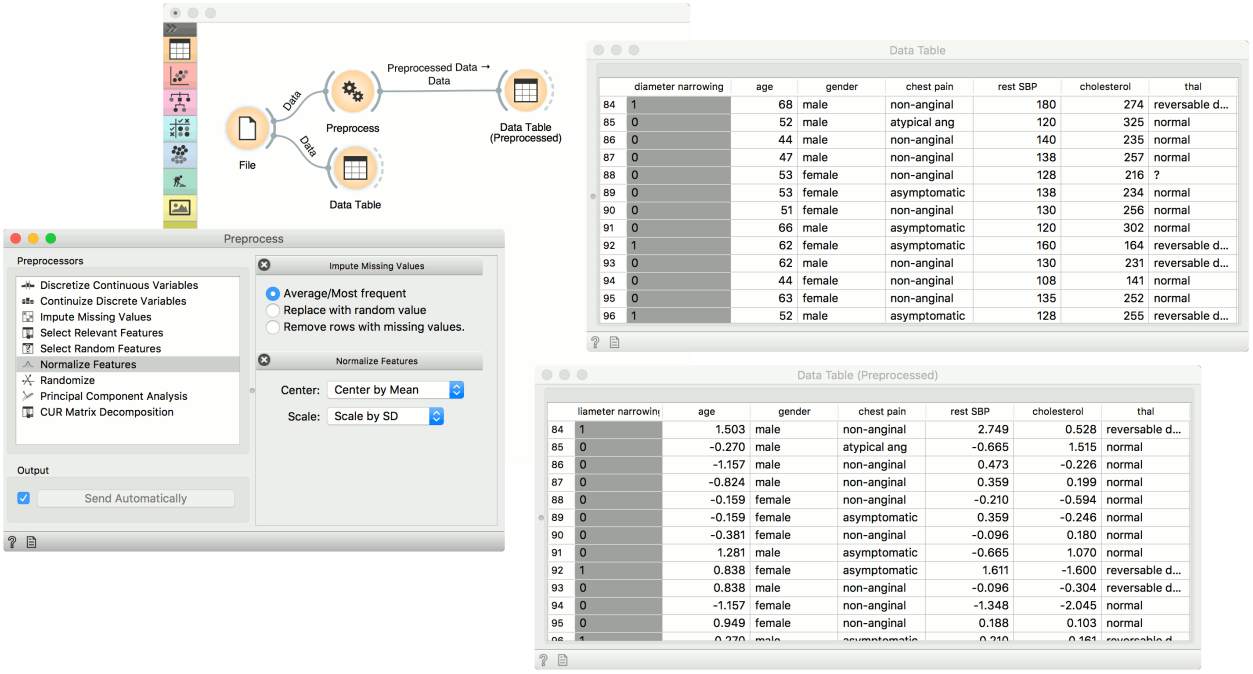
2. Continuization of discrete values:
   - *Most frequent as base* treats the most frequent discrete value as 0 and others as 1. The discrete attributes with more than 2 values, the most frequent will be considered as a base and contrasted with remaining values in corresponding columns.
   - *One feature per value* creates columns for each value, place 1 where an instance has that value and 0 where it doesn't. Essentially One Hot Encoding.
   - *Remove non-binary features* retains only categorical features that have values of either 0 or 1 and transforms them into continuous.
   - *Remove categorical features* removes categorical features altogether.
   - *Treat as ordinal* takes discrete values and treats them as numbers. If discrete values are categories, each category will be assigned a number as they appear in the data.
   - *Divide by number of values* is similar to treat as ordinal, but the final values will be divided by the total number of values and hence the range of the new continuous variable will be [0, 1].
3. Impute missing values:
   - *Average/Most frequent* replaces missing values (NaN) with the average (for continuous) or most frequent (for discrete) value.
   - *Replace with random value* replaces missing values with random ones within the range of each variable.
   - *Remove rows with missing values*.
4. Select relevant features:
   - Similar to Rank, this preprocessor outputs only the most informative features. Score can be determined by information gain, gain ratio, gini index, ReliefF, fast correlation based filter, ANOVA, Chi2, RReliefF, and Univariate Linear Regression.
   - *Strategy* refers to how many variables should be on the output. *Fixed* returns a fixed number of top scored variables, while *Percentile* return the selected top percent of the features.



1. *Select random features* outputs either a fixed number of features from the original data or a percentage. This is mainly used for advanced testing and educational purposes.
2. Normalize adjusts values to a common scale. Center values by mean or median or omit centering altogether. Similar for scaling, one can scale by SD (standard deviation), by span or not at all.
3. Randomize instances. Randomize classes shuffles class values and destroys connection between instances and class. Similarly, one can randomize features or meta data. If replicable shuffling is on, randomization results can be shared and repeated with a saved workflow. This is mainly used for advanced testing and educational purposes.
4. Principal component analysis outputs results of a PCA transformation. Similar to the PCA widget.
5. CUR matrix decomposition is a dimensionality reduction method, similar to SVD.

Examples

In the first example, we have used the *heart_disease.tab* dataset available in the dropdown menu of the File widget. then we used **Preprocess** to impute missing values and normalize features. We can observe the changes in the Data Table and compare it to the non-processed data.



In the second example, we show how to use **Preprocess** for predictive modeling.

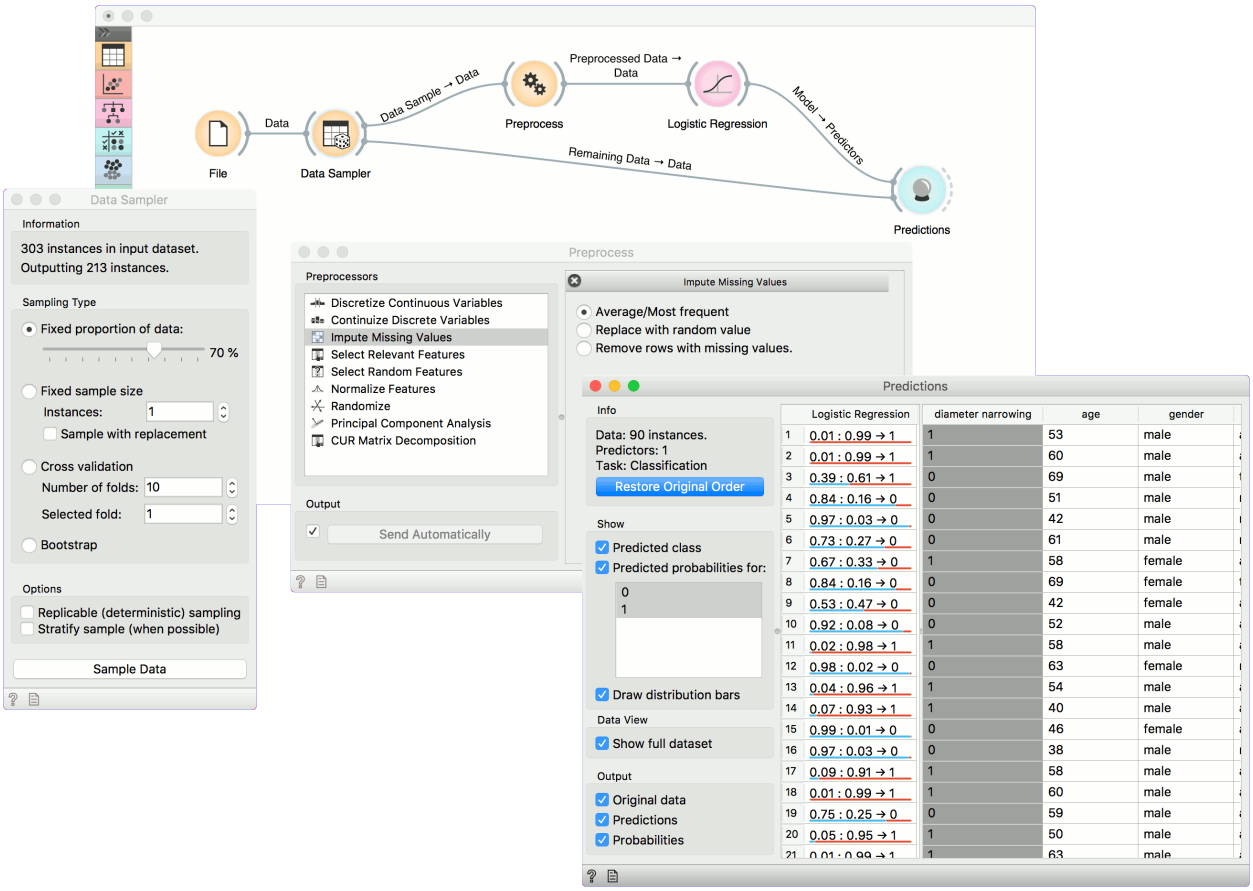
This time we are using the *heart_disease.tab* data from the File widget. You can access the data through the dropdown menu. This is a dataset with 303 patients that came to the doctor suffering from a chest pain. After the tests were done, some patients were found to have diameter narrowing and others did not (this is our class variable).

The heart disease data have some missing values and we wish to account for that. First, we will split the data set into train and test data with Data Sampler.

Then we will send the *Data Sample* into Preprocess. We will use *Impute Missing Values*, but you can try any combination of preprocessors on your data. We will send preprocessed data to Logistic Regression and the constructed model to **Predictions**.

Finally, **Predictions** also needs the data to predict on. We will use the output of Data Sampler for prediction, but this time not the *Data Sample*, but the *Remaining Data*, this is the data that wasn't used for training the model.

Notice how we send the remaining data directly to **Predictions** without applying any preprocessing. This is because Orange handles preprocessing on new data internally to prevent any errors in the model construction. The exact same preprocessor that was used on the training data will be used for predictions. The same process applies to Test & Score.



2.1.25 Transform

Given dataset and preprocessor transforms the dataset.

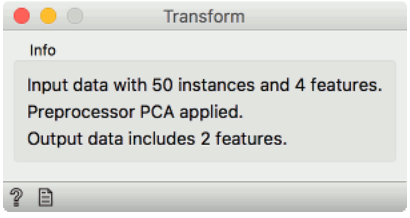
Inputs

- Data: input dataset
- Preprocessor: preprocessor for transformation

Outputs

- Transformed Data: transformed dataset

Transform maps new data into a transformed space. For example, if we transform some data with PCA and wish to observe new data in the same space, we can use transform to map the new data into the PCA space created from the original data.



Widget accepts new data on the input and a preprocessor that was used to transform the old data.

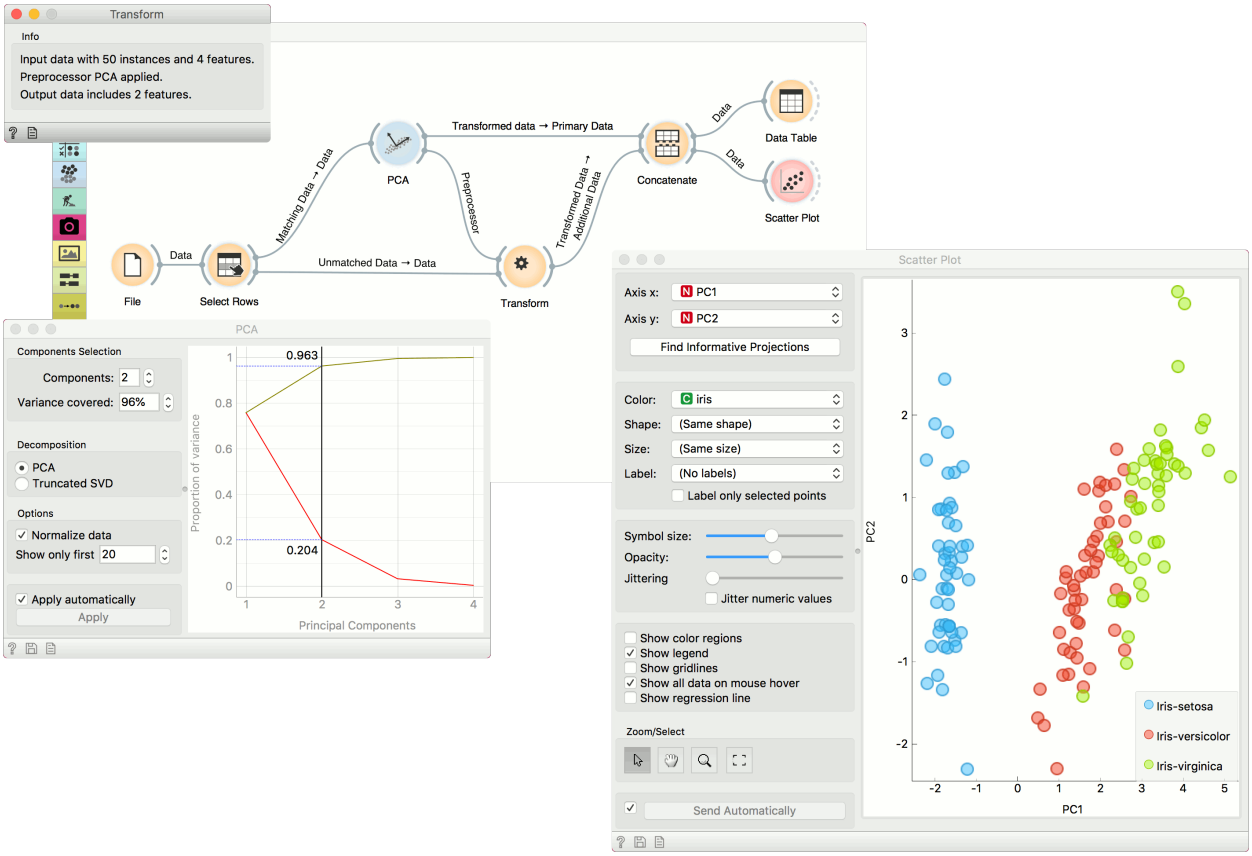
Example

We will use iris data from the File widget for this example. To create two separate data sets, we will use Select Rows and set the condition to *iris is one of iris-setosa, iris-versicolor*. This will output a data set with a 100 rows, half of them belonging to iris-setosa class and the other half to iris-versicolor.

We will transform the data with PCA and select the first two components, which explain 96% of variance. Now, we would like to apply the same preprocessing on the 'new' data, that is the remaining 50 iris virginicas. Send the unused data from **Select Rows** to **Transform**. Make sure to use the *Unmatched Data* output from **Select Rows** widget. Then add the *Preprocessor* output from **PCA**.

Transform will apply the preprocessor to the new data and output it. To add the new data to the old data, use Concatenate. Use *Transformed Data* output from **PCA** as *Primary Data* and *Transformed Data* from **Transform** as *Additional Data*.

Observe the results in a Data Table or in a Scatter Plot to see the new data in relation to the old one.



2.1.26 Purge Domain

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

Inputs

- Data: input dataset

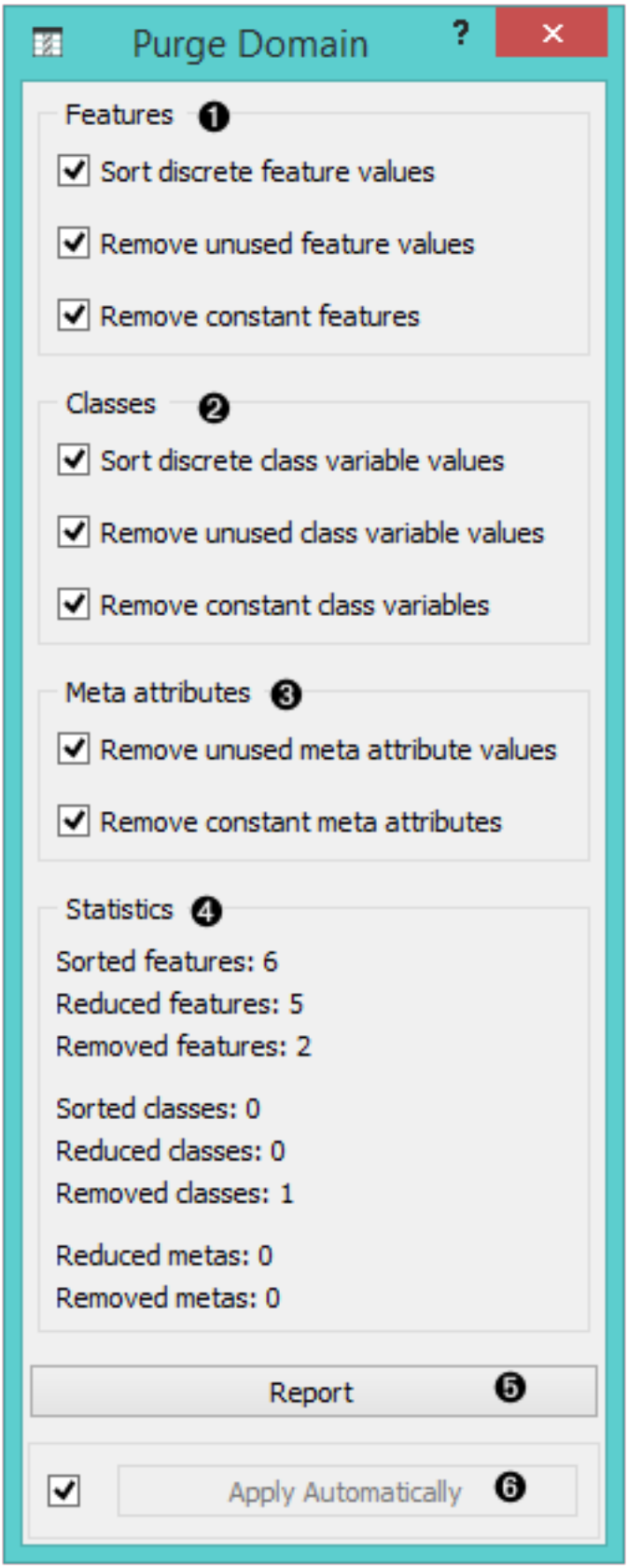
Outputs

- Data: filtered dataset

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.


Purge Domain
Features
1
Sort discrete feature values
Remove unused feature values
Remove constant features
Classes
2
Sort discrete class variable values
Remove unused class variable values
Remove constant class variables
Meta attributes
3
Remove unused meta attribute values
Remove constant meta attributes
Statistics
4
Sorted features: 6
Reduced features: 5
Removed features: 2
Sorted classes: 0
Reduced classes: 0
Removed classes: 1
Reduced metas: 0
Removed metas: 0
Report
5
Apply Automatically
6

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.

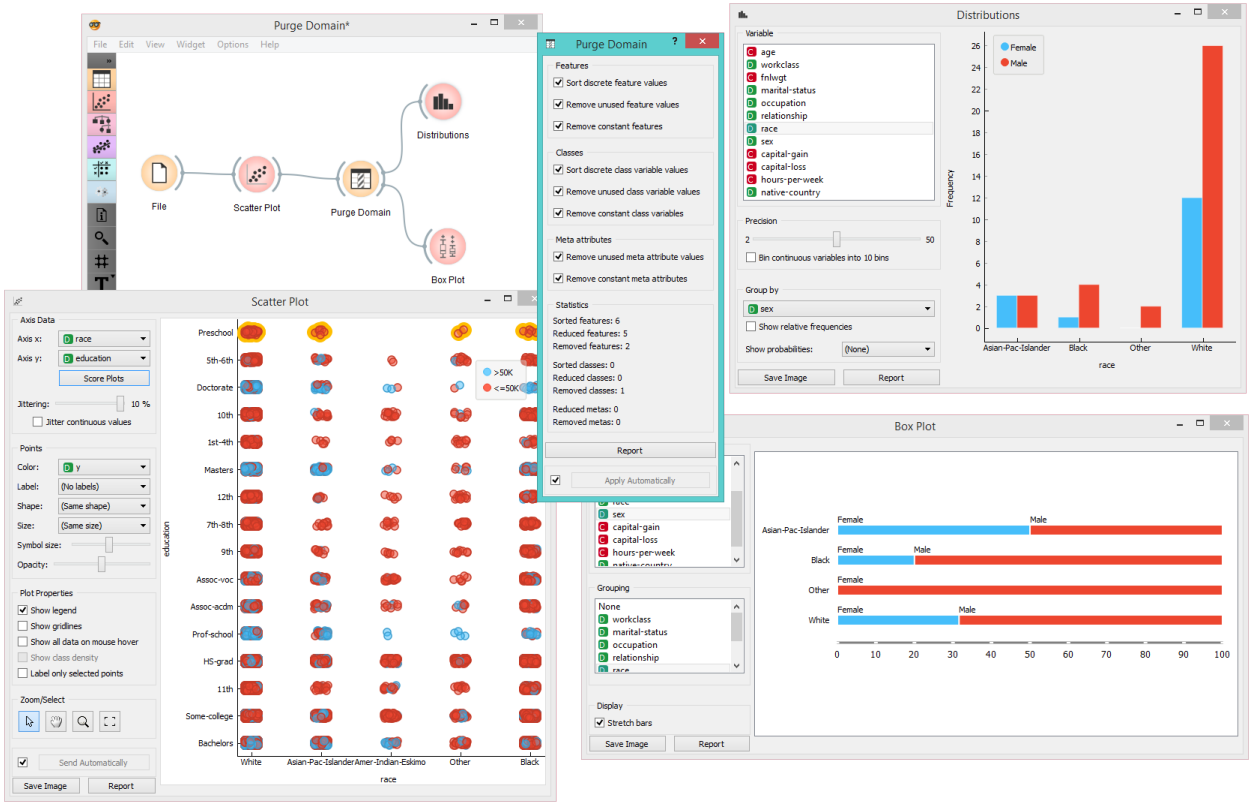
Such purification is done by the widget **Purge Domain**. Ordinary attributes and class attributes are treated separately. For each, we can decide if we want the values sorted or not. Next, we may allow the widget to remove attributes with less than two values or remove the class attribute if there are 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* dataset: 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.



2.1.27 Rank

Ranking of attributes in classification or regression datasets.

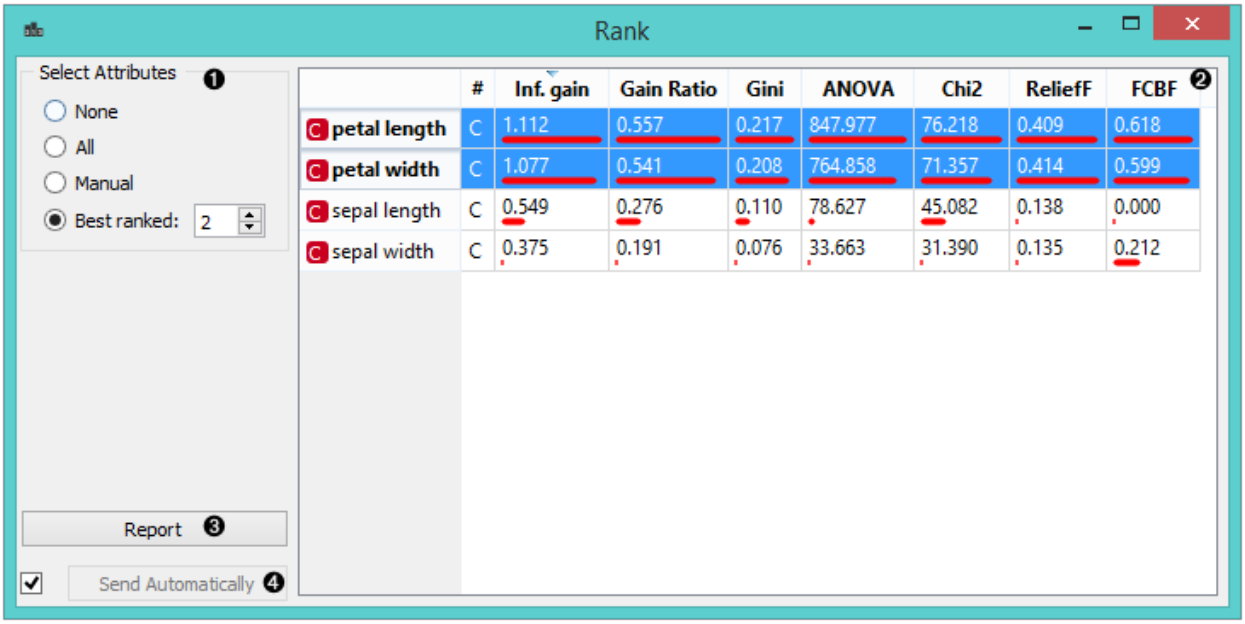
Inputs

- Data: input dataset
- Scorer: models for feature scoring

Outputs

- Reduced Data: dataset with selected attributes

The **Rank** widget considers class-labeled datasets (classification or regression) and scores the attributes according to their correlation with the class. Rank accepts also models for scoring, such as linear regression, logistic regression, random forest, SGD, etc.



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.

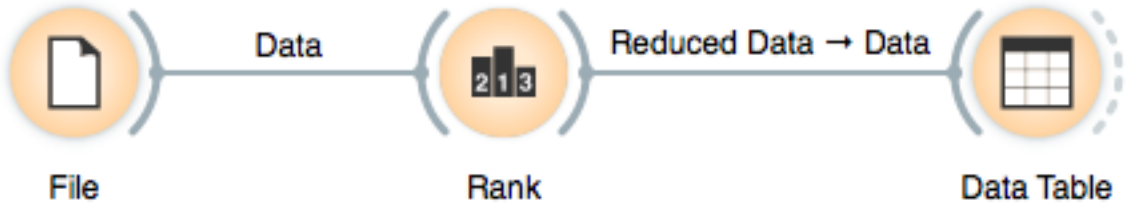
Scoring methods

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 vaules 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

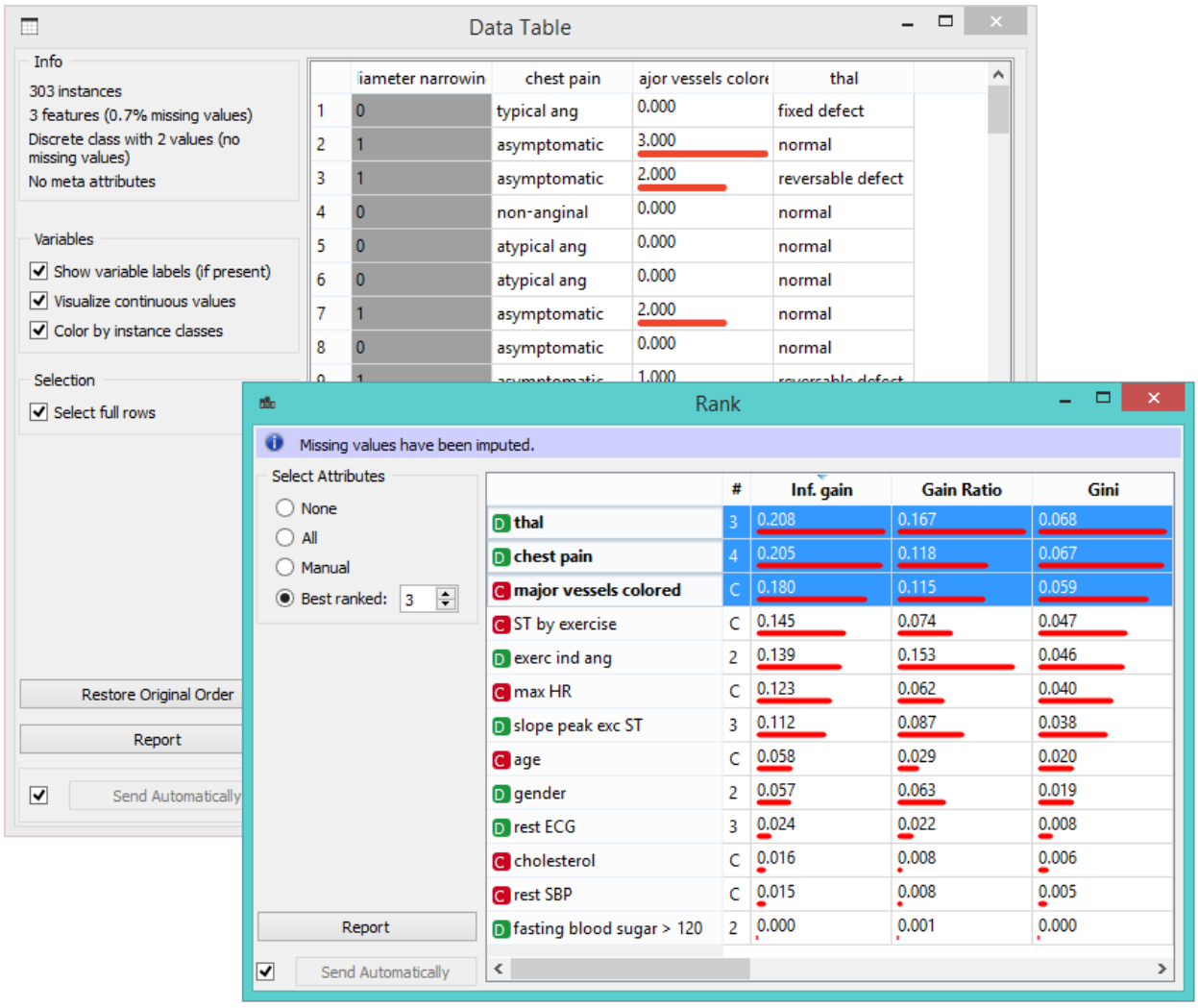
Additionally, you can connect certain learners that enable scoring the features according to how important they are in models that the learners build (e.g. Linear Regression / Logistic Regression, Random Forest, SGD).

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:

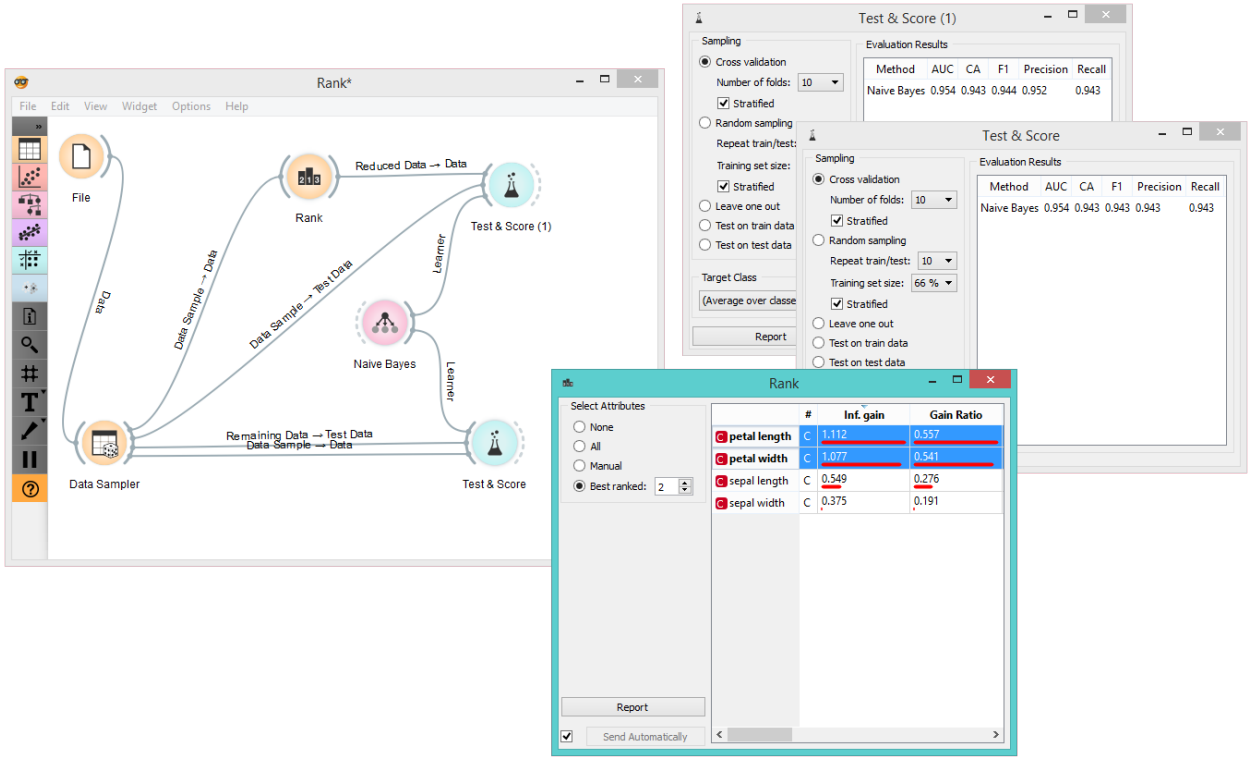


Notice how the widget outputs a dataset that includes only the best-scored attributes:



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 datasets are passed to their own Test & Score widgets, which develop a *Naive Bayes* classifier and score it on a test set.



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

2.1.28 Correlations

Compute all pairwise attribute correlations.

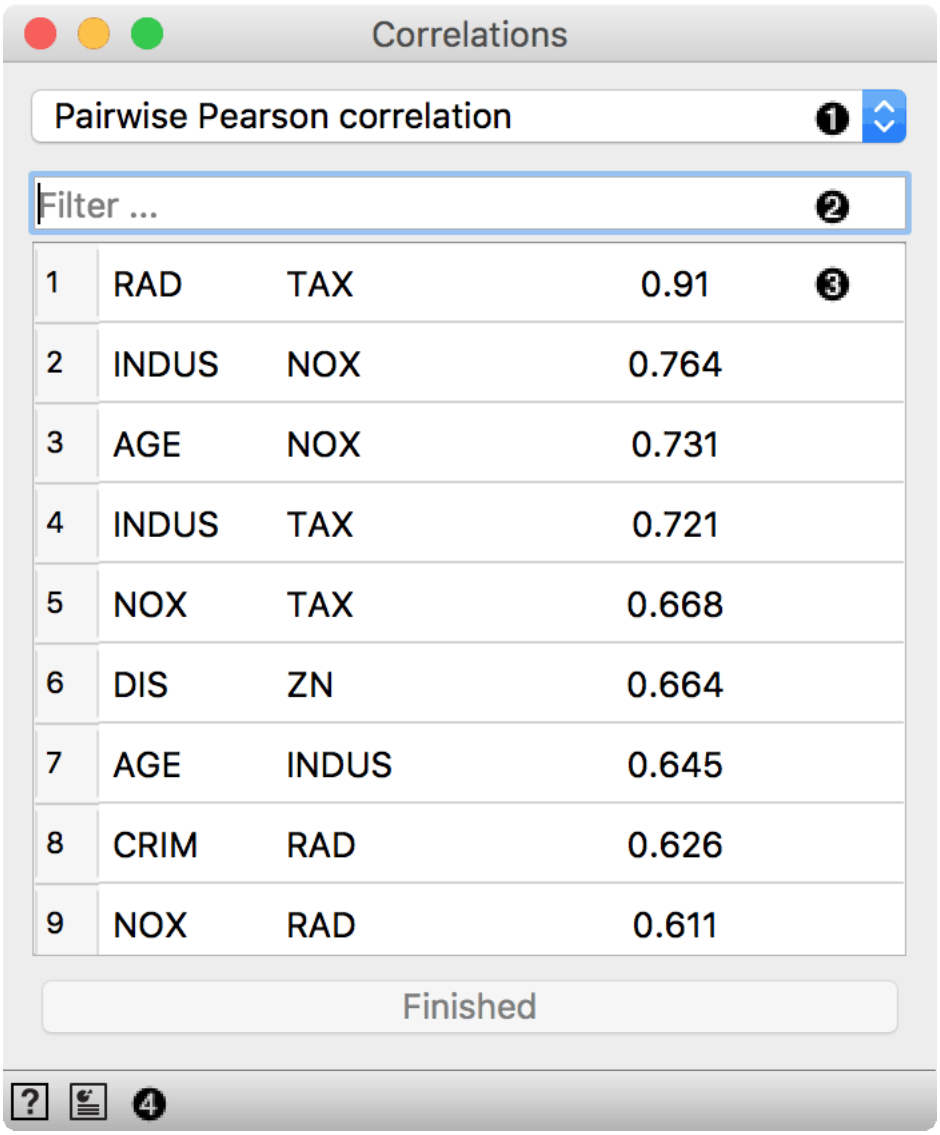
Inputs

- Data: input dataset

Outputs

- Data: input dataset
- Features: selected pair of features
- Correlations: data table with correlation scores

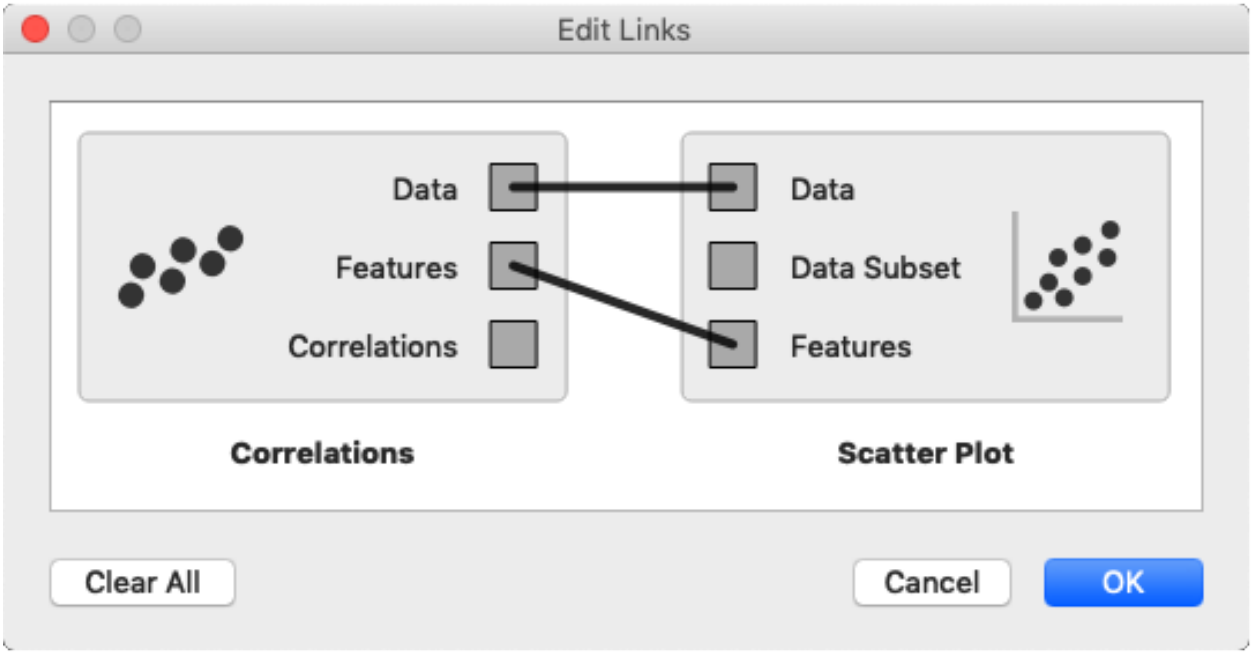
Correlations computes Pearson or Spearman correlation scores for all pairs of features in a dataset. These methods can only detect monotonic relationship.



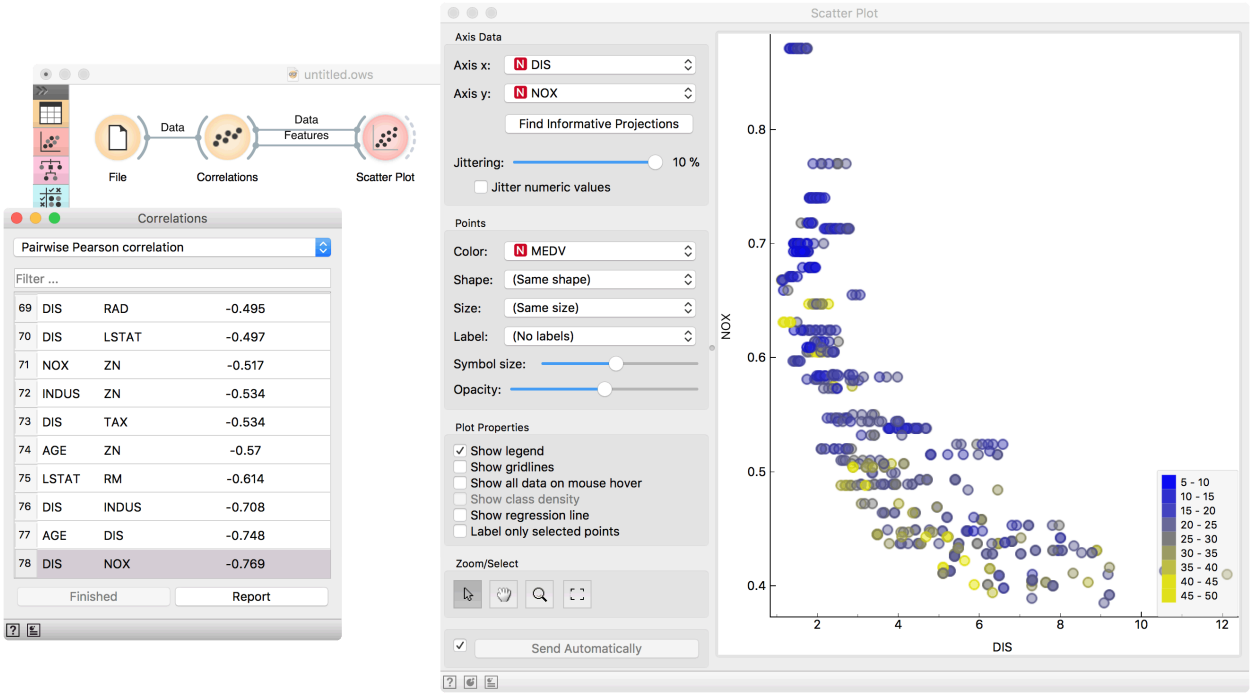
1. Correlation measure:
   - Pairwise Pearson correlation.
   - Pairwise Spearman correlation.
2. Filter for finding attribute pairs.
3. A list of attribute pairs with correlation coefficient. Press *Finished* to stop computation for large datasets.
4. Access widget help and produce report.

Example

Correlations can be computed only for numeric (continuous) features, so we will use *housing* as an example data set. Load it in the File widget and connect it to **Correlations**. Positively correlated feature pairs will be at the top of the list and negatively correlated will be at the bottom.



Go to the most negatively correlated pair, DIS-NOX. Now connect Scatter Plot to **Correlations** and set two outputs, Data to Data and Features to Features. Observe how the feature pair is immediately set in the scatter plot. Looks like the two features are indeed negatively correlated.



2.1.29 Color

Set color legend for variables.

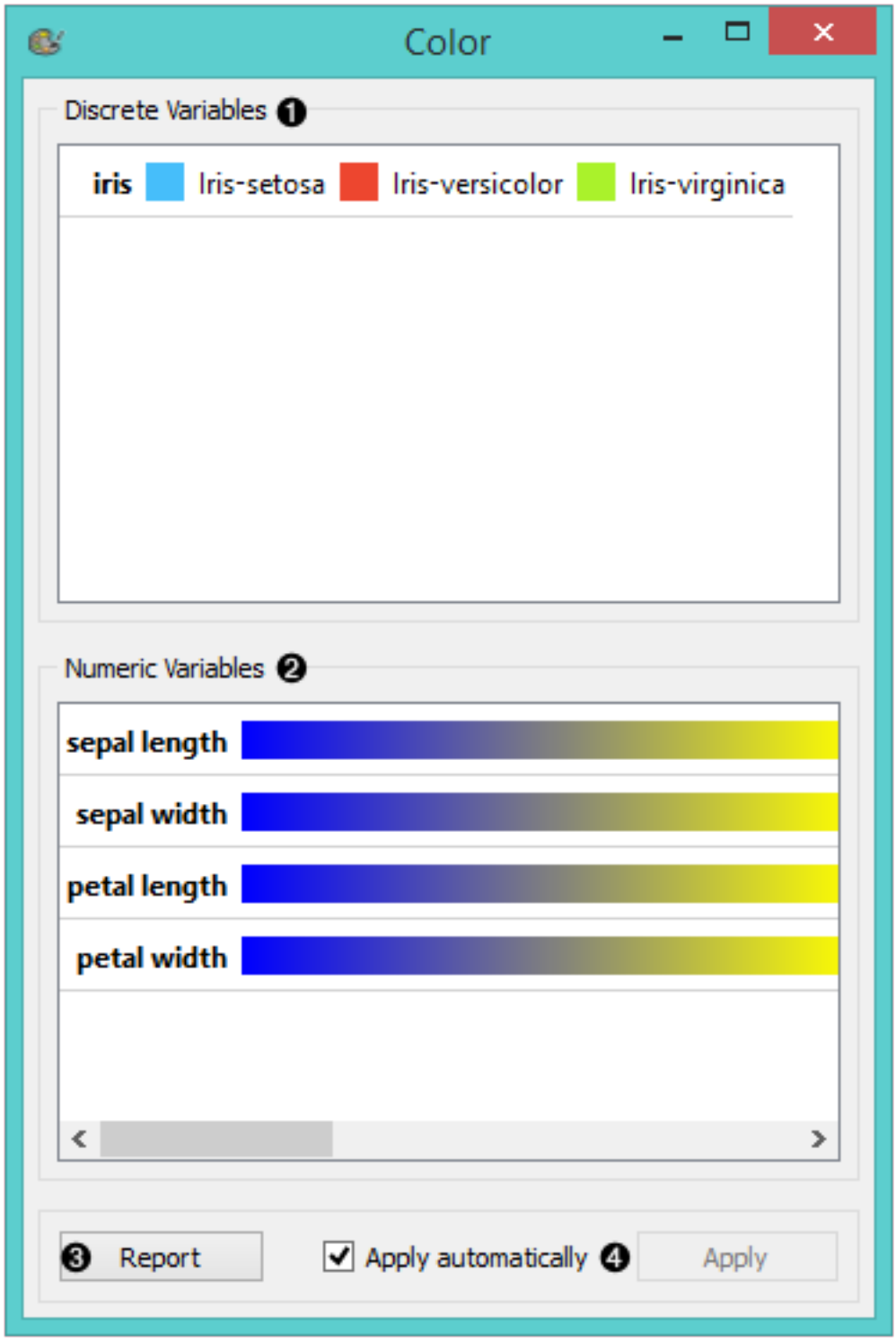
Inputs

- Data: input data set

Outputs

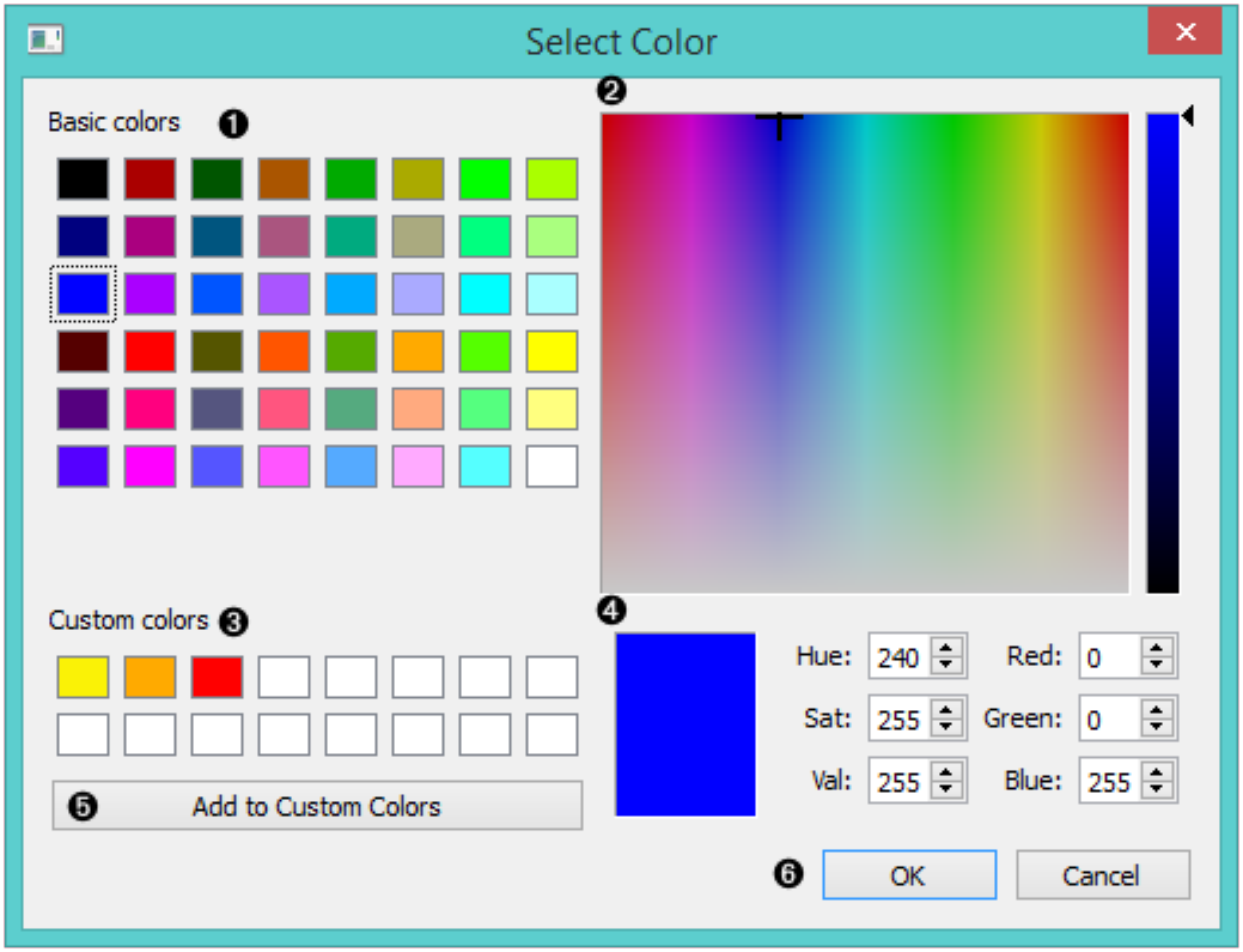
- Data: data set with a new color legend

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 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*.

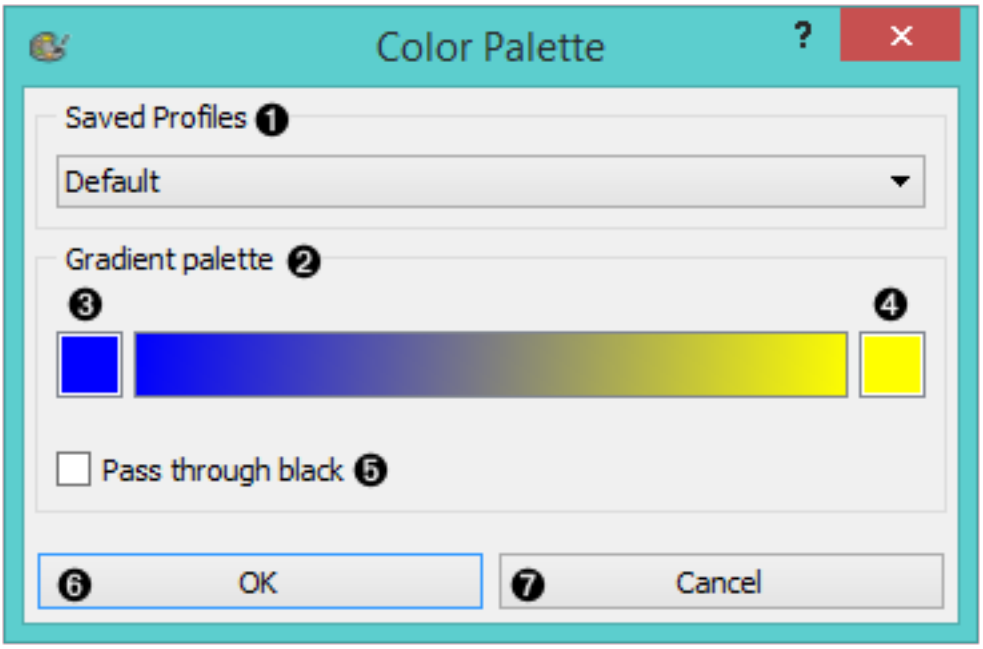
Discrete variables



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.

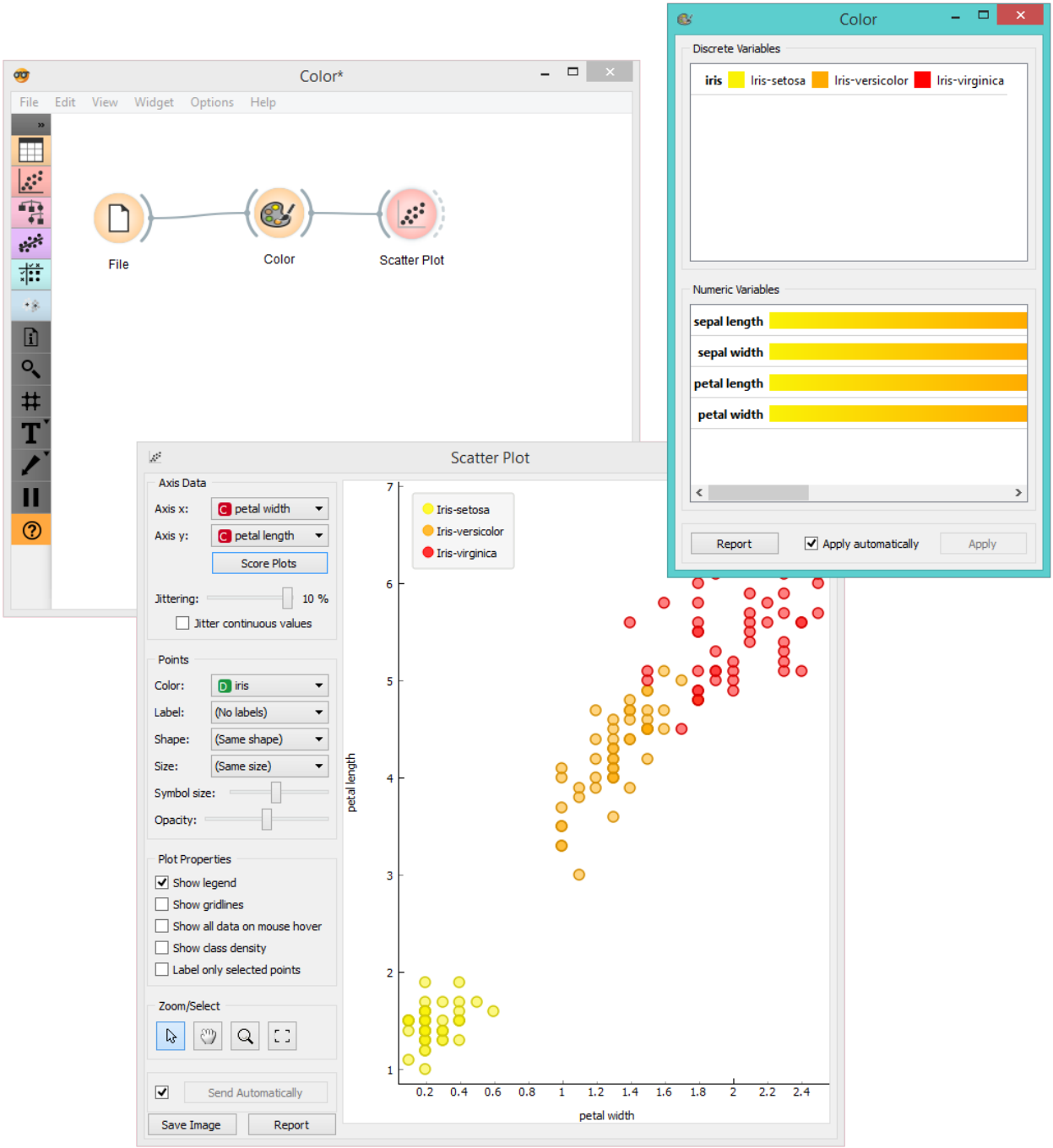
Numeric variables



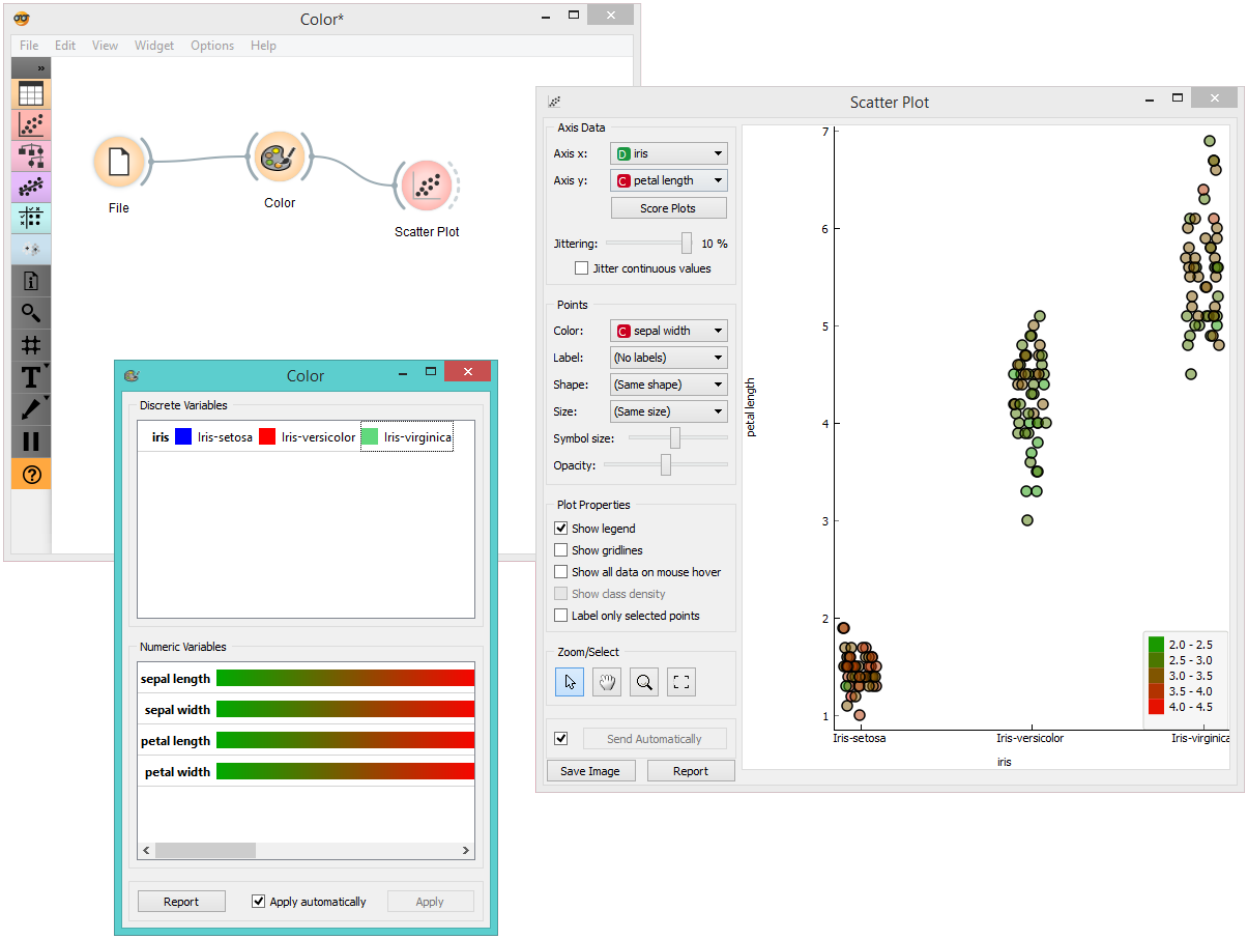
1. Choose a gradient from your saved profiles. The default profile is already set.
2. The gradient palette
3. Select the left side of the gradient. Double clicking the color opens the *Select Color* window.
4. Select the right side of the gradient. Double clicking the color opens the *Select Color* window.
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.



2.1.30 Feature Statistics

Show basic statistics for data features.

Inputs

- Data: input data

Outputs

- Reduced data: table containing only selected features
- Statistics: table containing statistics of the selected features

The **Feature Statistics** widget provides a quick way to inspect and find interesting features in a given data set.

The Feature Statistics widget on the *heart-disease* data set. The feature *exerc ind ang* was manually changed to a meta variable for illustration purposes.

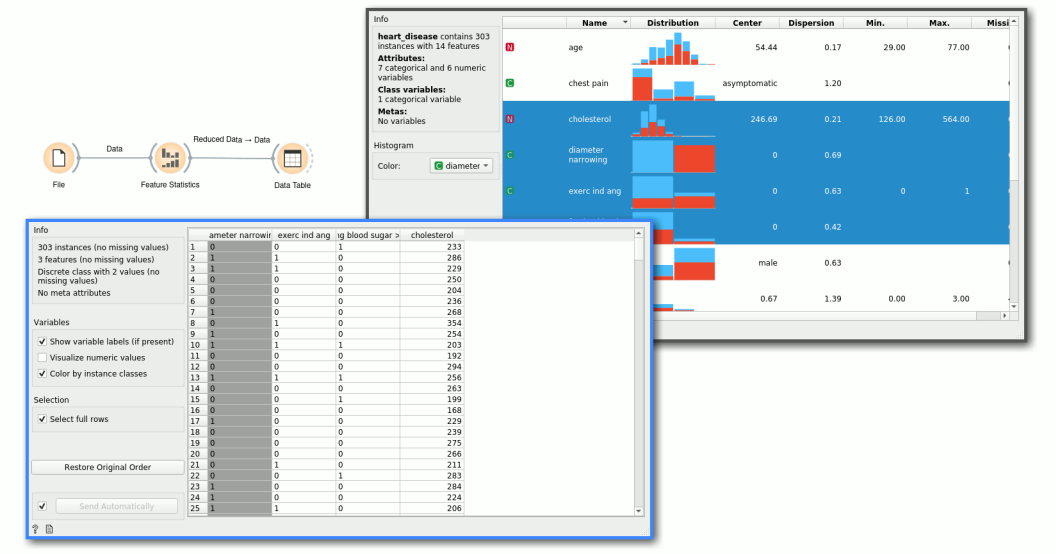
1. Info on the current data set size and number and types of features
2. The histograms on the right can be colored by any feature. If the selected feature is categorical, a discrete color palette is used (as shown in the example). If the selected feature is numerical, a continuous color palette is used. The table on the right contains statistics about each feature in the data set. The features can be sorted by each statistic, which we now describe.
3. The feature type - can be one of categorical, numeric, time and string.
4. The name of the feature.
5. A histogram of feature values. If the feature is numeric, we appropriately discretize the values into bins. If the feature is categorical, each value is assigned its own bar in the histogram.
6. The central tendency of the feature values. For categorical features, this is the mode. For numeric features, this is mean value.
7. The dispersion of the feature values. For categorical features, this is the entropy of the value distribution. For numeric features, this is the index of dispersion.
8. The minimum value. This is computed for numerical and ordinal categorical features.
9. The maximum value. This is computed for numerical and ordinal categorical features.
10. The number of missing values in the data.

Notice also that some rows are colored differently. White rows indicate regular features, gray rows indicate class variables and the lighter gray indicates meta variables.

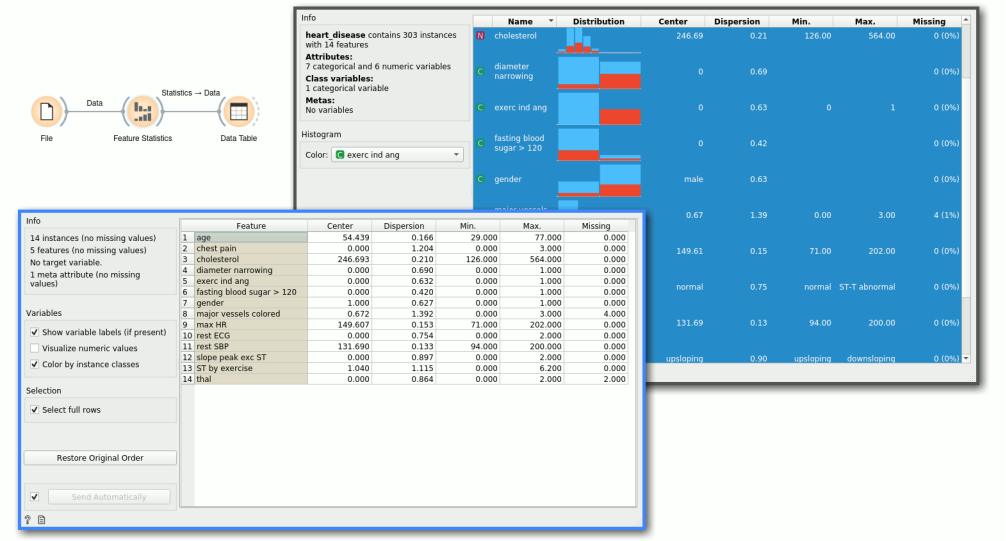
Example

The Feature Statistics widget is most often used after the File widget to inspect and find potentially interesting features in the given data set. In the following examples, we use the *heart-disease* data set.

Once we have found a subset of potentially interesting features, or we have found features that we would like to exclude, we can simply select the features we want to keep. The widget outputs a new data set with only these features.



Alternatively, if we want to store feature statistics, we can use the *Statistics* output and manipulate those values as needed. In this example, we simply select all the features and display the statistics in a table.



2.1.31 Neighbors

Compute nearest neighbors in data according to reference.

Inputs

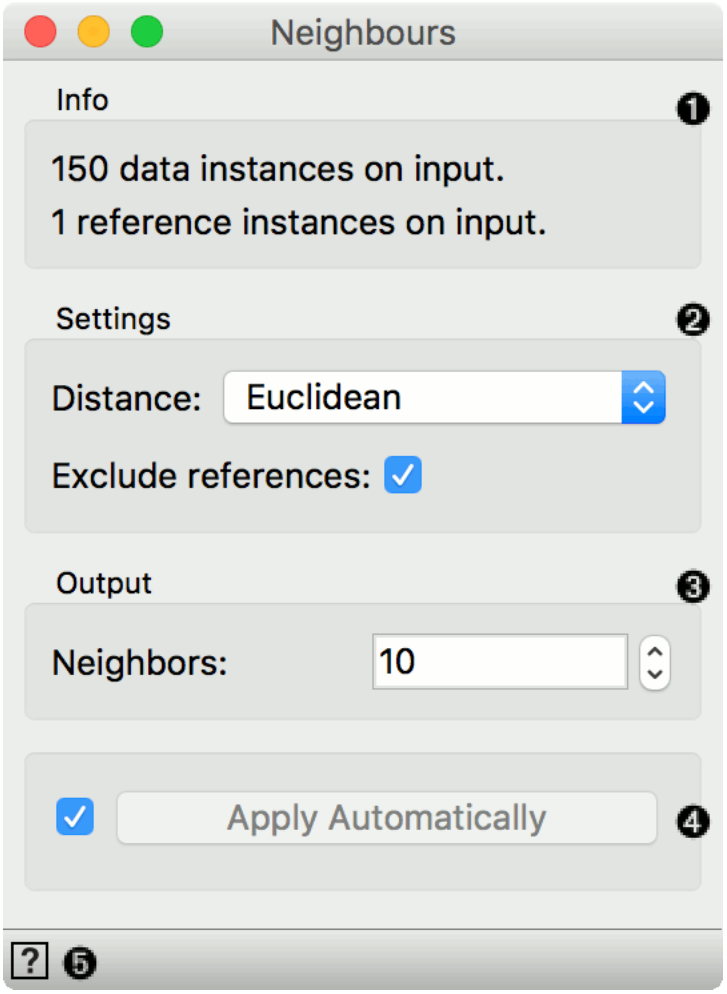
- Data: An input data set.

- Reference: A reference data instance for neighbor computation.

Outputs

- Neighbors: A data table of nearest neighbors according to reference.

The **Neighbors** widget computes nearest neighbors for a given reference and for a given distance measure.

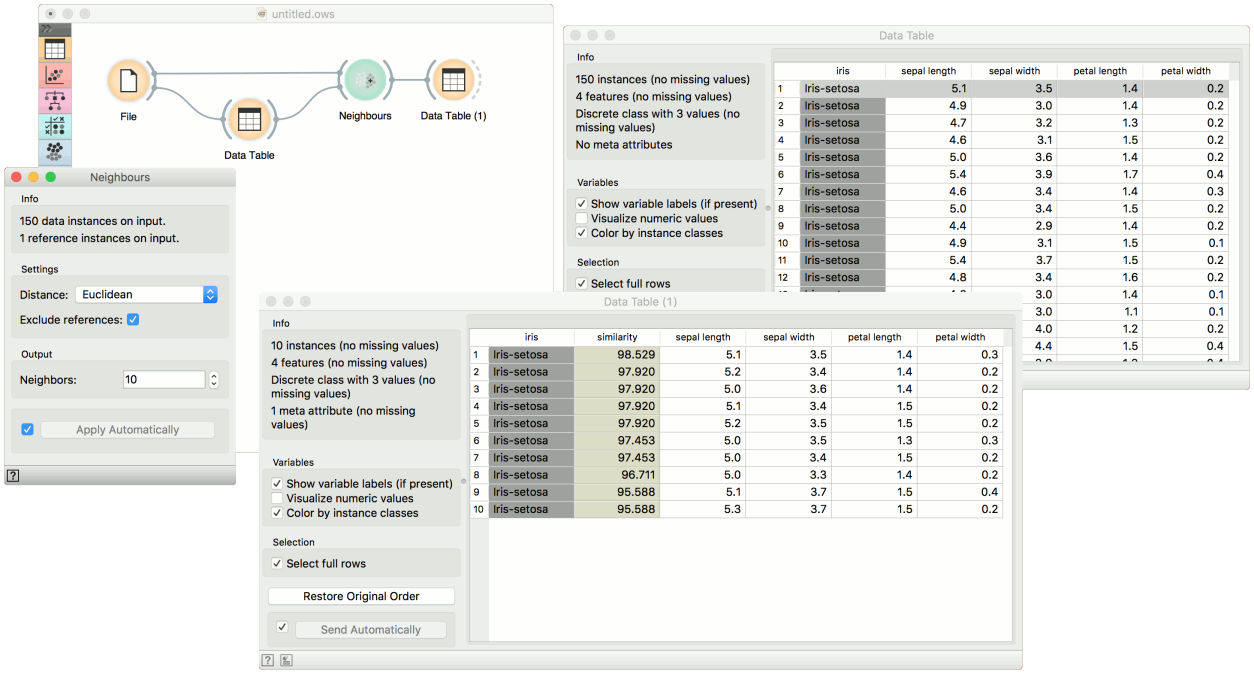


1. Information on the input data.
2. Distance measure for computing neighbors. Supported measures are: Euclidean, Manhattan, Mahalanobis, Cosine, Jaccard, Spearman, absolute Spearman, Pearson, absolute Pearson. If *Exclude references* is ticked, data instances that are highly similar to the reference (distance < 1e-5), will be excluded.
3. Number of neighbors on the output.
4. Click *Apply* to commit the changes. To communicate changes automatically tick *Apply Automatically*.
5. Access widget help.

Examples

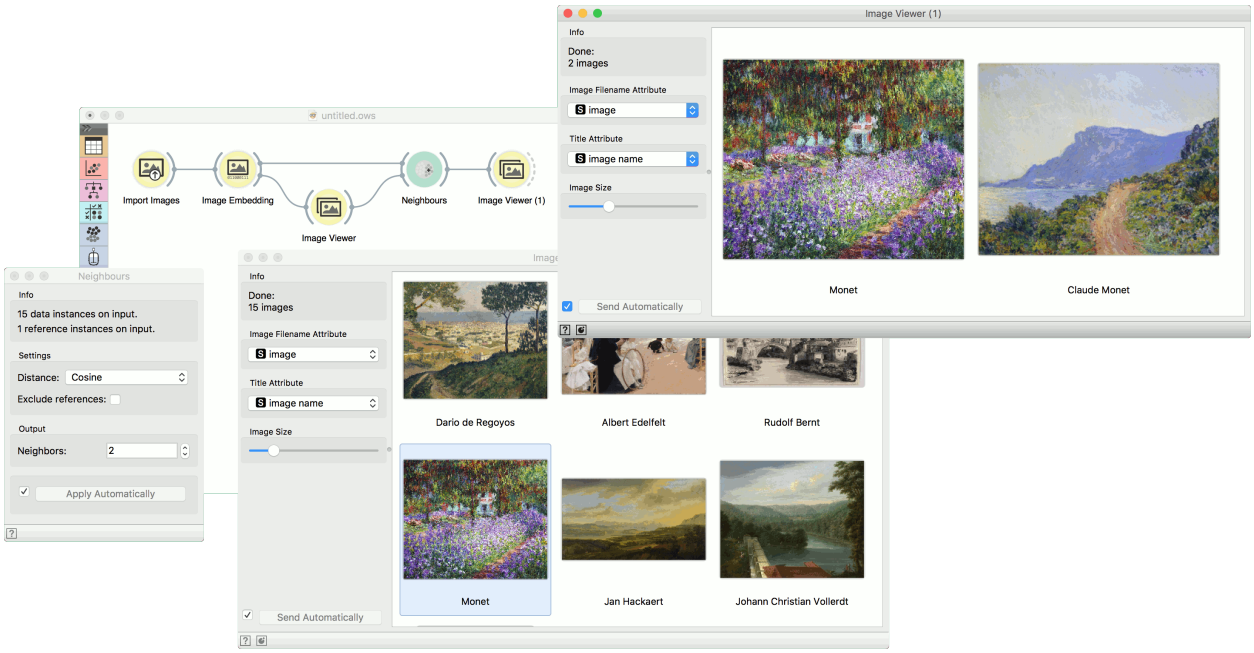
In the first example, we used *iris* data and passed it to **Neighbors** and to Data Table. In **Data Table**, we selected an instance of iris, that will serve as our reference, meaning we wish to retrieve 10 closest examples to the select data instance. We connect **Data Table** to **Neighbors** as well.

We can observe the results of neighbor computation in **Data Table (1)**, where we can see 10 closest images to our selected iris flower.



Another example requires the installation of Image Analytics add-on. We loaded 15 paintings from famous painters with **Import Images** widget and passed them to **Image Embedding**, where we selected *Painters* embedder.

Then the procedure is the same as above. We passed embedded images to **Image Viewer** and selected a painting from Monet to serve as our reference image. We passed the image to **Neighbors**, where we set the distance measure to *cosine*, ticked off *Exclude reference* and set the neighbors to 2. This allows us to find the actual closest neighbor to a reference painting and observe them side by side in **Image Viewer (1)**.



2.2 Visualize

2.2.1 Box Plot

Shows distribution of attribute values.

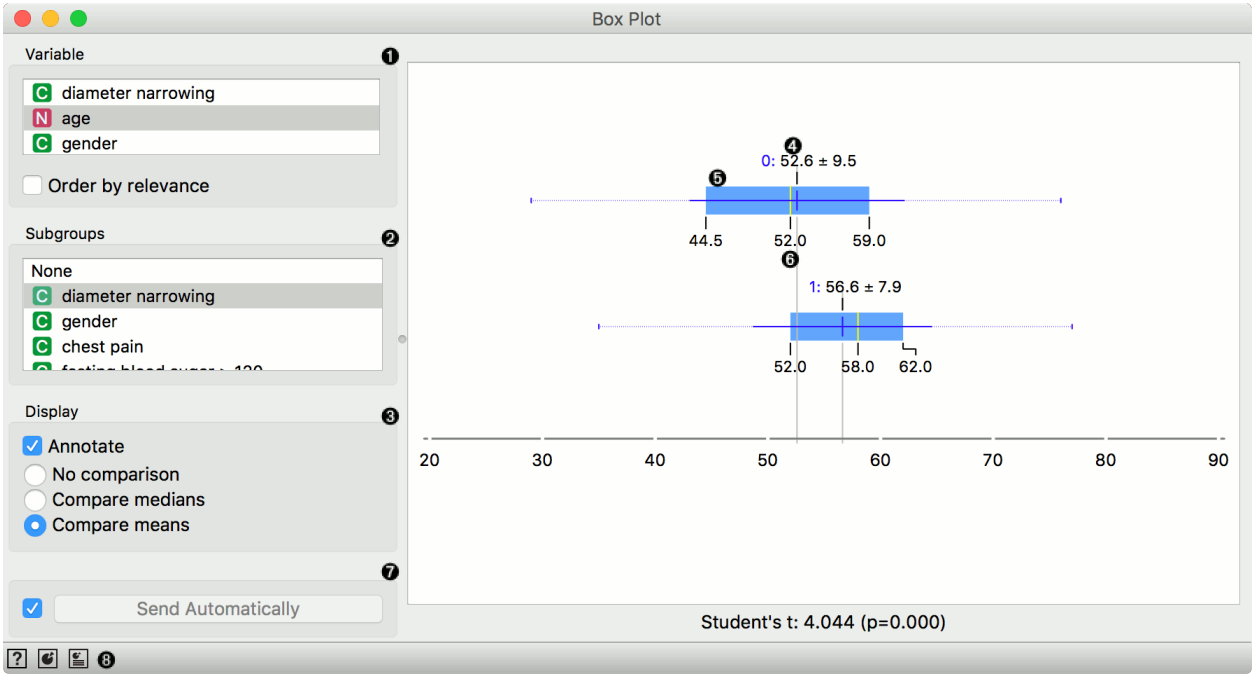
Inputs

- Data: input dataset

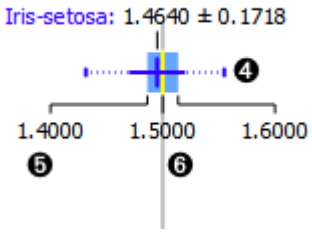
Outputs

- Selected Data: instances selected from the plot
- Data: data with an additional column showing whether a point is selected

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 plot. Tick *Order by relevance* to order variables by Chi2 or ANOVA over the selected subgroup.
2. Choose *Subgroups* to see box plots displayed by a discrete subgroup.
3. When instances are grouped by a subgroup, 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 subgroups.

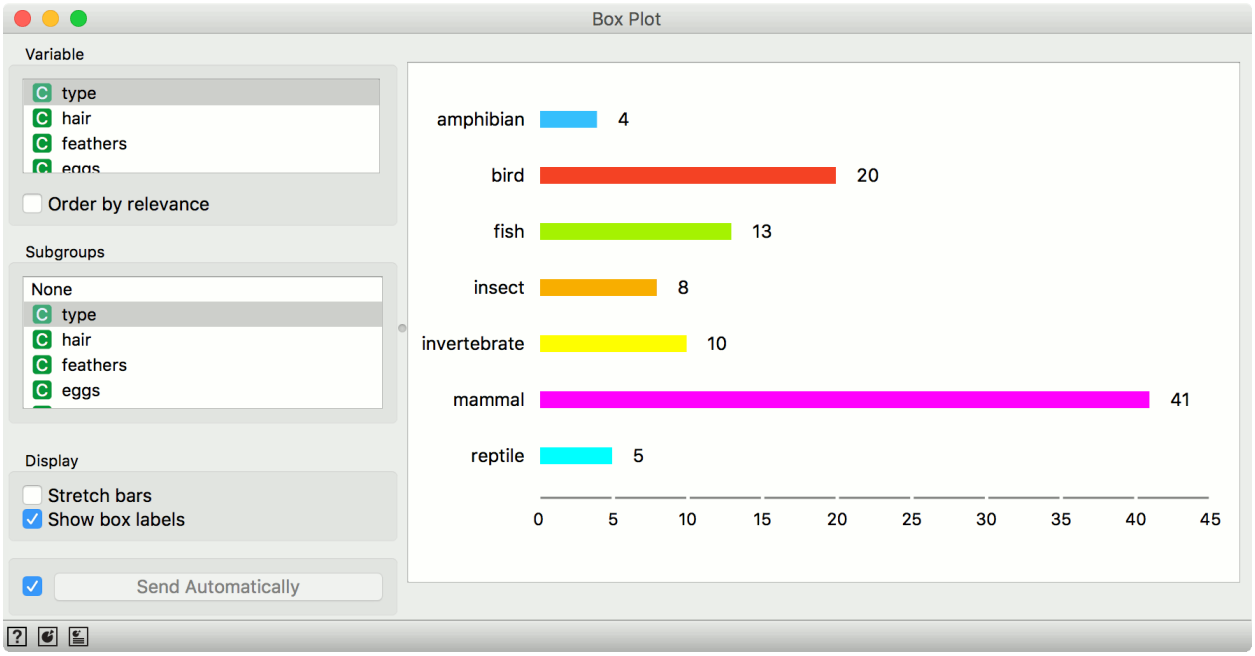


continuous

4. The mean (the dark blue vertical line). The thin blue line represents the standard deviation.

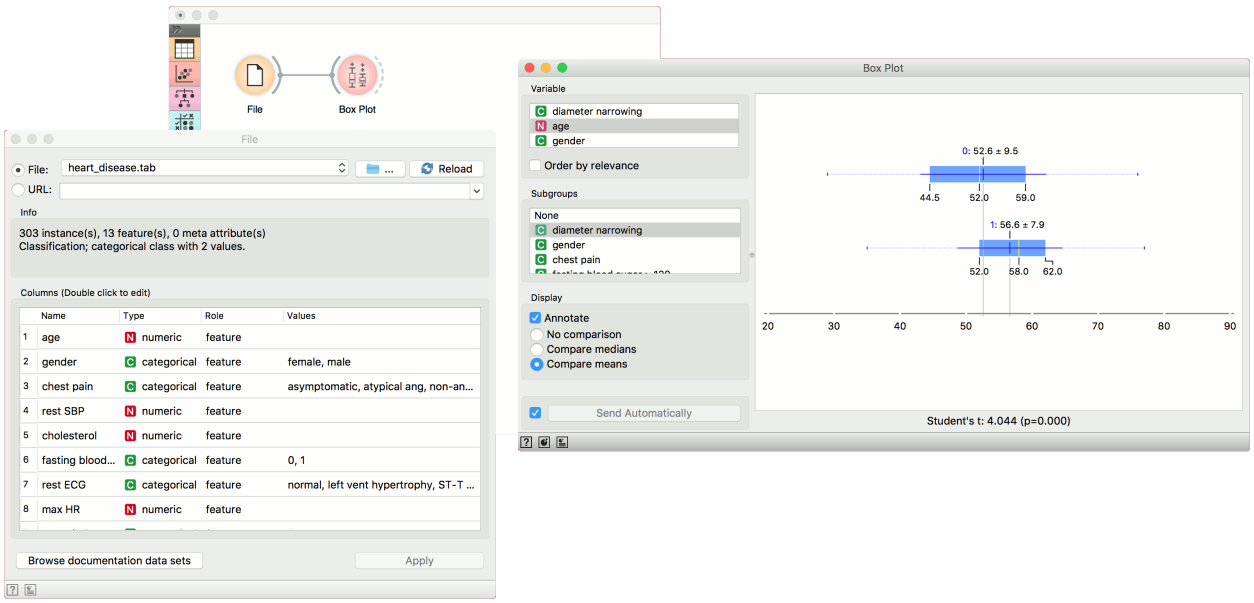
5. Values of the first (25%) and the third (75%) quantile. The blue highlighted area represents the values between the first and the third quartile.
6. The median (yellow vertical line).
7. If *Send automatically* is ticked, changes are communicated automatically. Alternatively, press *Send*.
8. Access help, save image or 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* dataset: there are 41 mammals, 13 fish, 20 birds and so on.



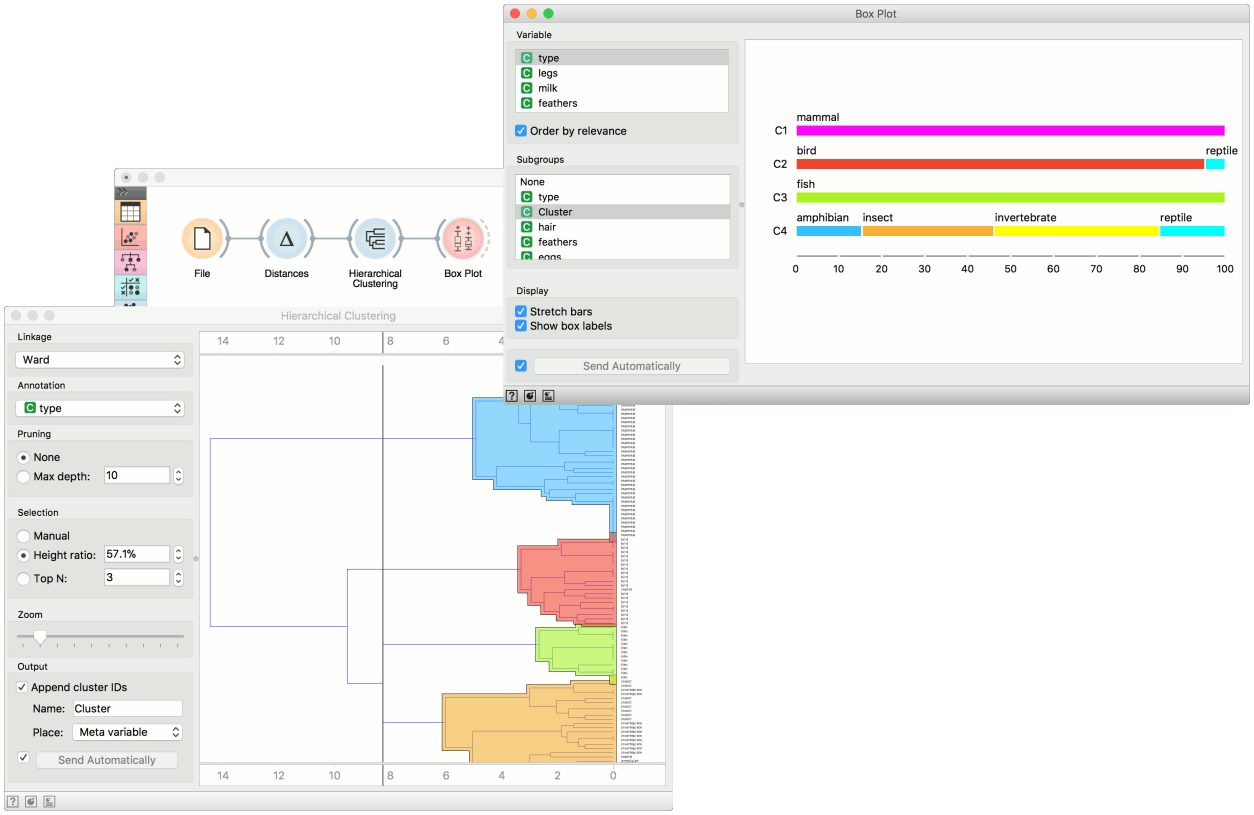
Example

The **Box Plot** widget is most commonly used immediately after the File widget to observe the statistical properties of a dataset. In the first example, we have used *heart-disease* data to inspect our variables.



Box Plot is also useful for finding the properties of a specific dataset, for instance a set of instances manually defined in another widget (e.g. Scatter Plot or instances belonging to some cluster or a classification tree node. Let us now use *zoo* data and create a typical clustering workflow with Distances and Hierarchical Clustering.

Now define the threshold for cluster selection (click on the ruler at the top). Connect **Box Plot** to **Hierarchical Clustering**, tick *Order by relevance* and select *Cluster* as a subgroup. This will order attributes by how well they define the selected subgroup, in our case a cluster. Seems like our clusters indeed correspond very well with the animal type!



2.2.2 Distributions

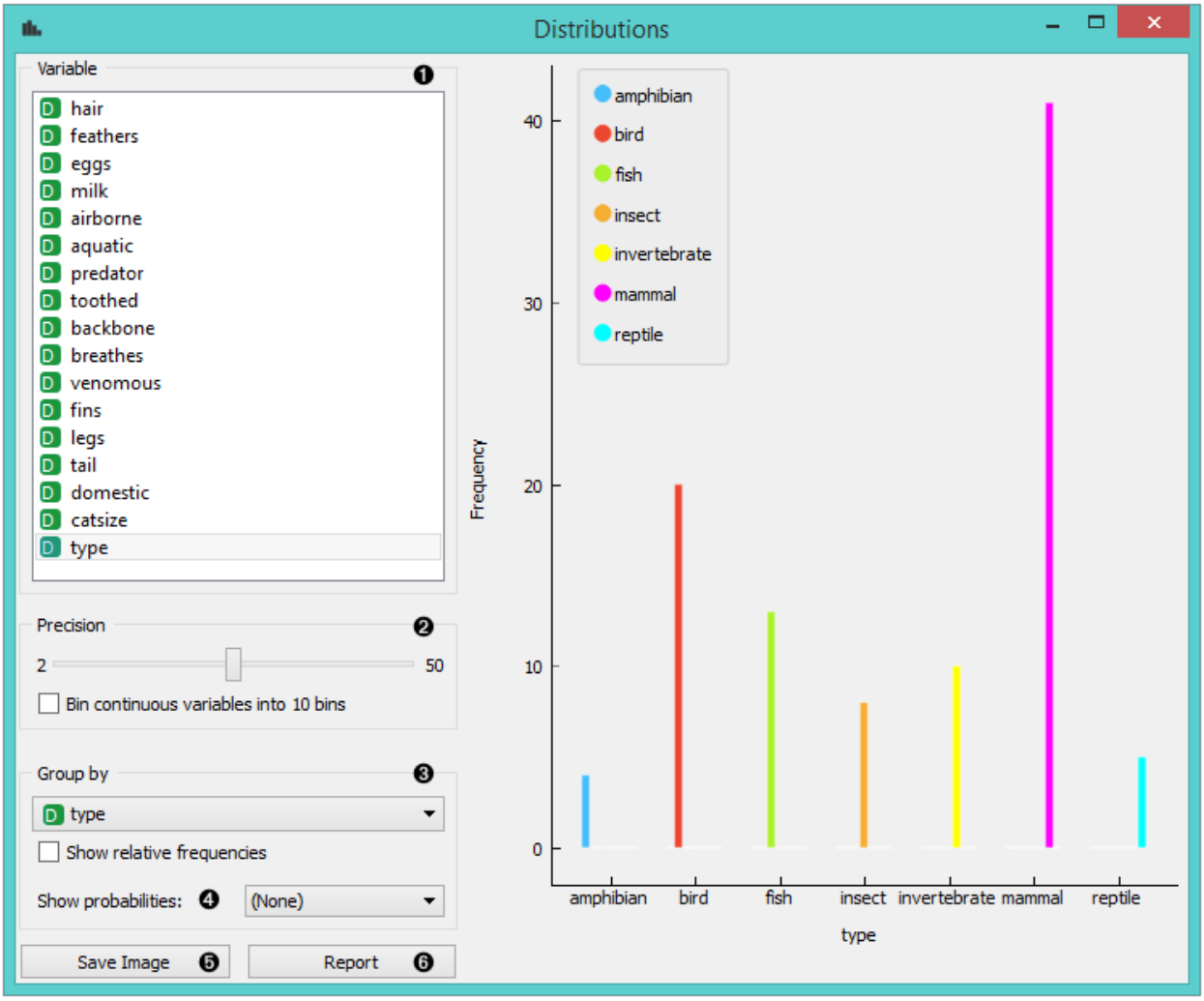
Displays value distributions for a single attribute.

Inputs

- Data: input dataset

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* dataset.

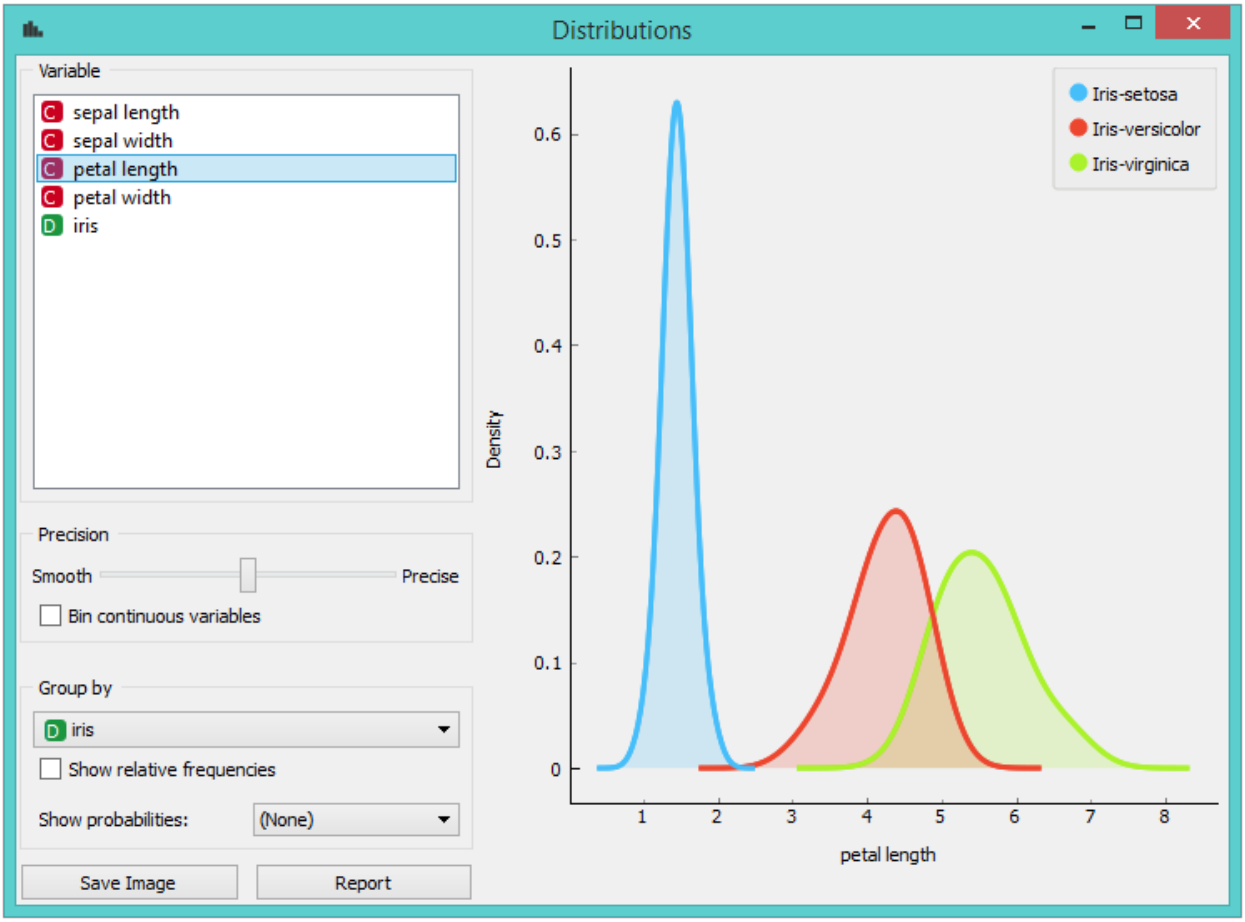


1. A list of variables for distributions display
2. If *Bin continuous variables* is ticked, the widget will discretize continuous variables by assigning them to intervals. The number of intervals is set by precision scale. Alternatively, you can set smoothness for the distribution curves of continuous 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 dataset.
4. Show probabilities.

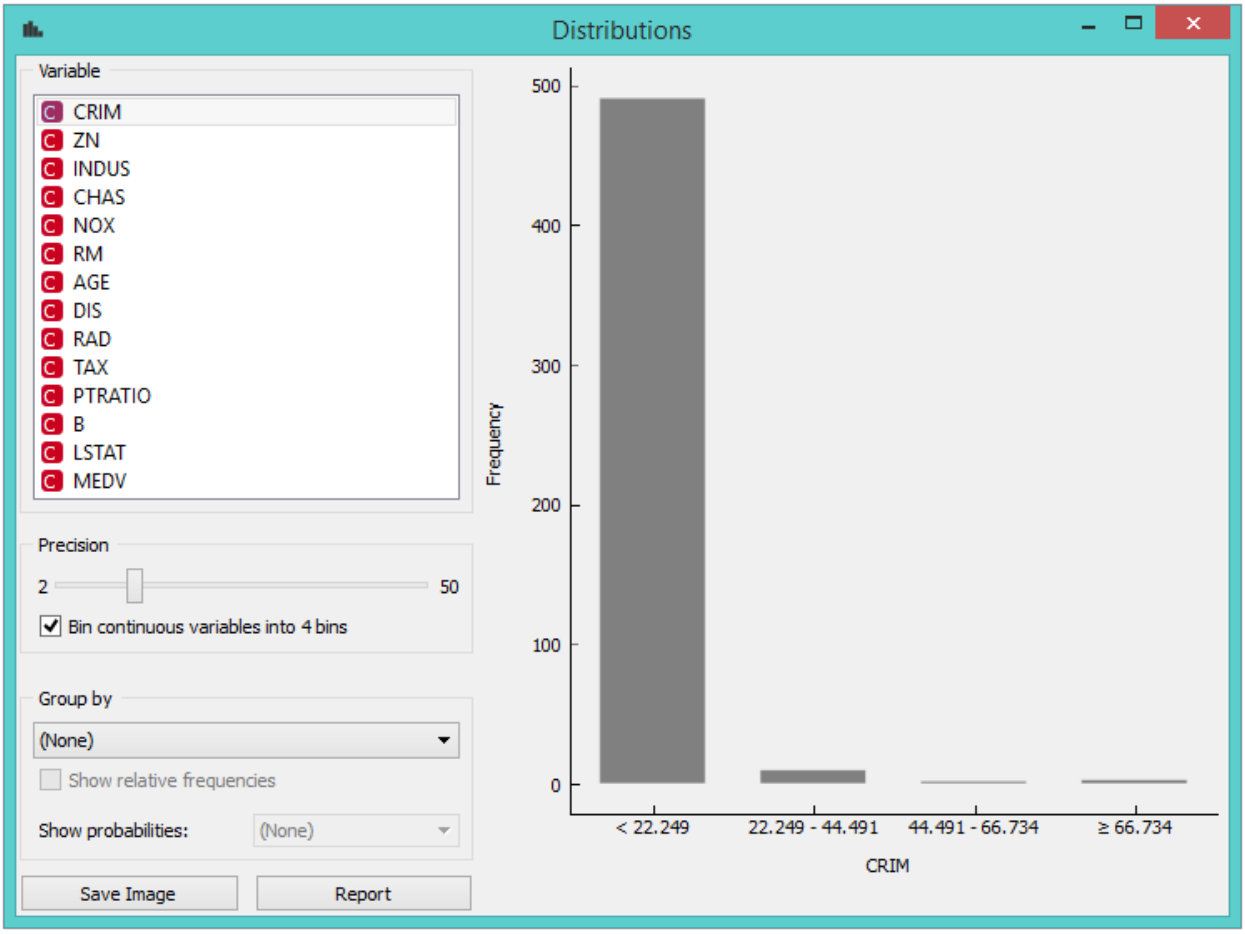
5. *Save image* saves the graph to your computer in a .svg or .png format.
6. Produce a report.

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* dataset.



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* dataset.



2.2.3 Heat Map

Plots a heat map for a pair of attributes.

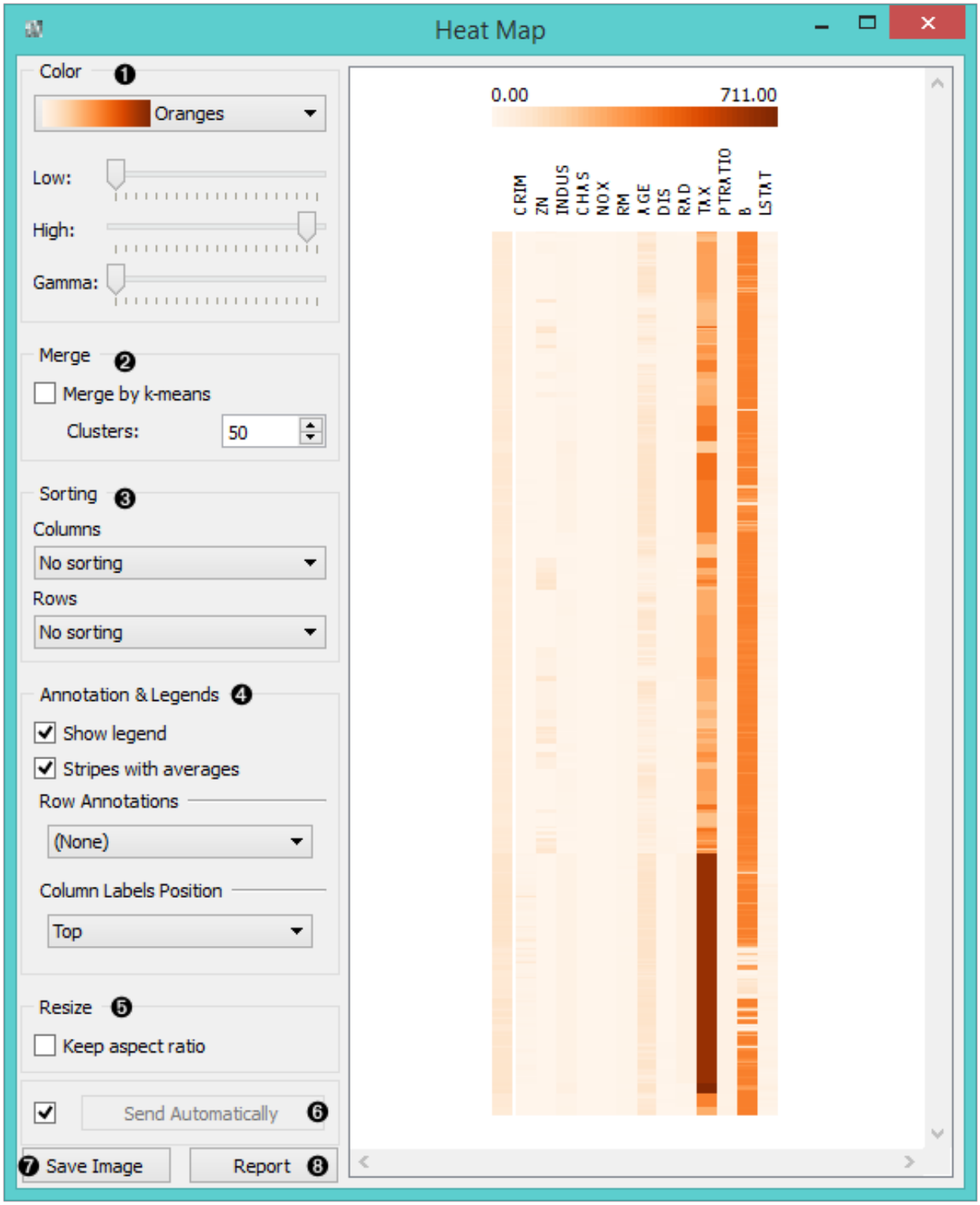
Inputs

- Data: input dataset

Outputs

- Selected Data: instances selected from the plot

Heat map is a graphical method for visualizing attribute values by class in a two-way matrix. It only works on datasets 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 dataset)
- **Clustering** (clusters data by similarity)
- **Clustering with ordered leaves** (maximizes the sum of similarities of adjacent elements)

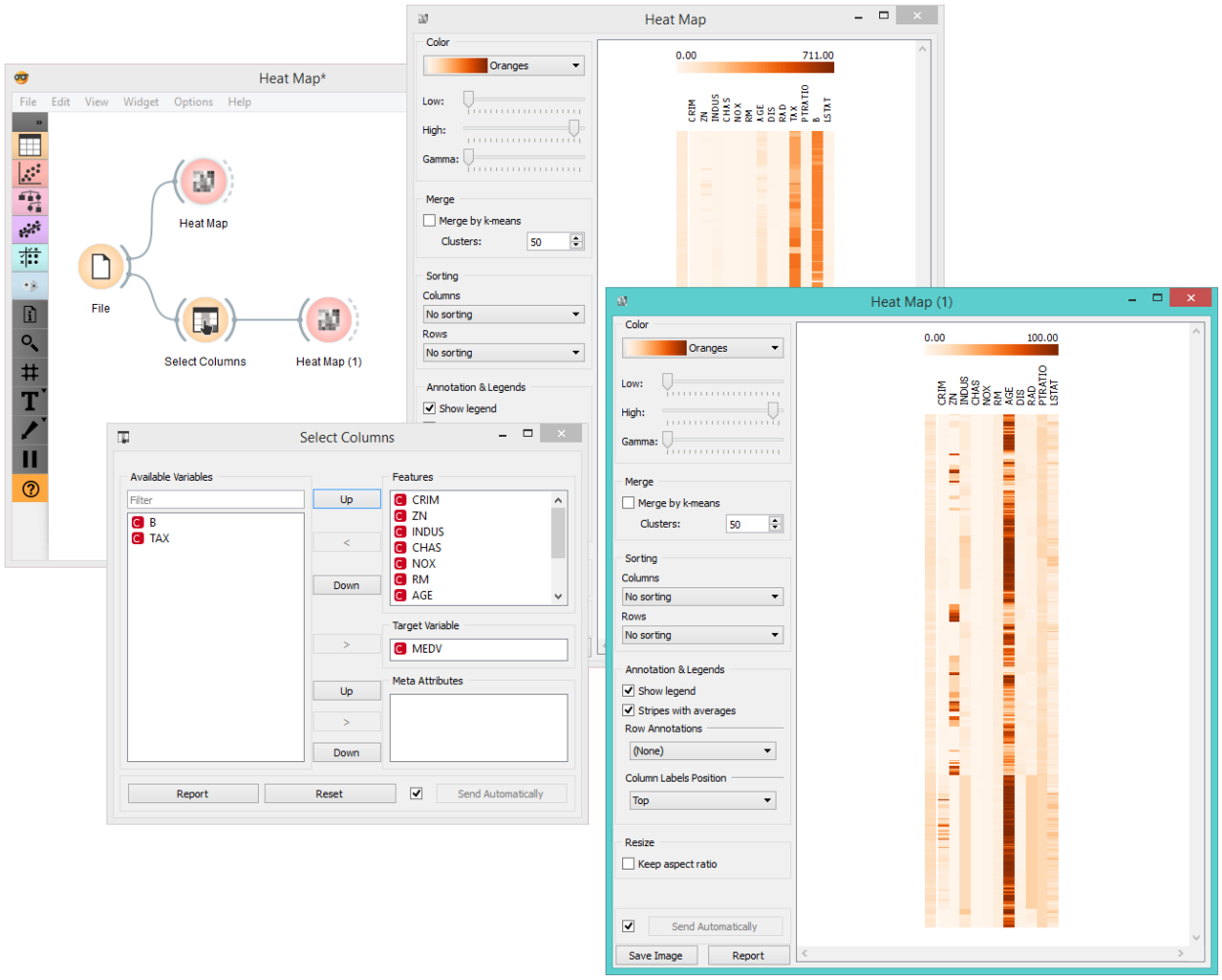
4. Set what is displayed in the plot in **Annotation & Legend**.
   - If *Show legend* is ticked, a color chart will be displayed above the map.
   - If *Stripes with averages* is ticked, a new line with attribute averages will be displayed on the left.
   - **Row Annotations** adds annotations to each 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* dataset. The aforementioned dataset 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 dataset. 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 Dataset

2.2.4 Scatter Plot

Scatter plot visualization with explorative analysis and intelligent data visualization enhancements.

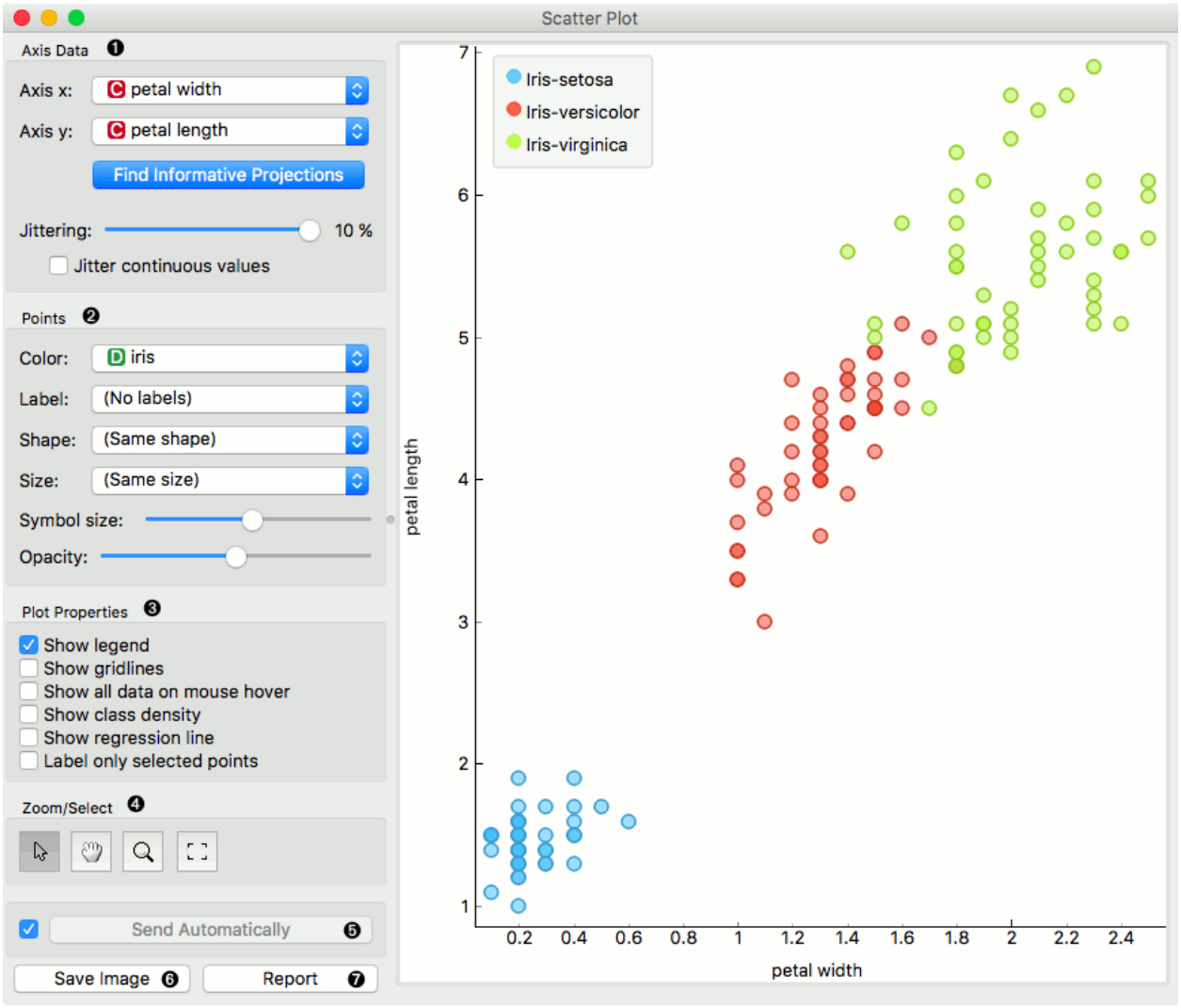
Inputs

- Data: input dataset
- Data Subset: subset of instances
- Features: list of attributes

Outputs

- Selected Data: instances selected from the plot
- Data: data with an additional column showing whether a point is selected

The **Scatter Plot** widget provides a 2-dimensional scatter plot 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 scatter plot of the *Iris* dataset with the coloring matching of the class attribute.

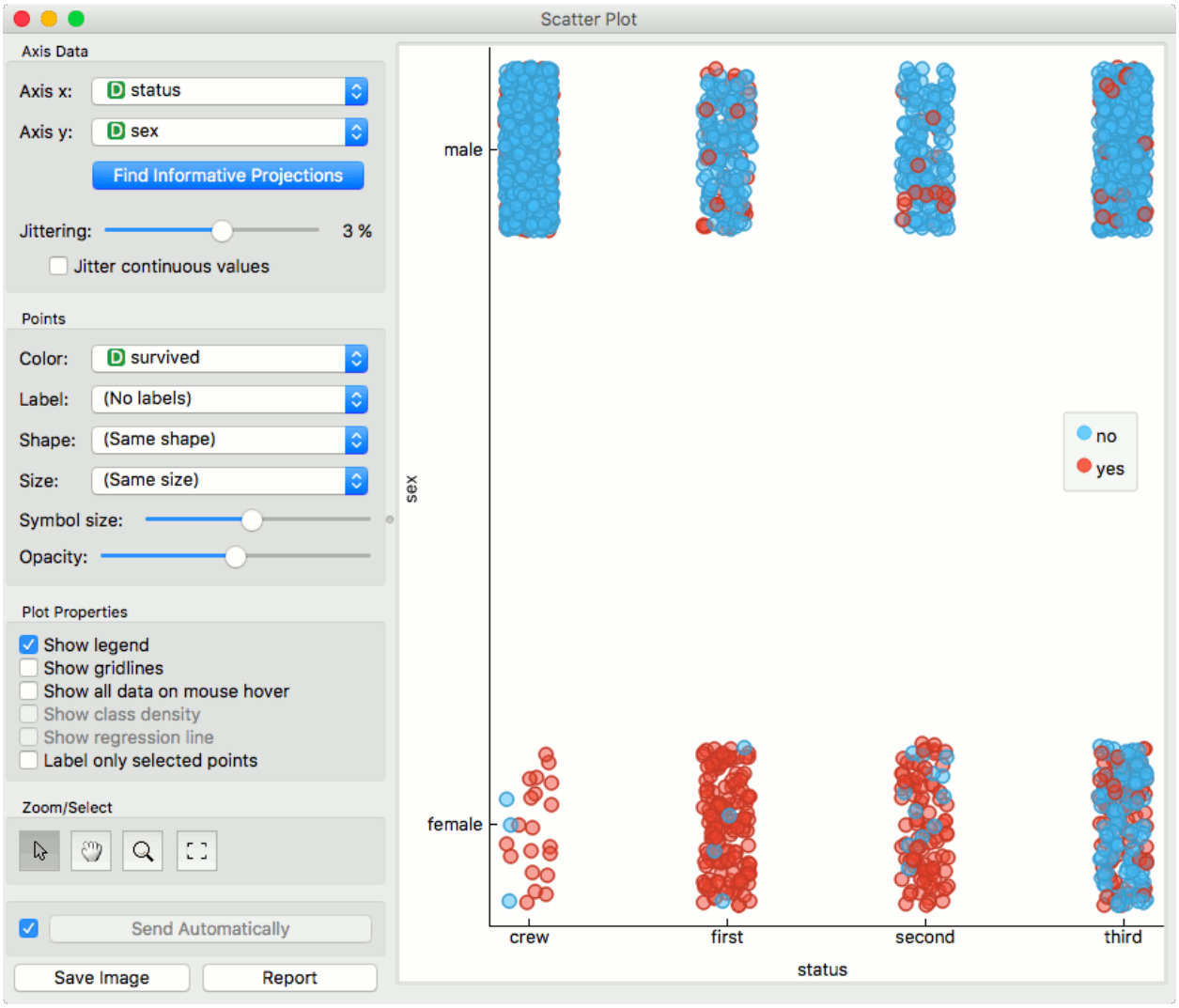


1. Select the x and y attribute. Optimize your projection by using **Rank Projections**. This feature scores attribute pairs by average classification accuracy and returns the top scoring pair with a simultaneous visualization update. Set jittering to prevent the dots overlapping. If *Jitter continuous 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.

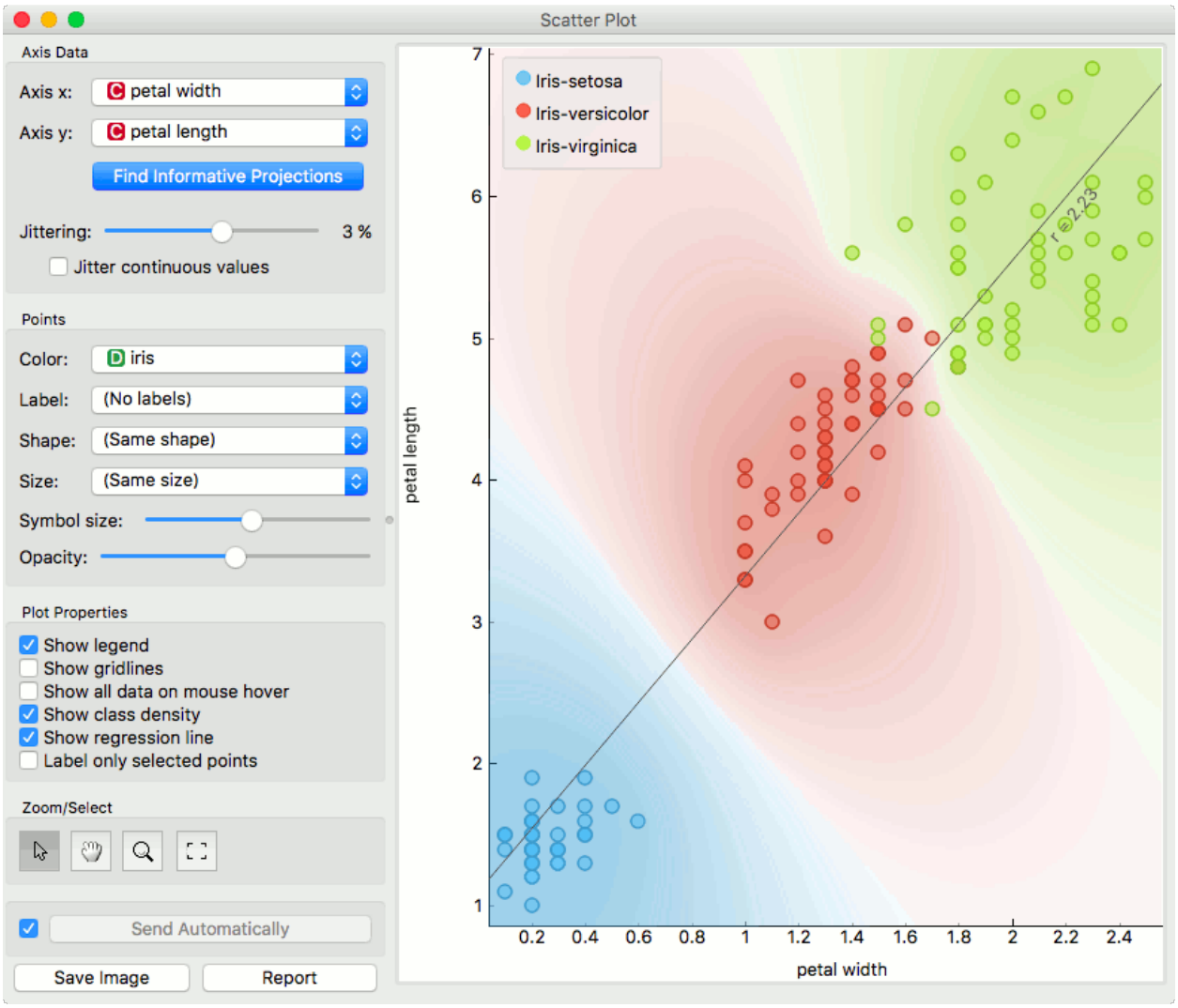
- *Show class density* colors the graph by class (see the screenshot below).
- *Show regression line* draws the regression line for pair of continuous attributes.
- *Label only selected points* allows you to select individual data instances and label them.

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 scatter plot for the Titanic dataset, reporting on the gender of the passengers and the traveling class is shown below; without jittering, the scatter plot would display only eight distinct points.



Here is an example of the **Scatter Plot** widget if the *Show class density* and *Show regression line* boxes are ticked.



Intelligent Data Visualization

If a dataset has many attributes, it is impossible to manually scan through all the pairs to find interesting or useful scatter plots. Orange implements intelligent data visualization with the **Find Informative Projections** option in the widget.

If a categorical variable is selected in the Color section, the score is computed as follows. For each data instance, the method finds 10 nearest neighbors in the projected 2D space, that is, on the combination of attribute pairs. It then checks how many of them have the same color. The total score of the projection is then the average number of same-colored neighbors.

Computation for continuous colors is similar, except that the coefficient of determination is used for measuring the local homogeneity of the projection.

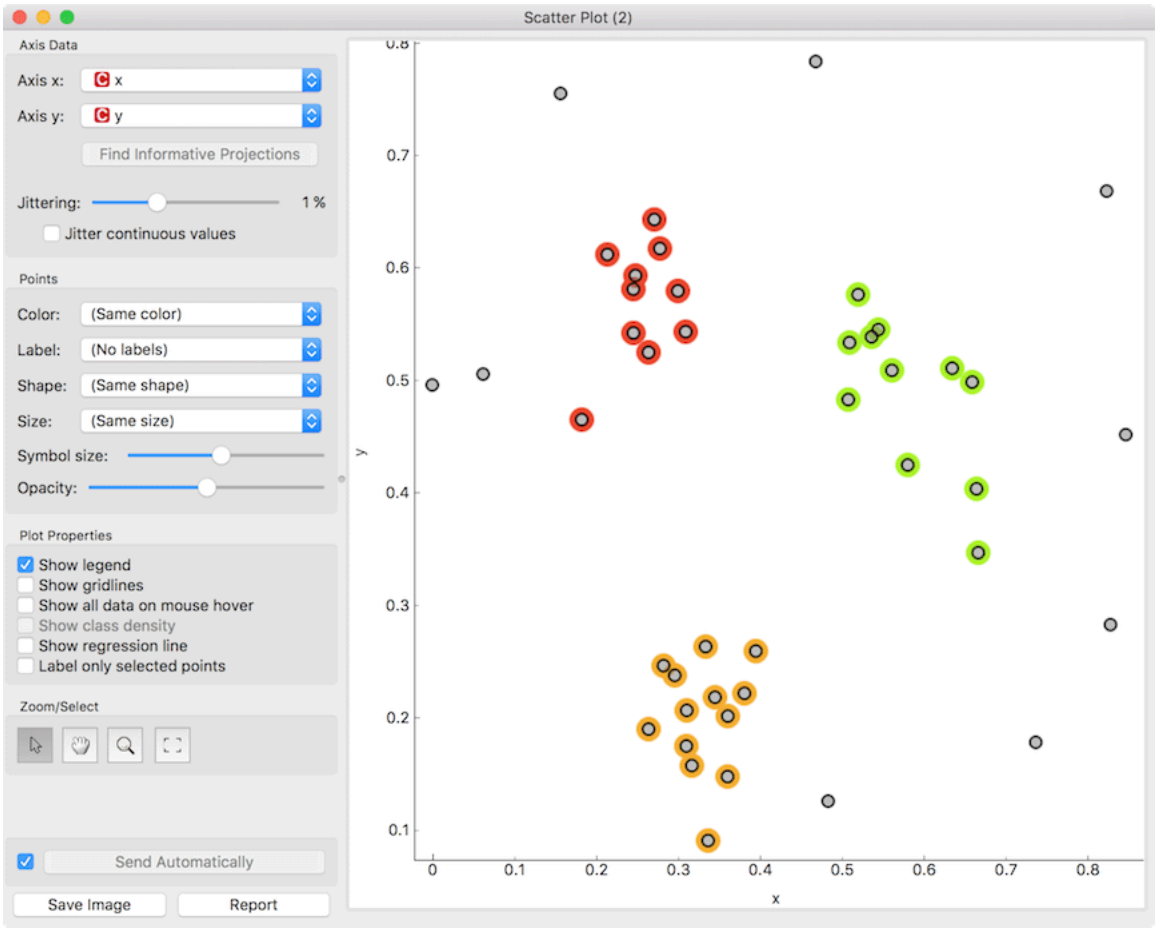
To use this method, go to the *Find Informative Projections* option in the widget, open the subwindow and press *Start Evaluation*. The feature will return a list of attribute pairs by average classification accuracy score.

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

Selection

Selection can be used to manually defined subgroups in the data. Use Shift modifier when selecting data instances to put them into a new group. Shift + Ctrl (or Shift + Cmd on macOs) appends instances to the last group.

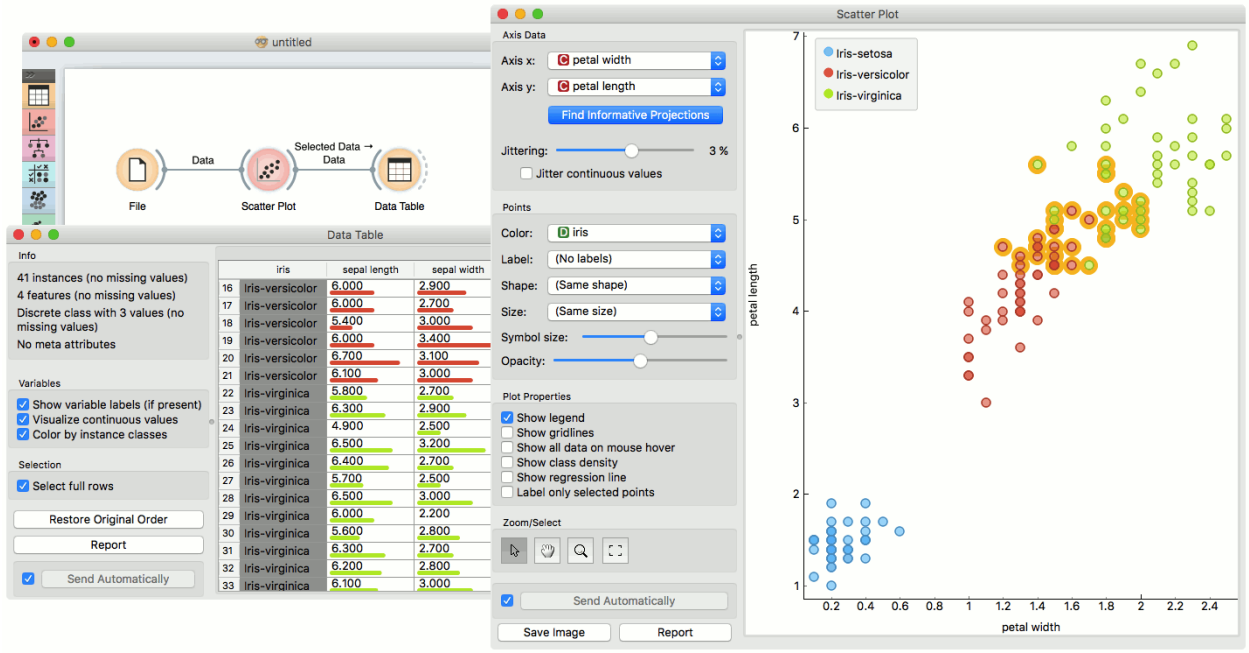
Signal data outputs a data table with an additional column that contains group indices.



Explorative Data Analysis

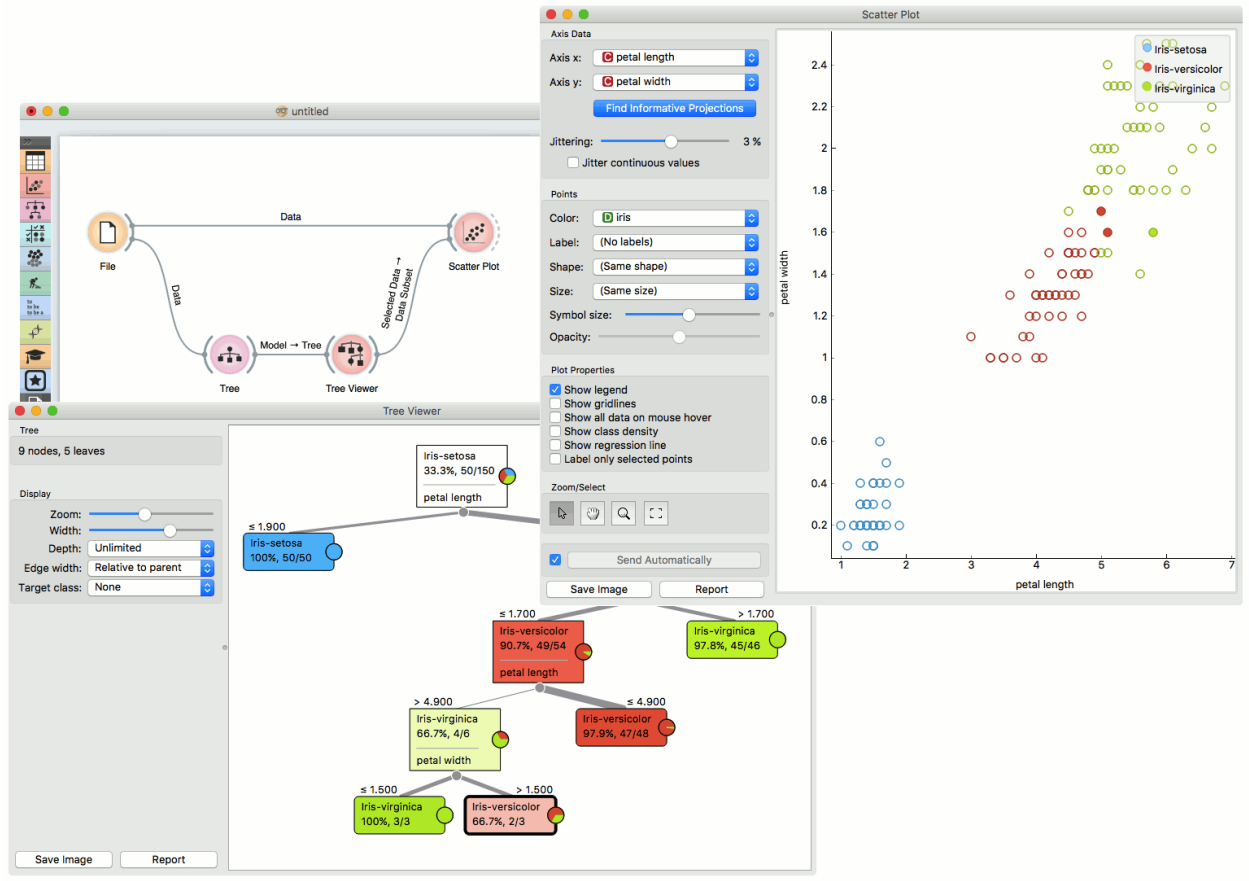
The **Scatter Plot**, as the rest of Orange widgets, supports zooming-in and out of part of the plot and a manual selection of data instances. These functions are available in the lower left corner of the widget.

The default tool is *Select*, which selects data instances within the chosen rectangular area. *Pan* enables you to move the scatter plot 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 **Scatter Plot** can be combined with any widget that outputs a list of selected data instances. In the example below, we combine Tree and **Scatter Plot** to display instances taken from a chosen decision tree node (clicking on any node of the tree will send a set of selected data instances to the scatterplot and mark selected instances with filled symbols).



References

Gregor Leban and Blaz Zupan and Gaj Vidmar and Ivan Bratko (2006) VizRank: Data Visualization Guided by Machine Learning. Data Mining and Knowledge Discovery, 13 (2). pp. 119-136. Available here.

2.2.5 Line Plot

Visualization of data profiles (e.g., time series).

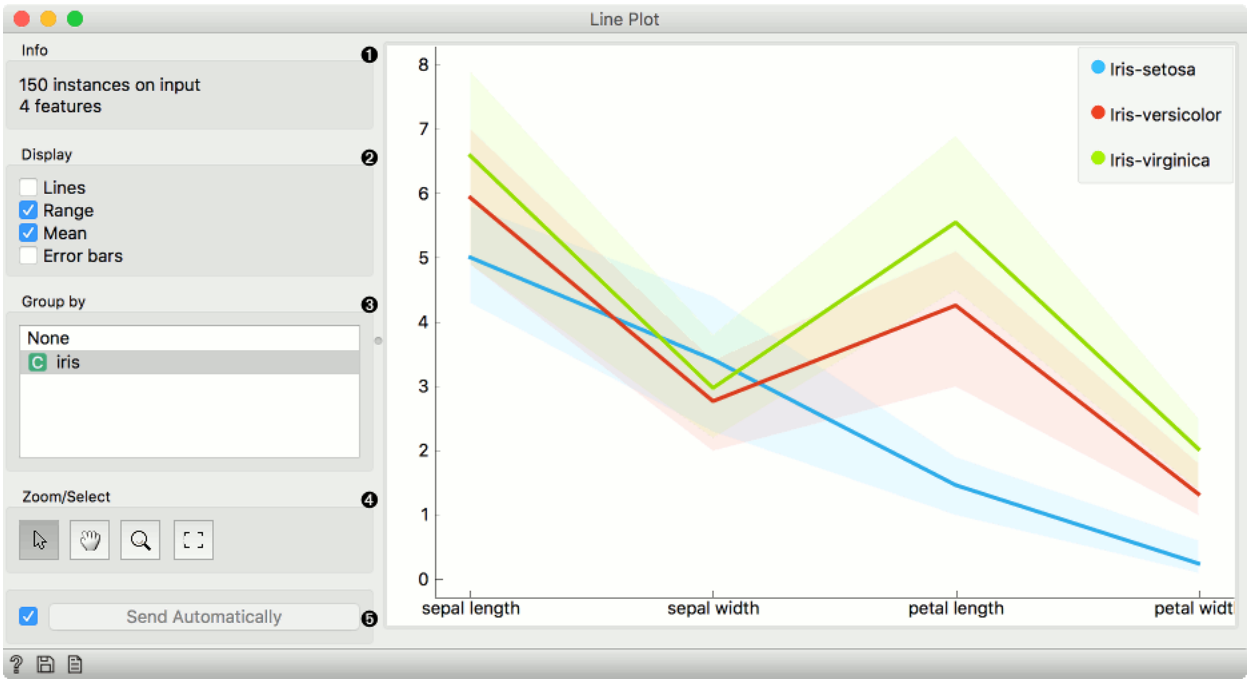
Inputs

- Data: input dataset
- Data Subset: subset of instances

Outputs

- Selected Data: instances selected from the plot
- Data: data with an additional column showing whether a point is selected

Line plot a type of plot which displays the data as a series of points, connected by straight line segments. It only works for numerical data, while categorical can be used for grouping of the data points.



1. Information on the input data.
2. Select what you wish to display:
   - Lines show individual data instances in a plot.
   - Range shows the range of data points between 10th and 90th percentile.
   - Mean adds the line for mean value. If group by is selected, means will be displayed per each group value.
   - Error bars show the standard deviation of each attribute.
3. Select a categorical attribute to use for grouping of data instances. Use None to show ungrouped data.
4. *Select, zoom, pan and zoom to fit* are the options for exploring the graph. The manual selection of data instances works as a line selection, meaning the data under the selected line plots will be sent on the output. Scroll in or out for zoom.
5. If *Send Automatically* is ticked, changes are communicated automatically. Alternatively, click *Send*.

Example

Line Plot is a standard visualization widget, which displays data profiles, normally of ordered numerical data. In this simple example, we will display the *iris* data in a line plot, grouped by the iris attribute. The plot shows how petal length nicely separates between class values.

If we observe this in a Scatter Plot, we can confirm this is indeed so. Petal length is an interesting attribute for separation of classes, especially when enhanced with petal width, which is also nicely separated in the line plot.

2.2.6 Venn Diagram

Plots a Venn diagram for two or more data subsets.

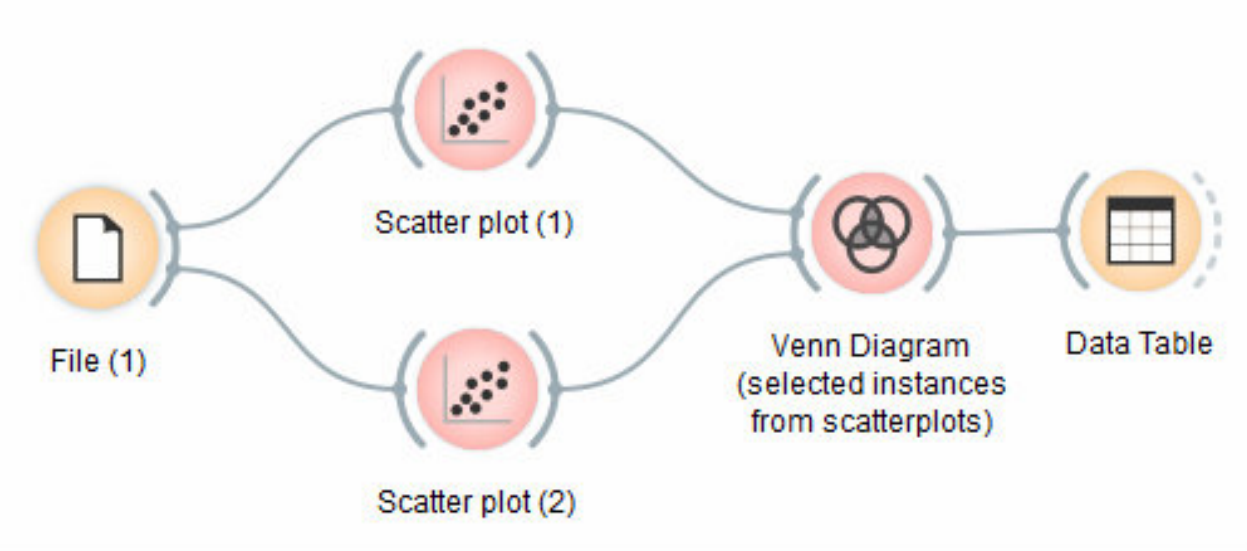
Inputs

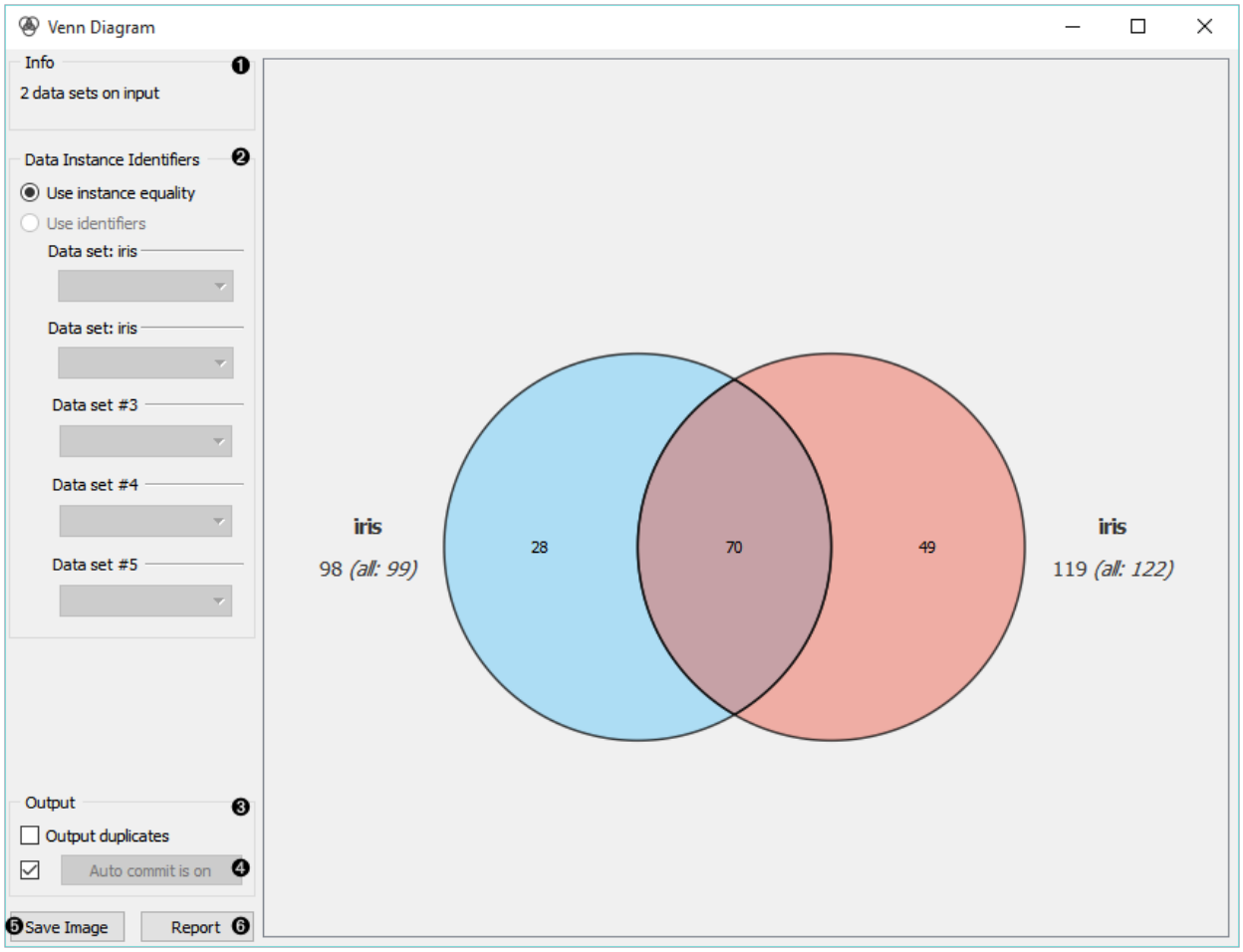
- Data: input dataset

Outputs

- Selected Data: instances selected from the plot

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



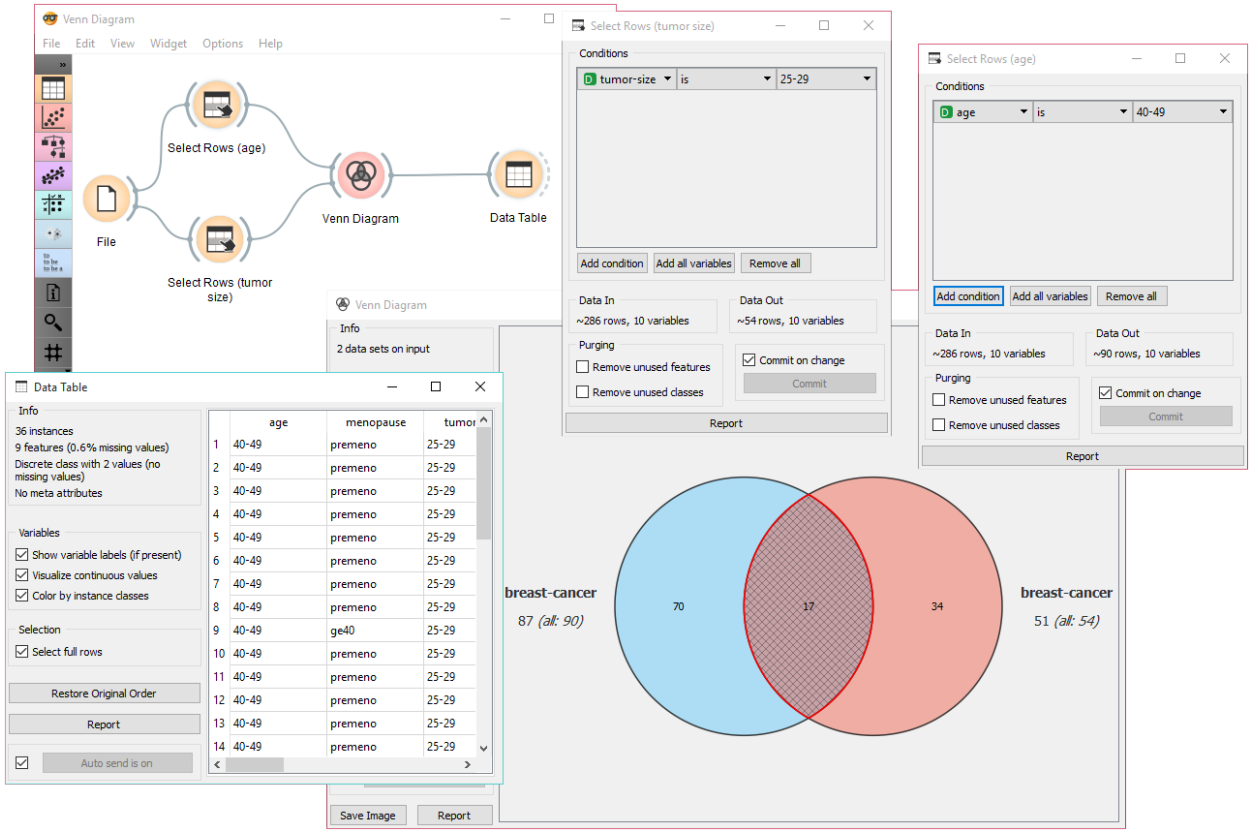


1. Information on the input data.
2. Select the identifiers by which to compare the data.
3. Tick *Output duplicates* if you wish to remove duplicates.
4. If *Auto commit* is on, changes are automatically communicated to other widgets. Alternatively, click *Commit*.

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

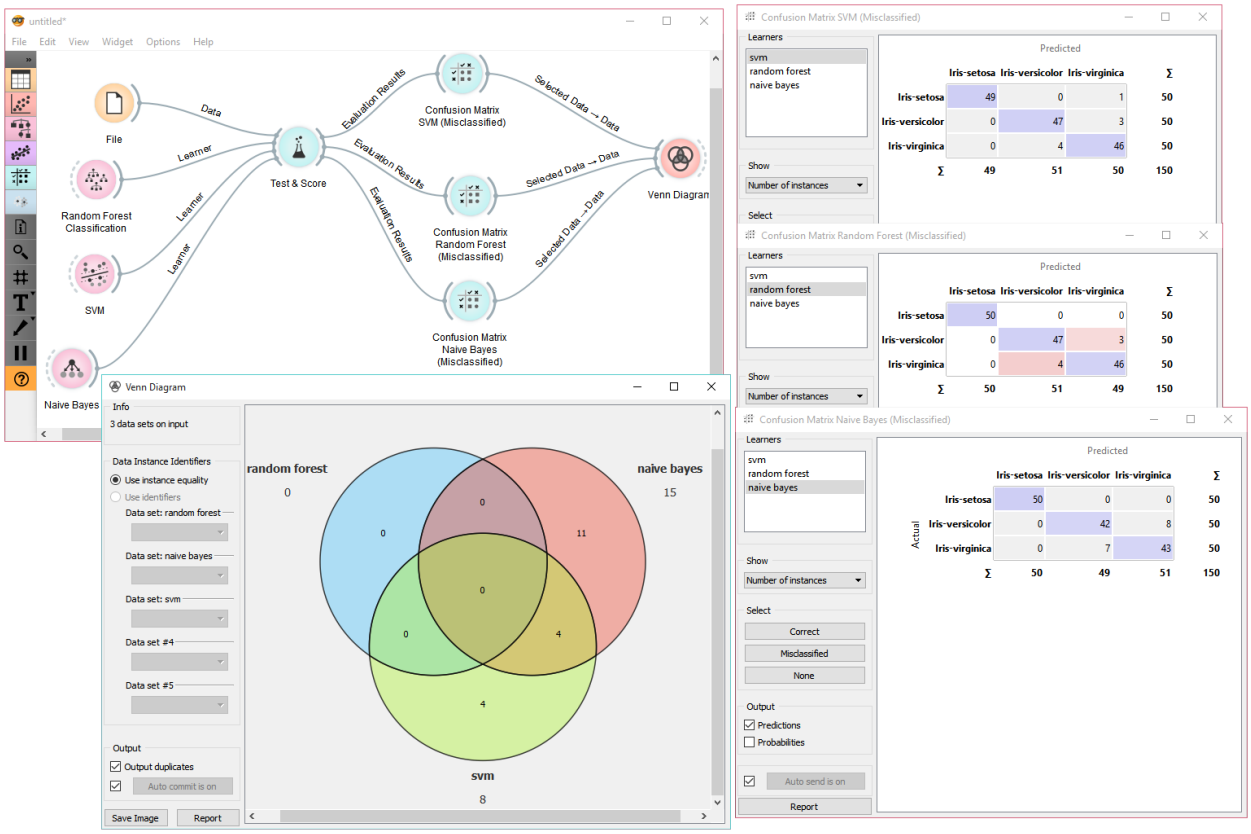
Examples

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* dataset 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 and Random Forest, 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. This is represented as an intersection of all three circles. Click on the intersection to see this two instances marked in the Scatter Plot widget. Try selecting different diagram sections to see how the scatter plot visualization changes.



2.2.7 Linear Projection

A linear projection method with explorative data analysis.

Inputs

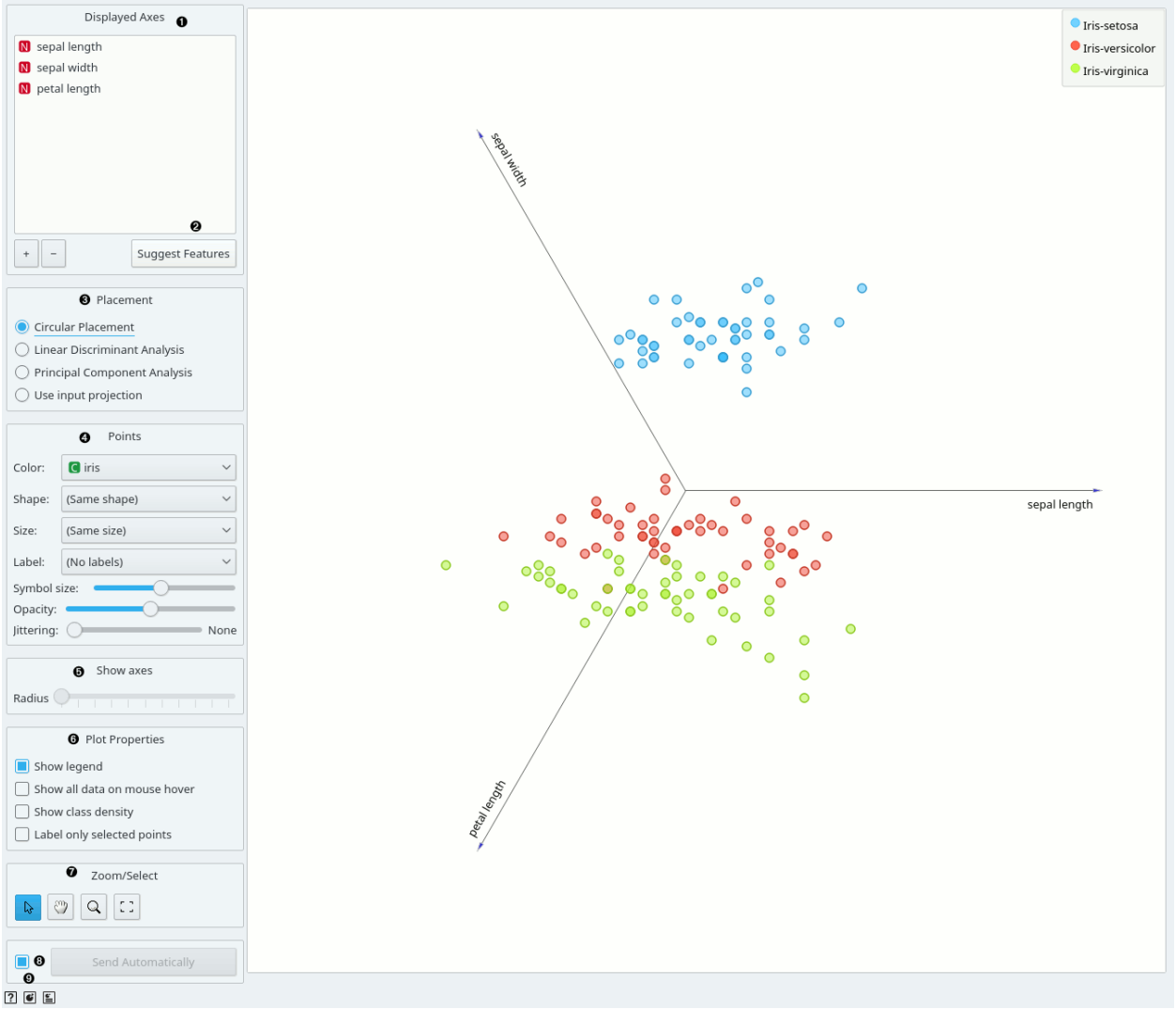
- Data: input dataset
- Data Subset: subset of instances
- Projection: custom projection vectors

Outputs

- Selected Data: instances selected from the plot
- Data: data with an additional column showing whether a point is selected
- Components: projection vectors

This widget displays linear projections of class-labeled data. It supports various types of projections such as circular, linear discriminant analysis, principal component analysis, and custom projection.

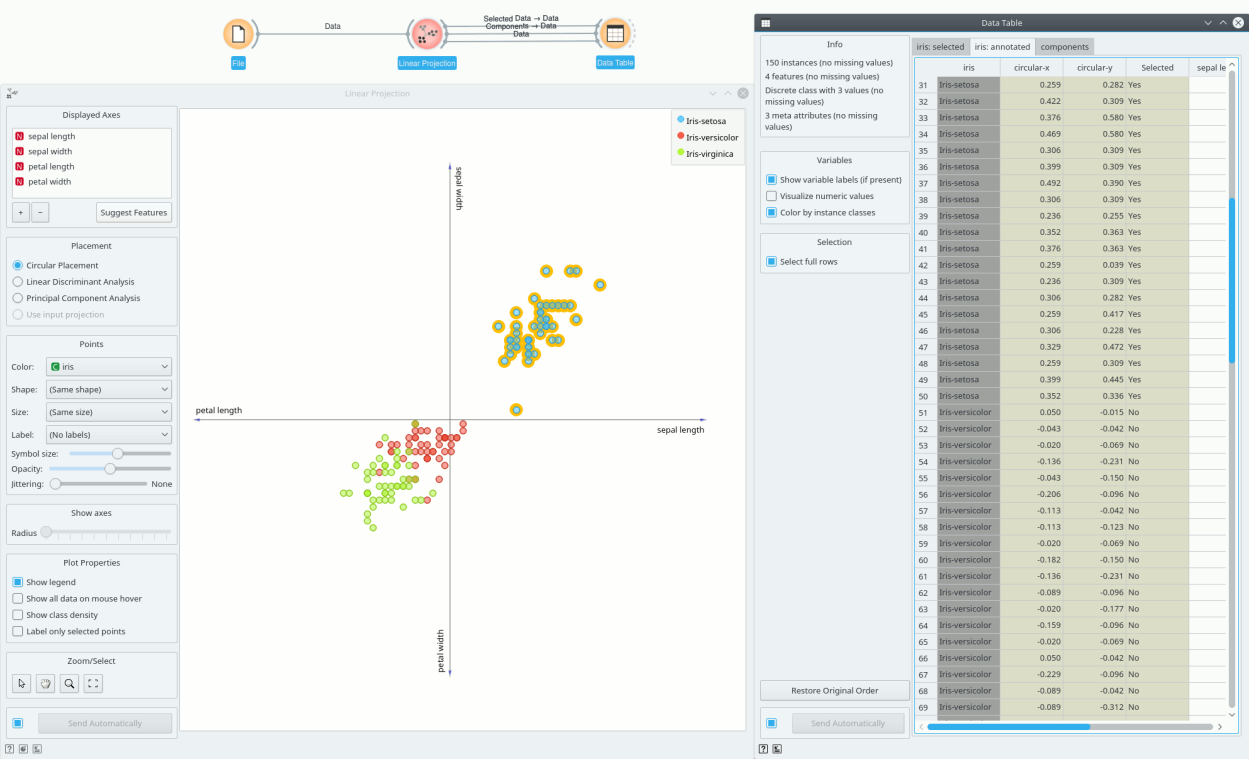
Consider, for a start, a projection of the *Iris* dataset shown below. Notice that it is the sepal width and sepal length that already separate *Iris setosa* from the other two, while the petal length is the attribute best separating *Iris versicolor* from *Iris virginica*.



1. Axes in the projection that are displayed and other available axes.
2. Optimize your projection by using **Suggest Features**. This feature scores attributes by average classification accuracy and returns the top scoring attributes with a simultaneous visualization update.
3. Choose the type of projection.
4. Axes inside a circle are hidden. Circle radius can be be changed using a slider.
5. Adjust *plot properties*:
   - Set jittering to prevent the dots from overlapping (especially for discrete attributes).
   - *Show legend* displays a legend on the right. Click and drag the legend to move it.
   - *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.
6. *Select, zoom, pan* and *zoom to fit* are the options for exploring the graph. Manual selection of data instances works as an angular/square selection tool. Double click to move the projection. Scroll in or out for zoom.
7. If *Send automatically* is ticked, changes are communicated automatically. Alternatively, press *Send*.
8. *Save Image* saves the created image to your computer in a .svg or .png format. 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.

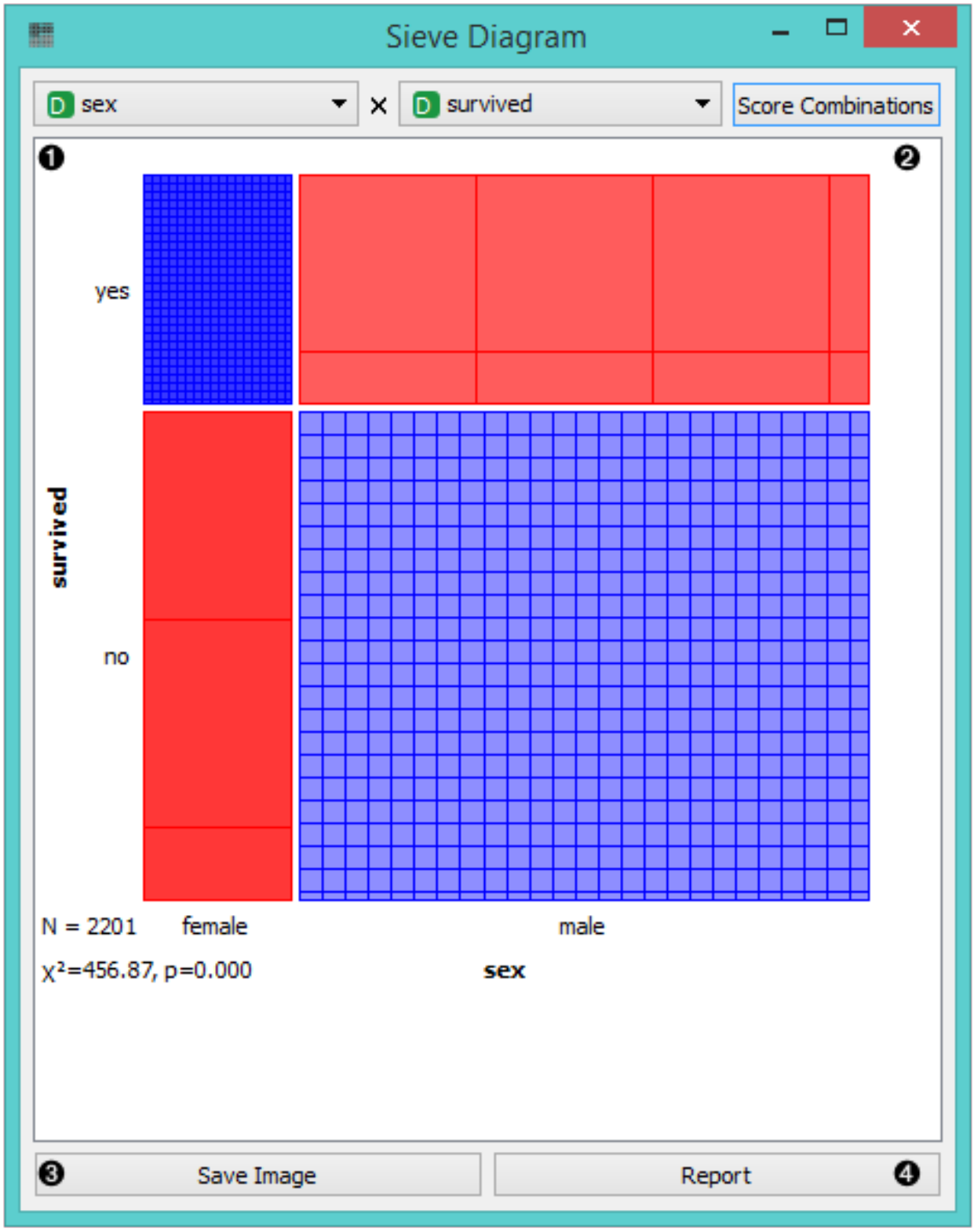
2.2.8 Sieve Diagram

Plots a sieve diagram for a pair of attributes.

Inputs

- Data: input dataset

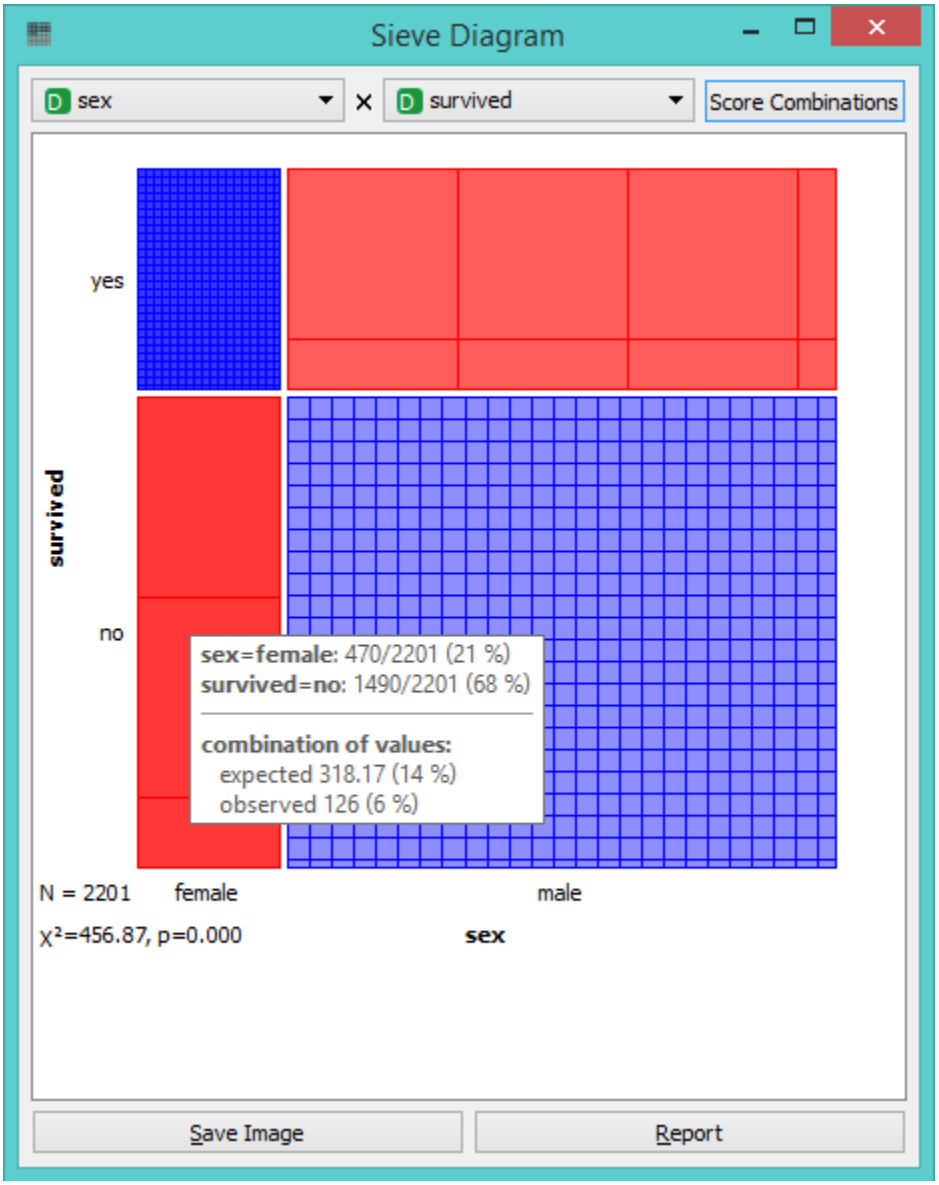
A **Sieve Diagram** is a graphical method for visualizing frequencies in a two-way contingency table and comparing them to expected frequencies under assumption of independence. It was proposed by Riedwyl and Schüpbach in a technical report in 1983 and later called a parquet diagram (Riedwyl and Schüpbach 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).



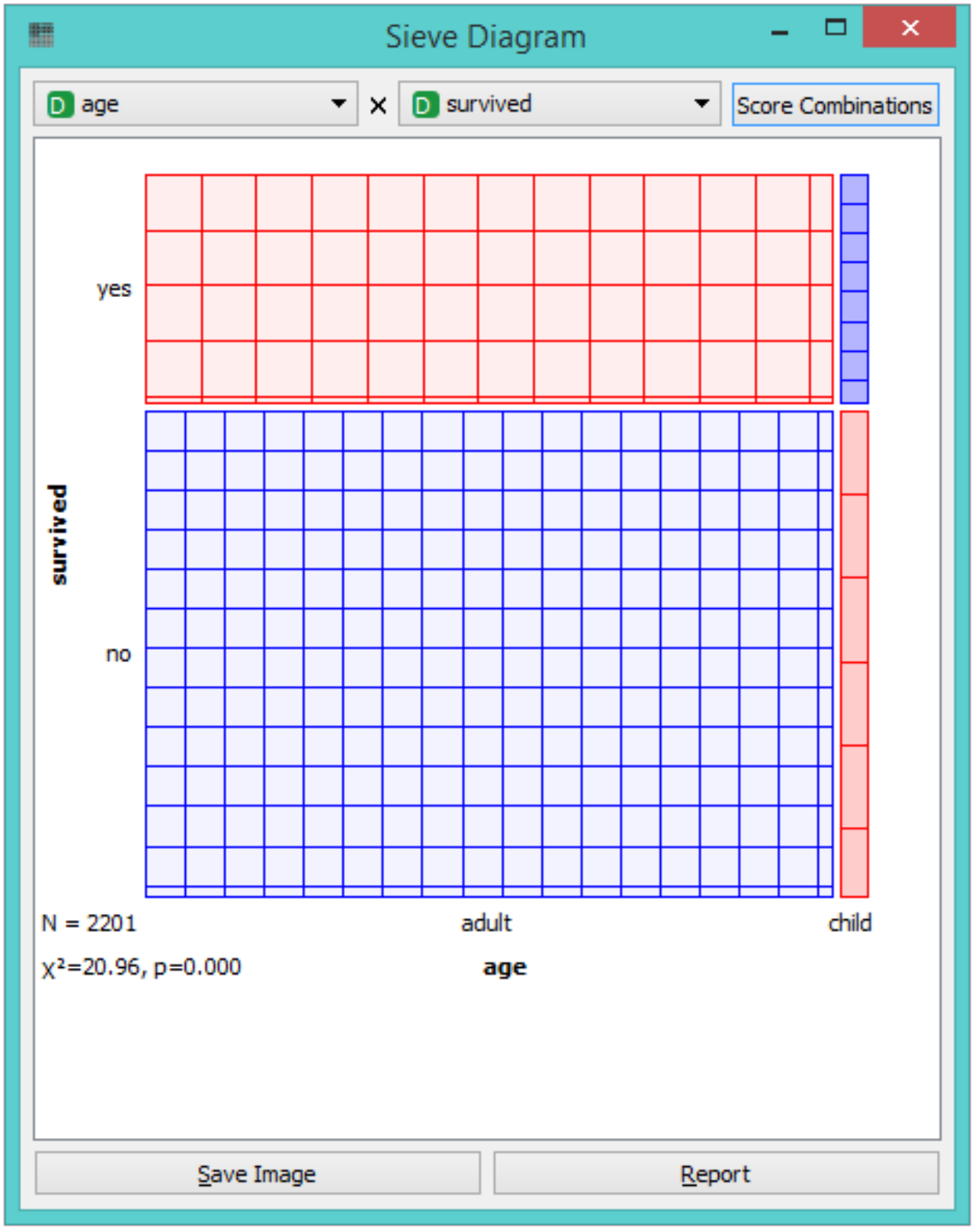
1. Select the attributes you want to display in the sieve plot.
2. Score combinations enables you to fin the best possible combination of attributes.
3. *Save Image* saves the created image to your computer in a .svg or .png format.

4. Produce a report.

The snapshot below shows a sieve diagram for the *Titanic* dataset and has the attributes *sex* and *survived* (the latter is a class attribute in this dataset). 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).

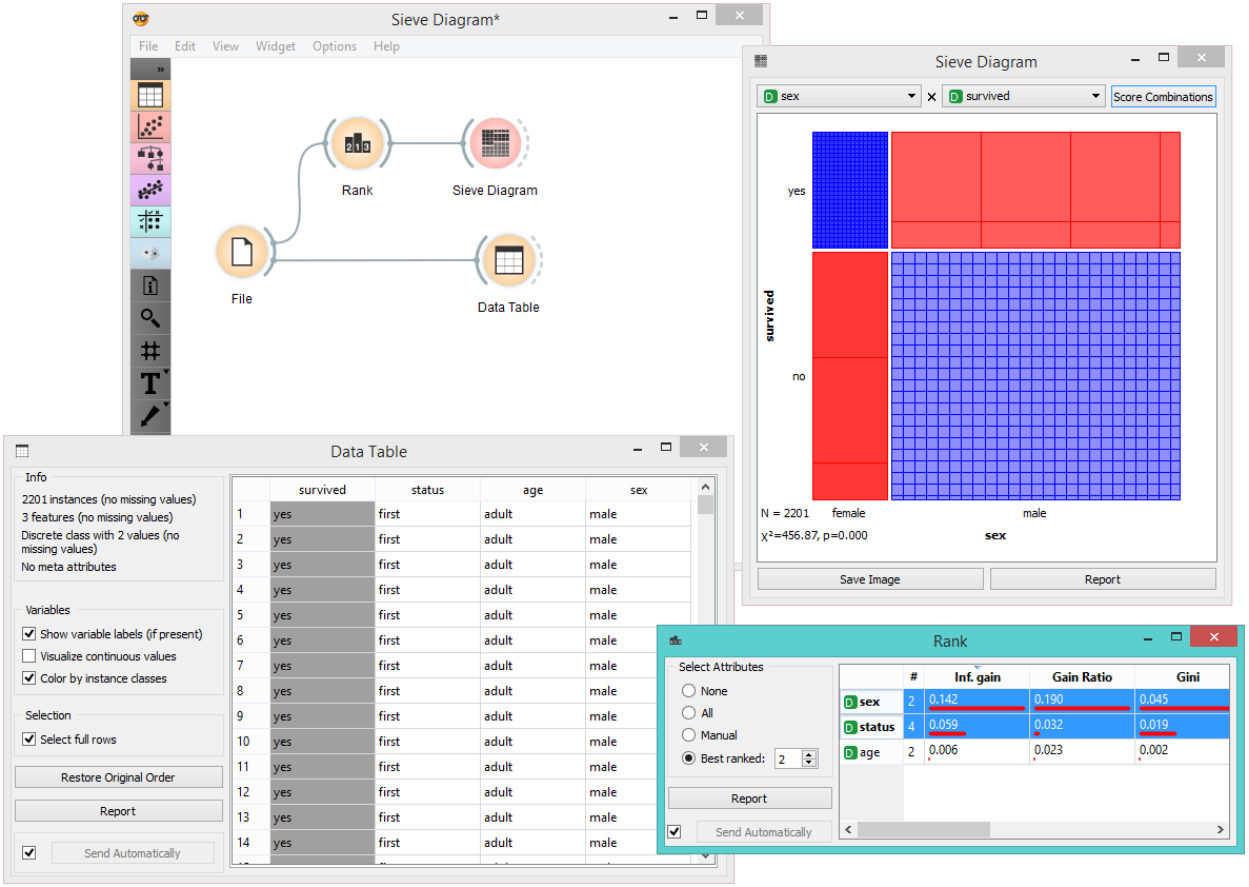


Pairs of attributes with interesting associations have a strong shading, such as the diagram shown in the above snapshot. For contrast, a sieve diagram of the least interesting pair (age vs. survival) is shown below.

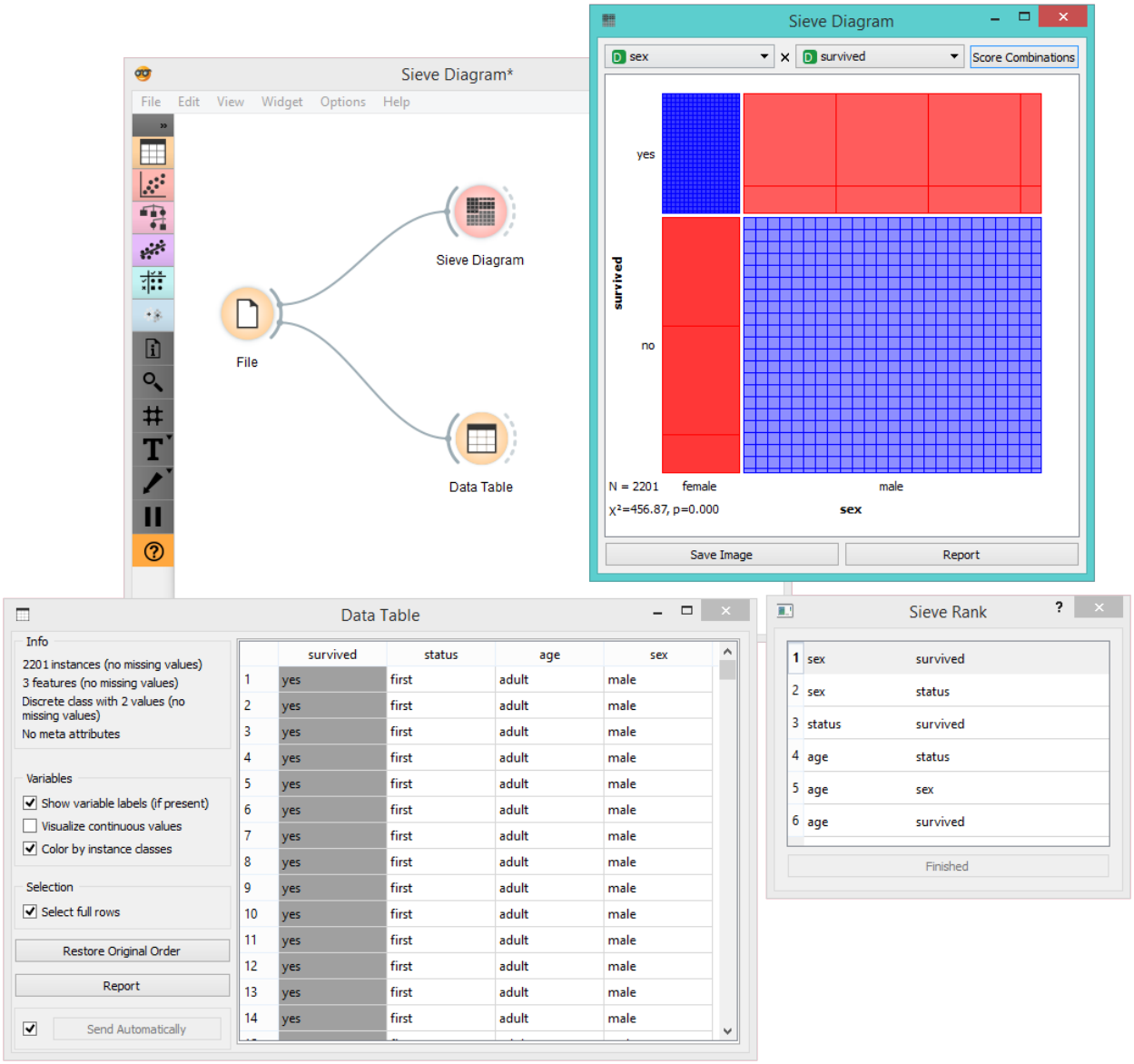

Sieve Diagram
age
survived
Score Combinations
yes
no
survived
N = 2201
adult
child
χ²=20.96, p=0.000
age
Save Image
Report

Example

Below, we see a simple schema using the *Titanic* dataset, 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.

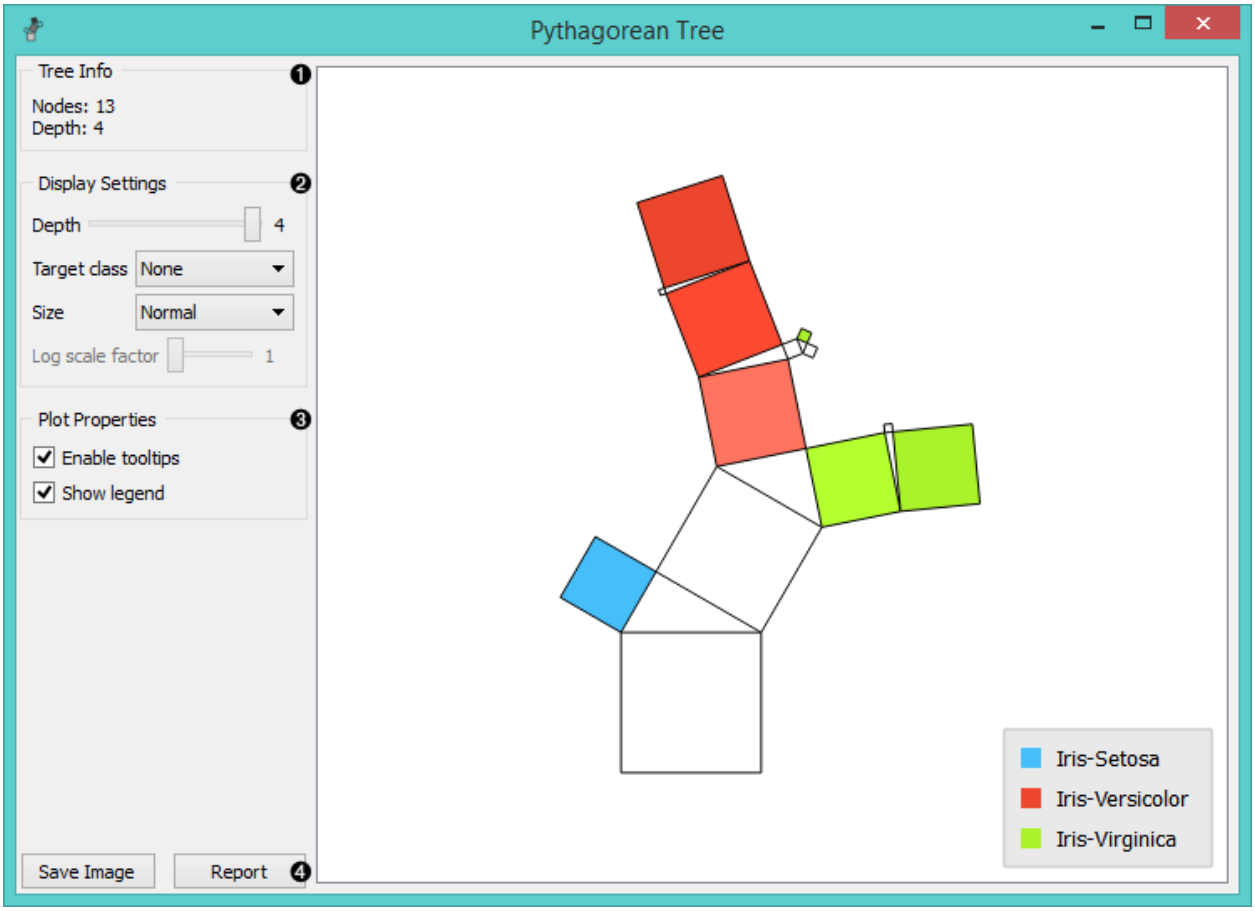
2.2.9 Pythagorean Tree

Pythagorean tree visualization for classification or regression trees.

Inputs

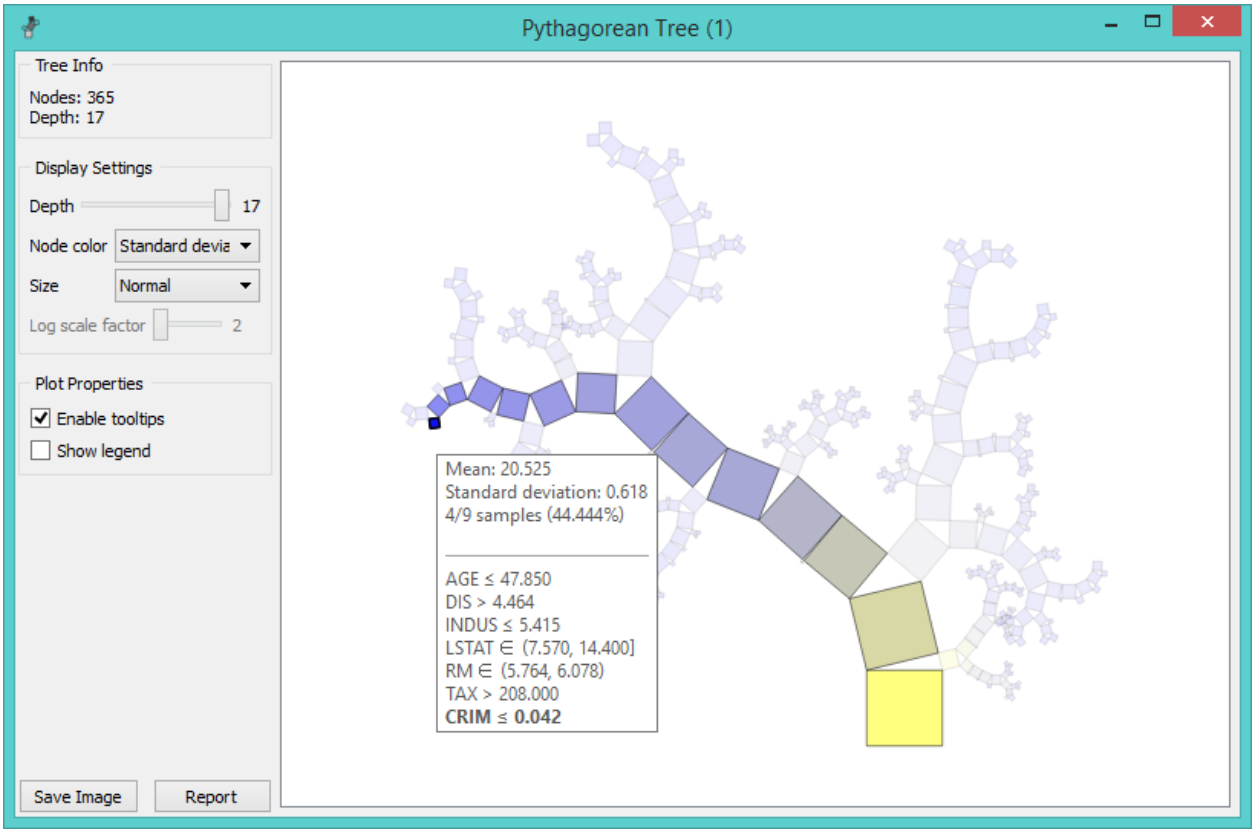
- Tree: tree model
- Selected Data: instances selected from the tree

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 Tree.



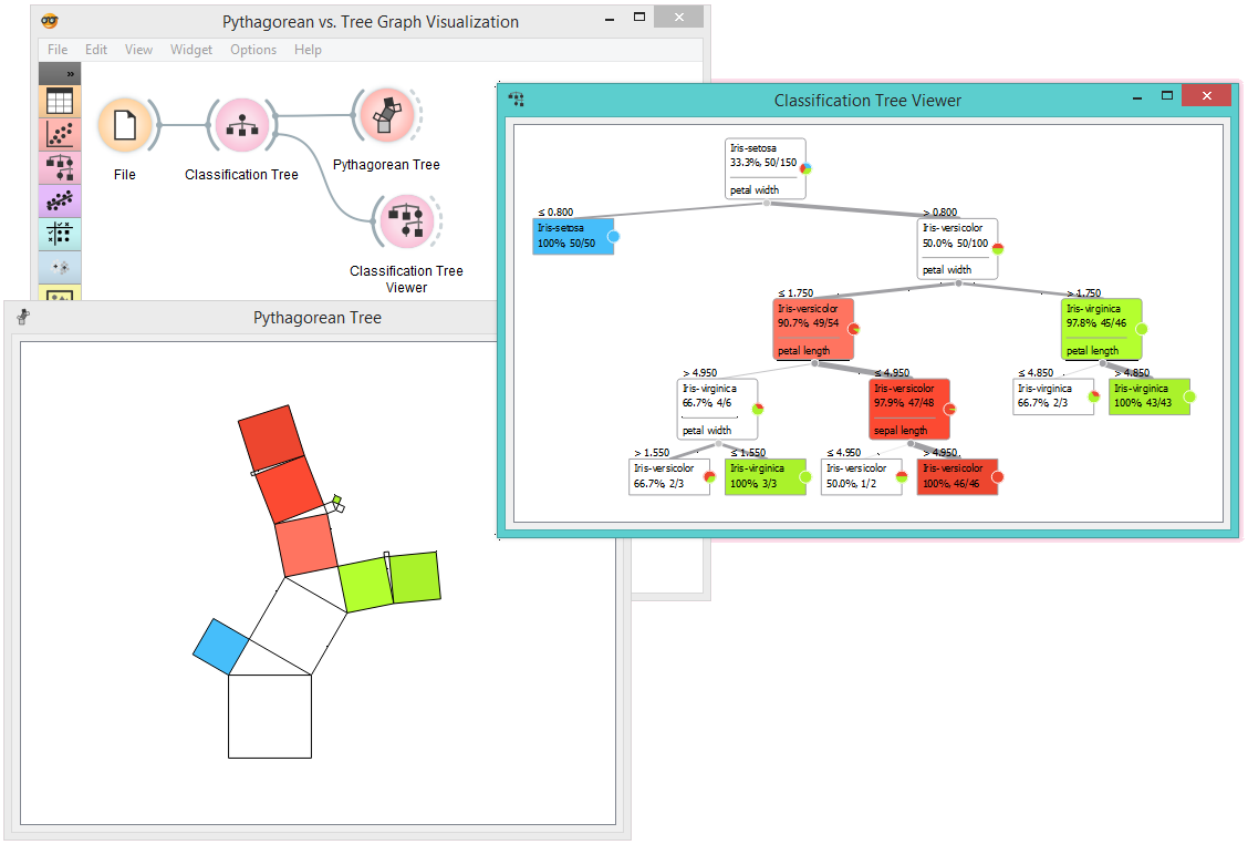
1. Information on the input tree model.
2. Visualization parameters:
    - *Depth*: set the depth of displayed trees.
    - *Target class* (for classification trees): the intensity of the color for nodes of the tree will correspond to the probability of the target class. If *None* is selected, the color 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.

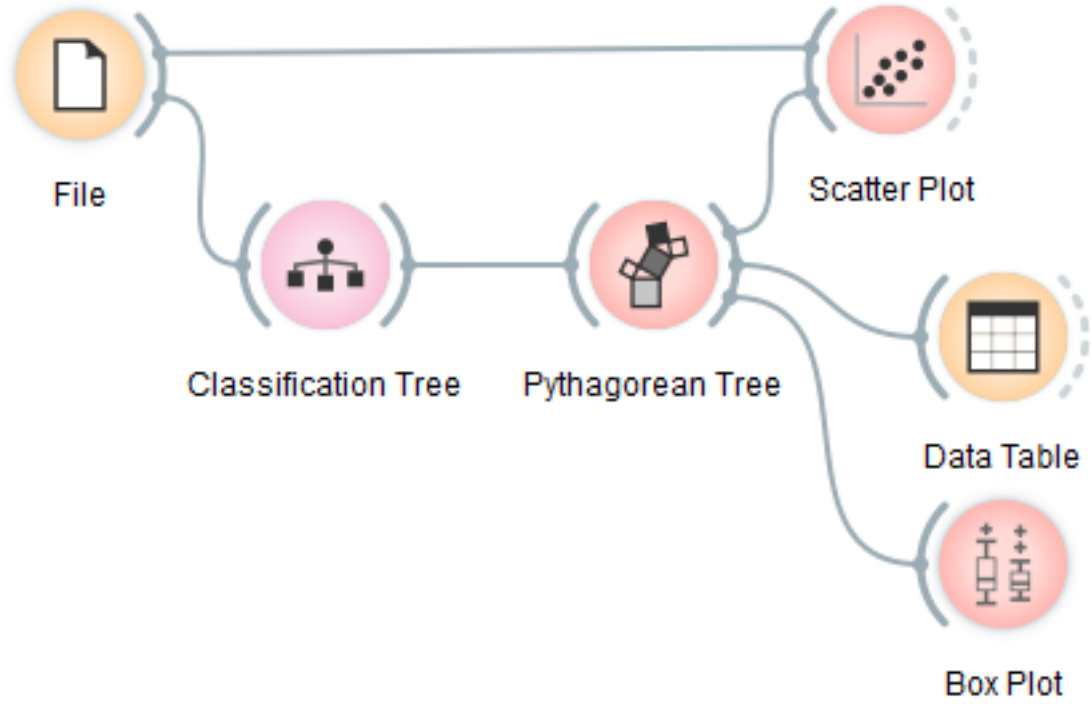


Example

The workflow from the screenshot below demonstrates the difference between Tree Viewer and Pythagorean Tree. They can both visualize Tree, but Pythagorean visualization takes less space and is more compact, even for a small Iris flower dataset. 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 dataset 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.

2.2.10 Pythagorean Forest

Pythagorean forest for visualizing random forests.

Inputs

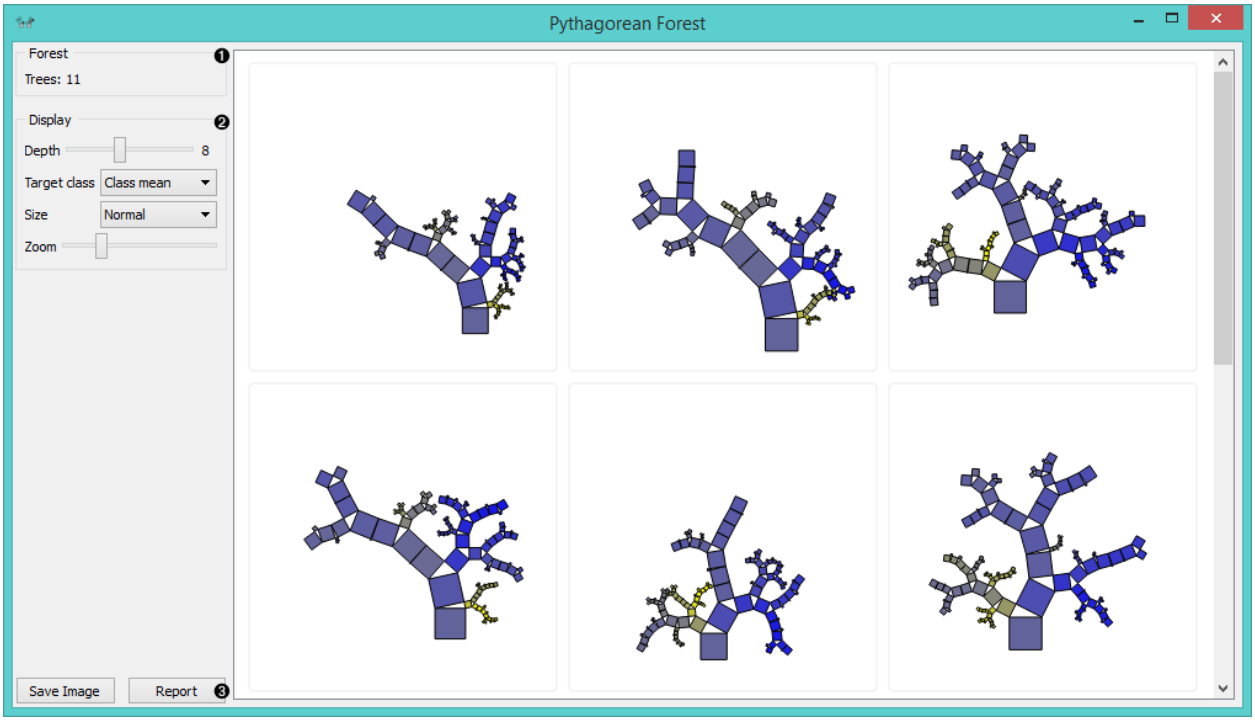
- Random Forest: tree models from random forest

Outputs

- Tree: selected tree model

Pythagorean Forest shows all learned decision tree models from Random Forest widget. It displays them 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 dataset, while regression requires a continuous target variable. Still, they both should be fed a Tree on the input.

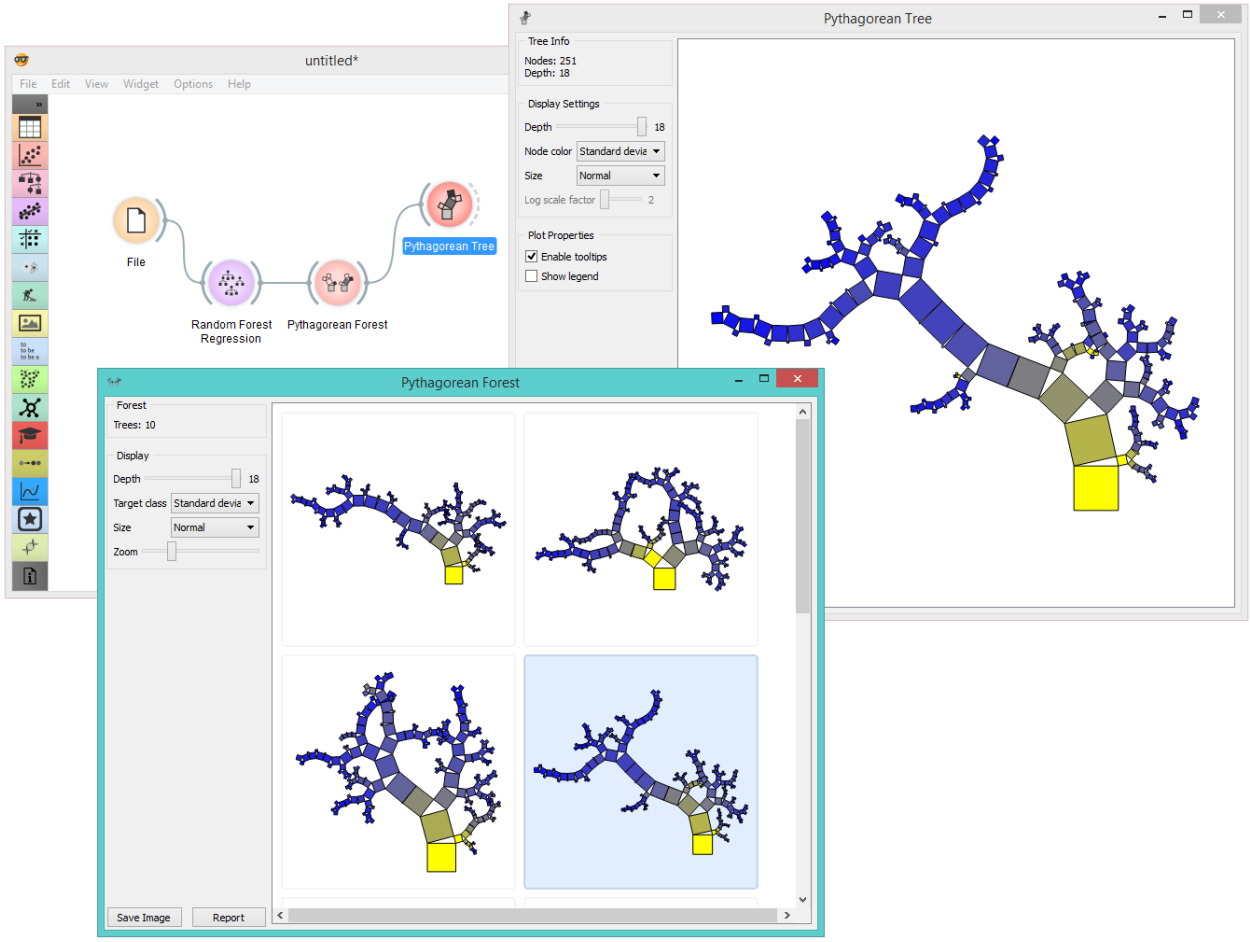


1. Information on the input random forest model.
2. Display parameters:
   - *Depth*: set the depth to which the trees are grown.
   - *Target class*: set the target class for coloring the trees. If *None* is selected, the tree will be white. If the input is a classification tree, you can color the nodes 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 them by the standard deviation value of the node.
   - *Size*: set the size of the nodes. *Normal* will keep the 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 see 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 used *housing* dataset and plotted all 10 trees we've grown with Random Forest. When changing the parameters in Random Forest, 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.

2.2.11 CN2 Rule Viewer

CN2 Rule Viewer

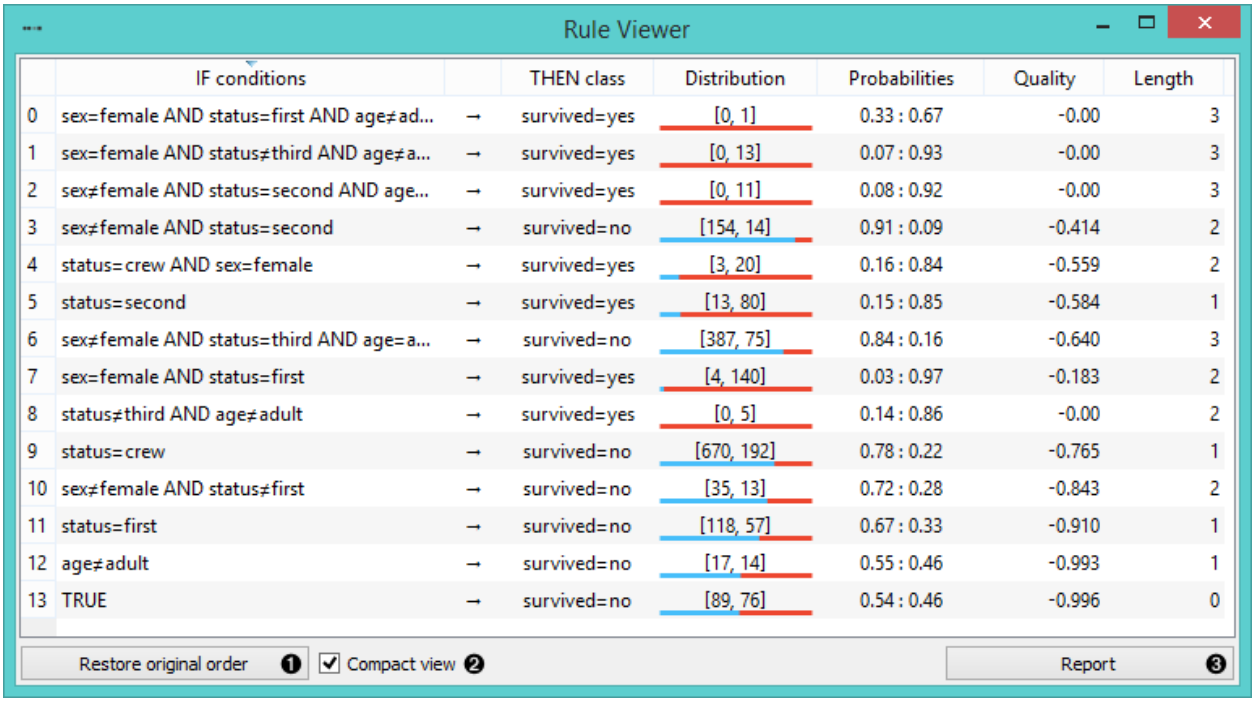
Inputs

- Data: dataset to filter
- CN2 Rule Classifier: CN2 Rule Classifier, including a list of induced rules

Outputs

- Filtered Data: data instances covered by all selected rules

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

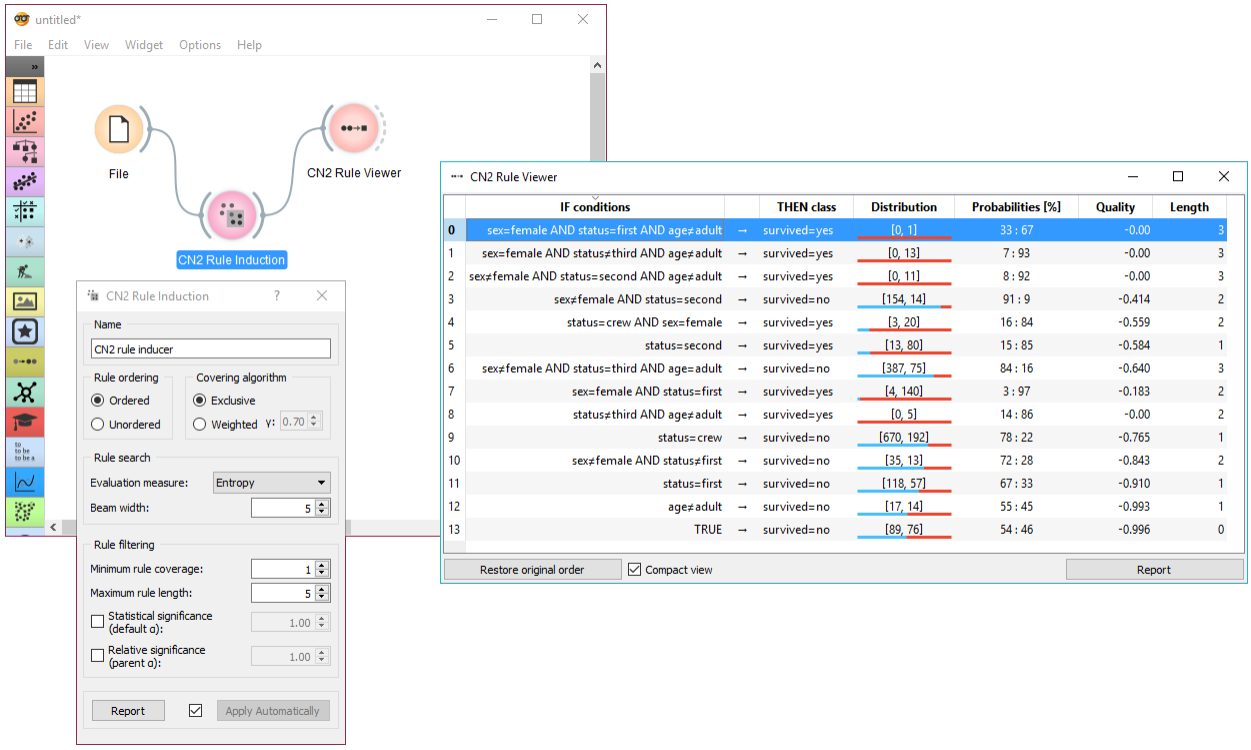


1. Original order of induced rules can be restored.
2. When rules are many and complex, the view can appear packed. For this reason, *compact view* was implemented, which allows a flat presentation and a cleaner inspection of rules.
3. Click *Report* to bring up a detailed description of the rule induction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

2.2.12 Mosaic Display

Display data in a mosaic plot.

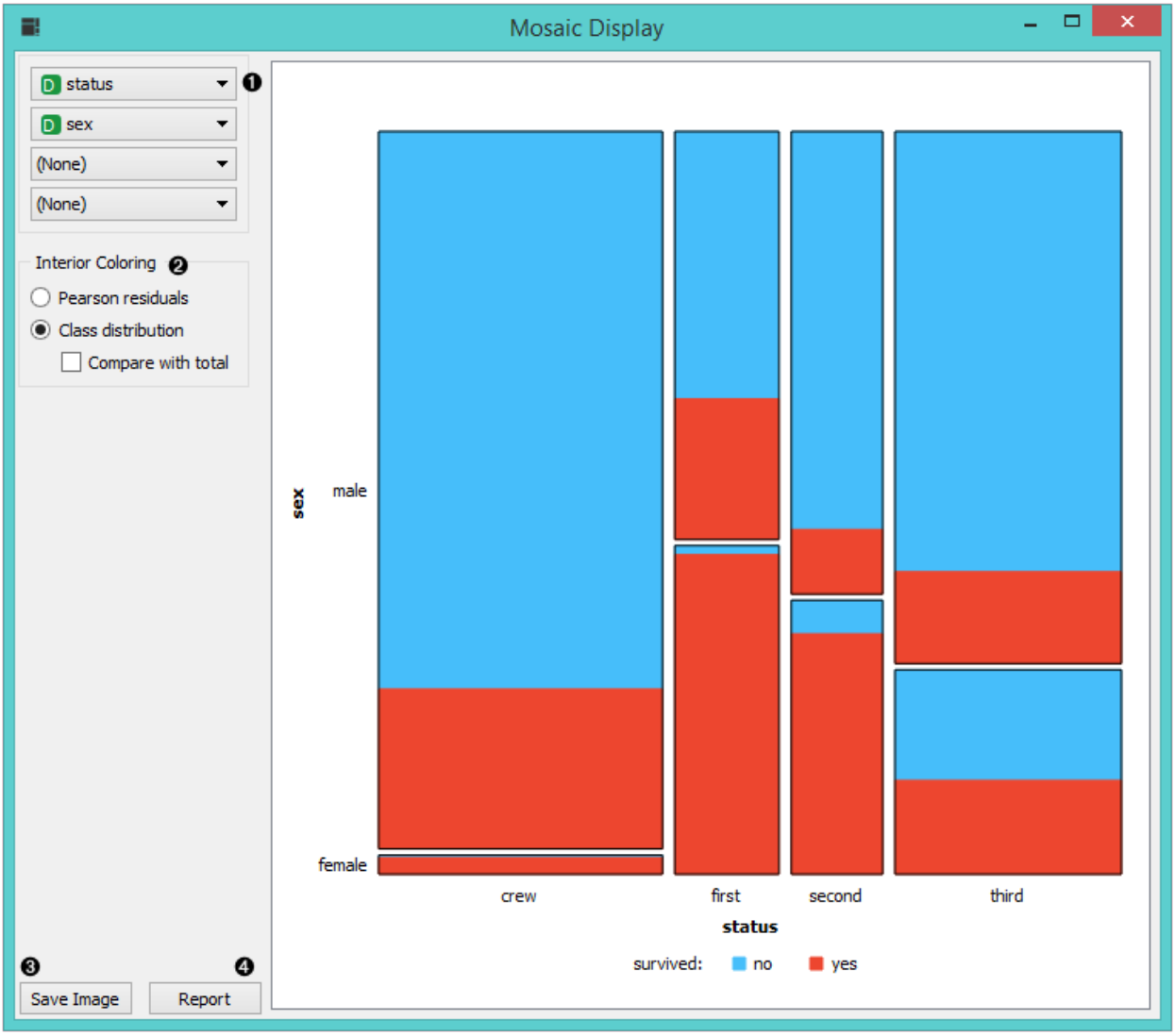
Inputs

- Data: input dataset
- Data subset: subset of instances

Outputs

- Selected data: instances selected from the plot

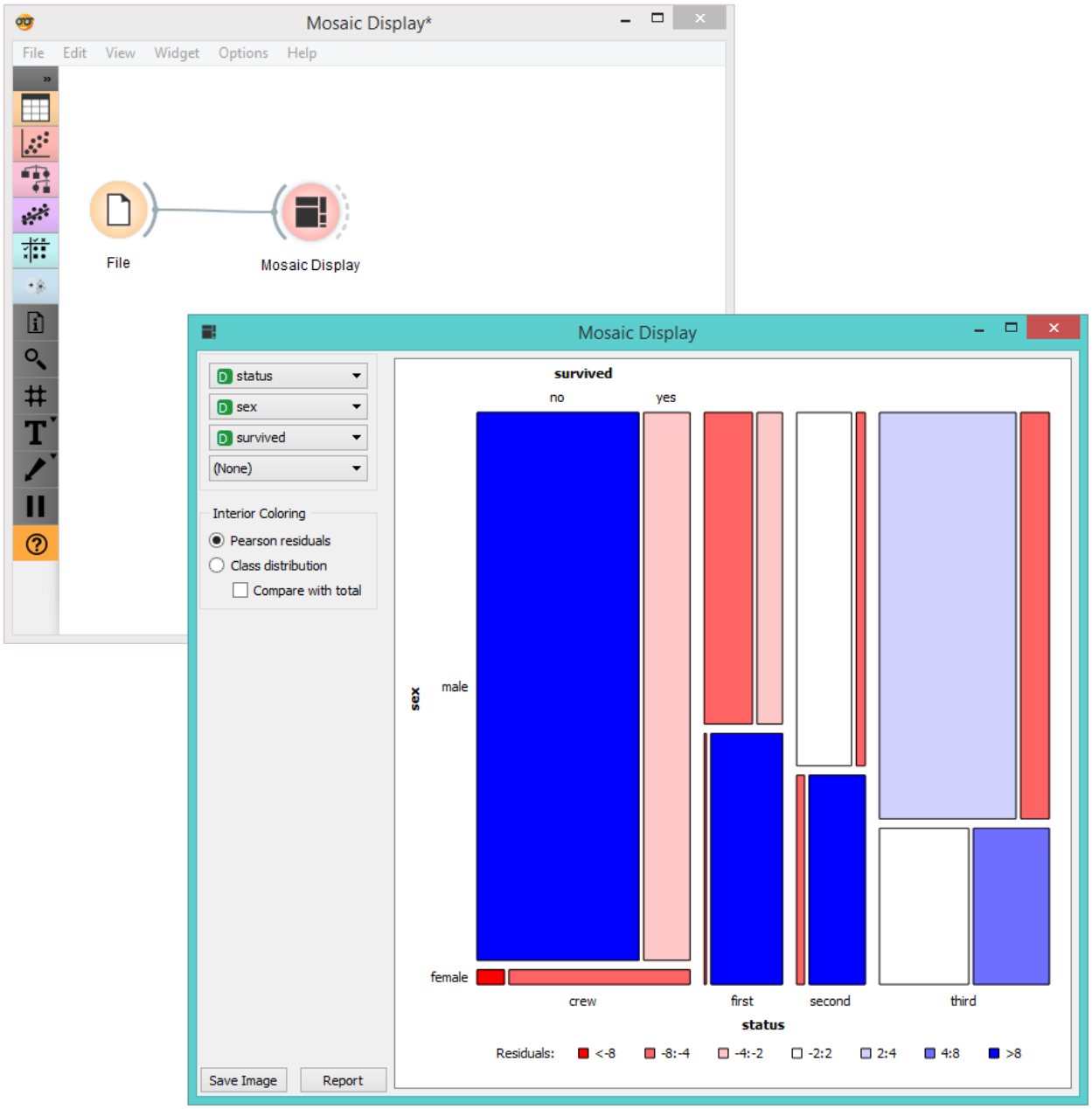
The **Mosaic plot** is a graphical representation of a two-way frequency table or a contingency table. It is used for visualizing data from two or more qualitative variables and was introduced in 1981 by Hartigan and Kleiner and expanded and refined by Friendly in 1994. It provides the user with the 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* dataset 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.

2.2.13 Silhouette Plot

A graphical representation of consistency within clusters of data.

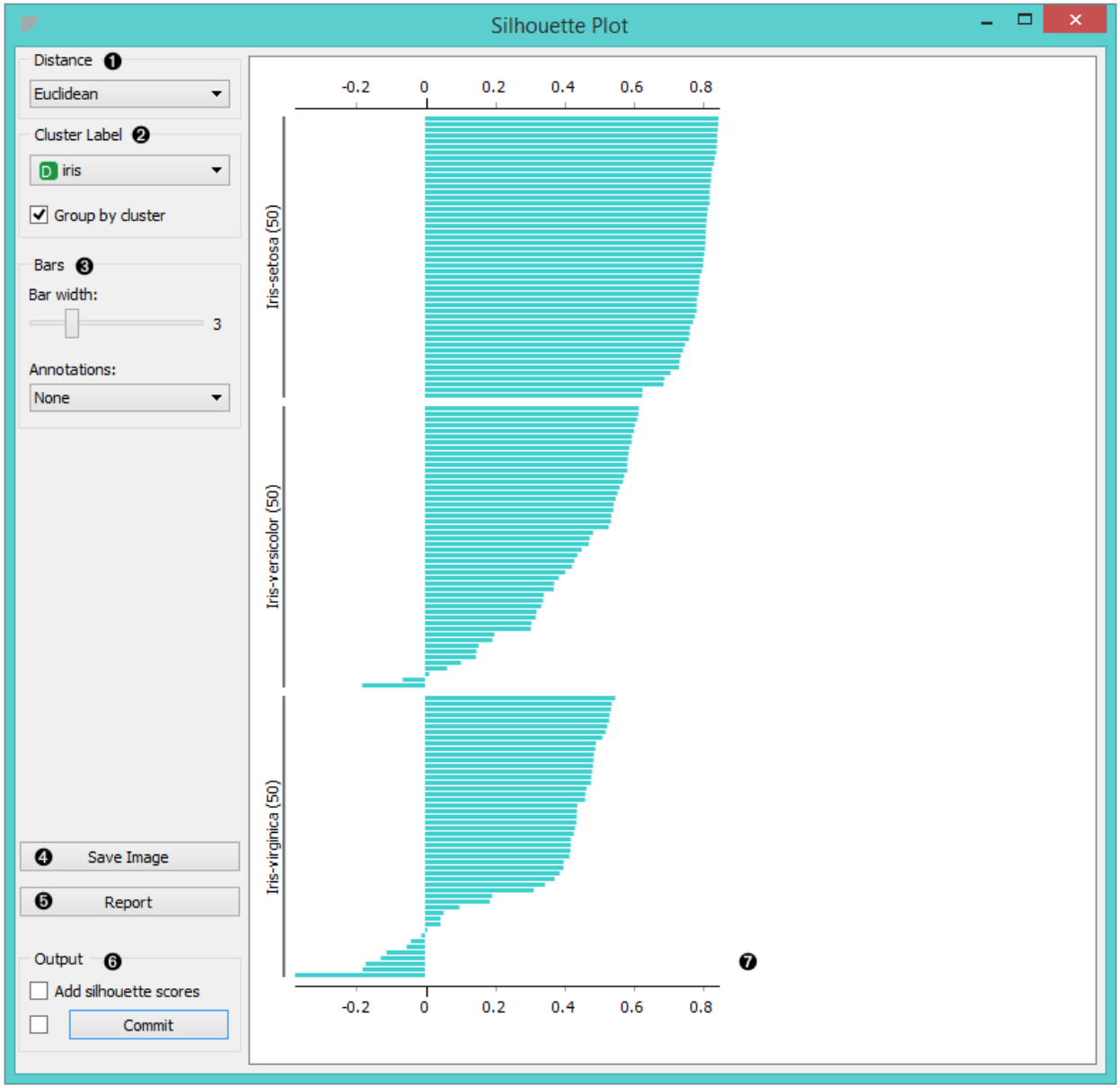
Inputs

- Data: input dataset

Outputs

- Selected Data: instances selected from the plot
- Data: data with an additional column showing whether a point is selected

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.



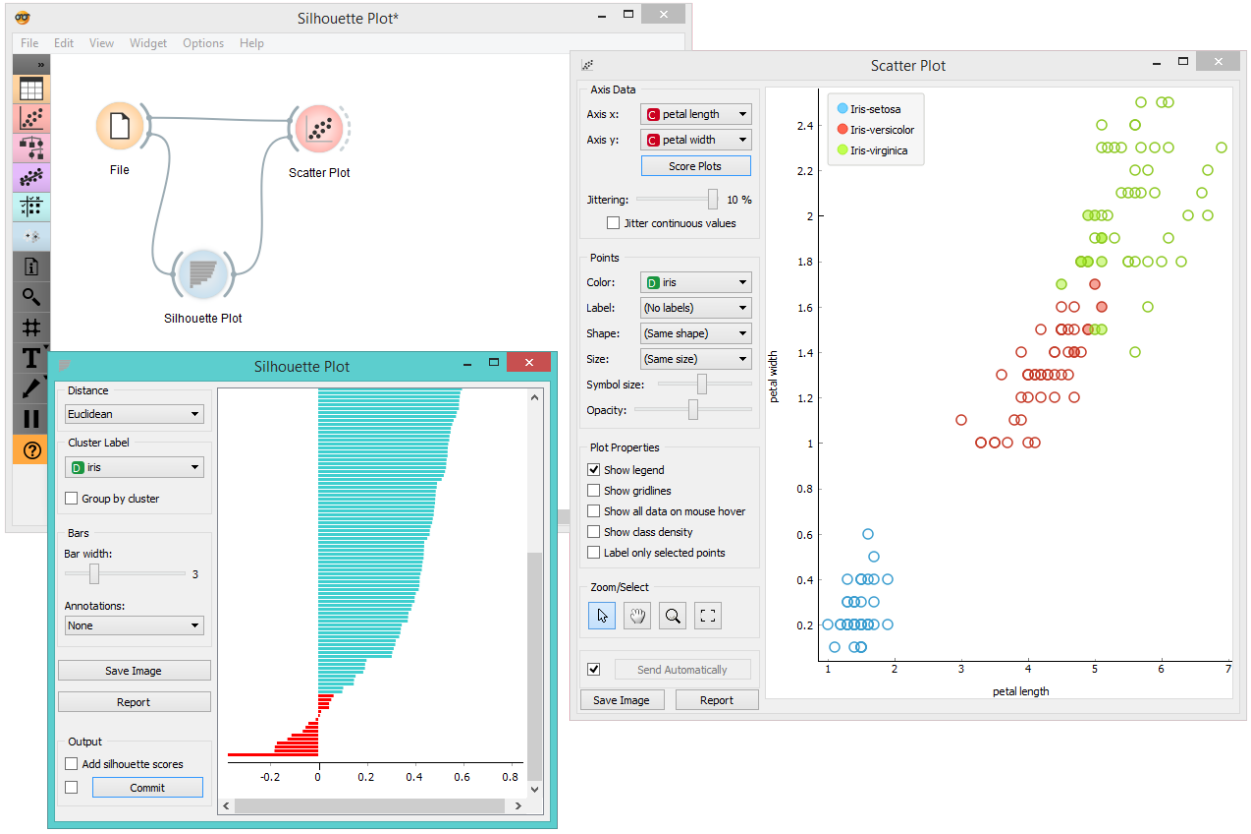
1. Choose the distance metric. You can choose between:
   - Euclidean ("straight line" distance between two points)
   - Manhattan (the sum of absolute differences for all attributes)
   - Cosine (1 - cosine of the angle between two vectors)
2. Select the cluster label. You can decide whether to group 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* dataset. 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.

2.2.14 Tree Viewer

A visualization of classification and regression trees.

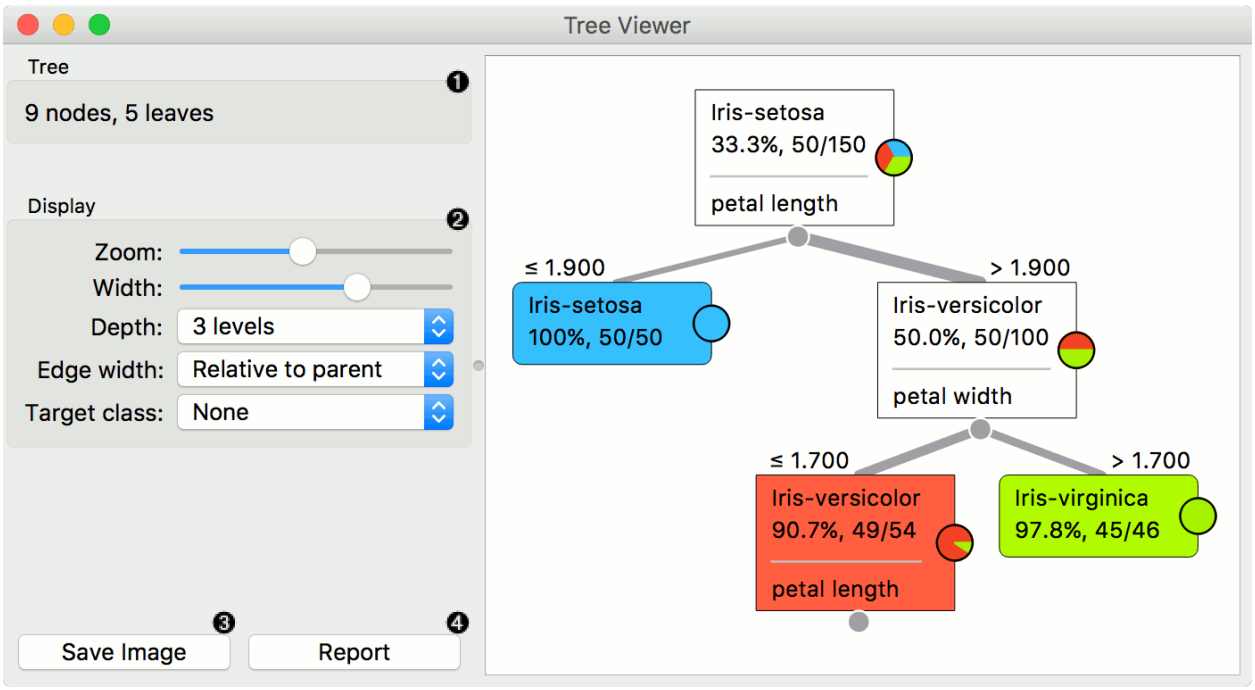
Inputs

- Tree: decision tree

Outputs

- Selected Data: instances selected from the tree node
- Data: data with an additional column showing whether a point is selected

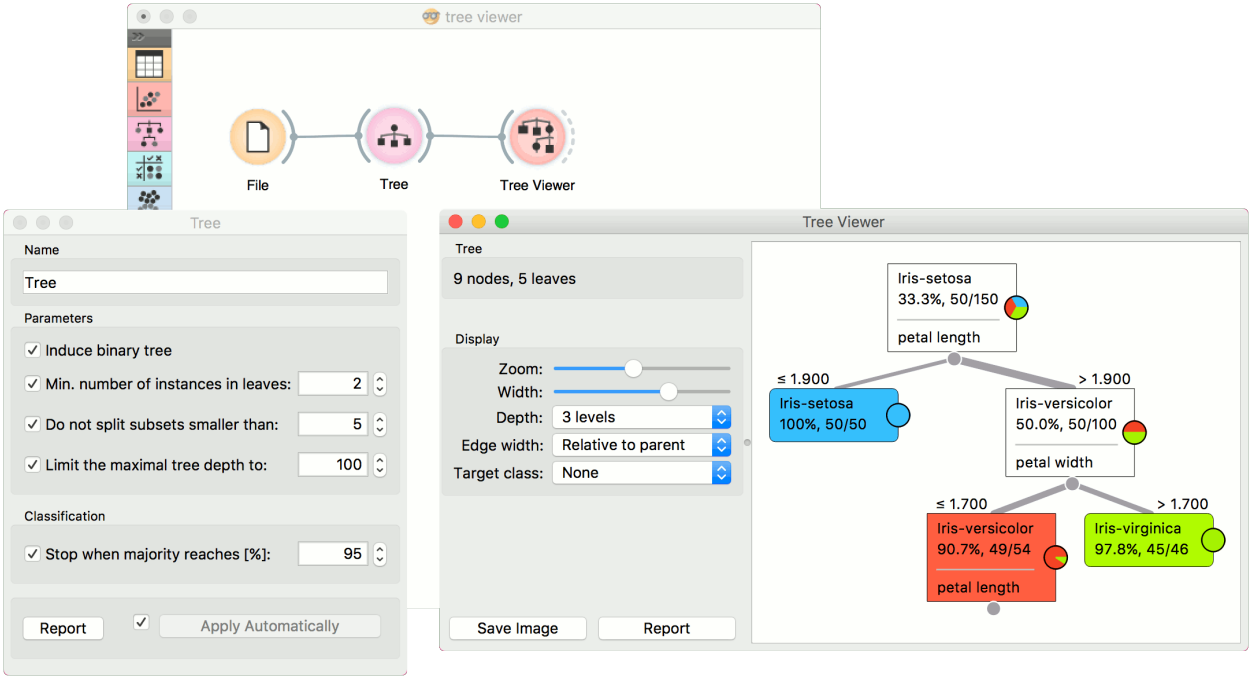
This is a versatile widget with 2-D visualization of classification and regression trees. The user can select a node, instructing the widget to output the data associated with the node, thus enabling explorative data analysis.



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 tree graph to your computer as a *.svg* or *.png* file.
4. Produce a report.

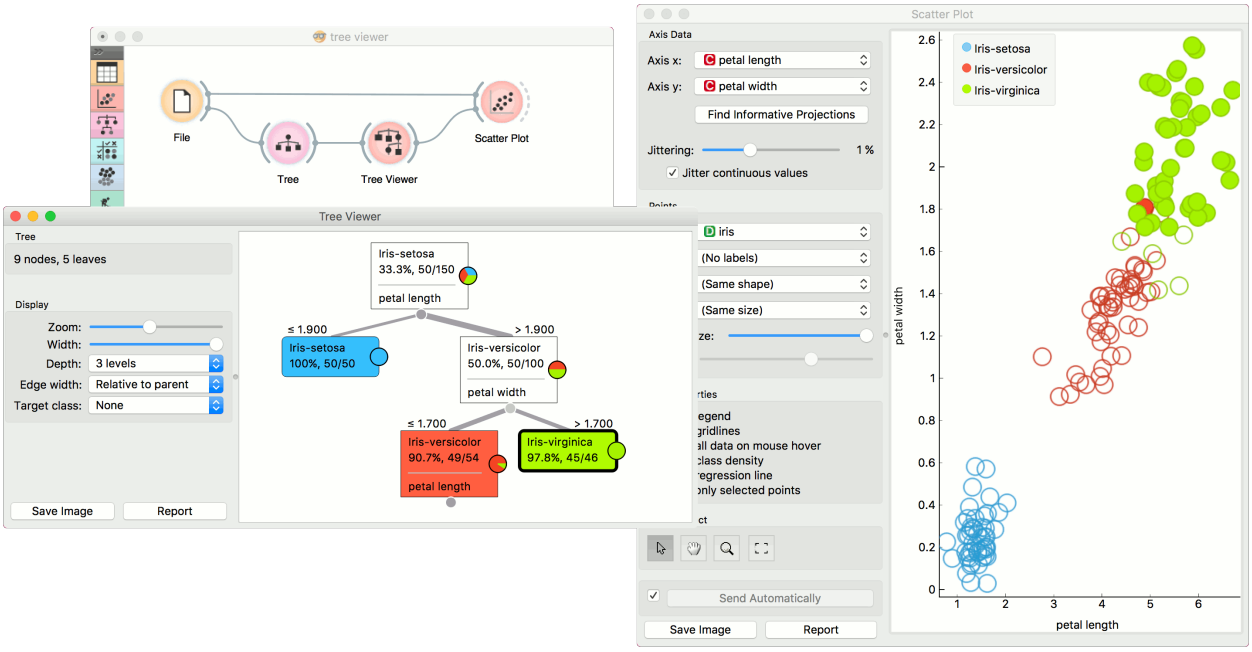
Examples

Below, is a simple classification schema, where we have read the data, constructed the decision tree and viewed it in our **Tree Viewer**. If both the viewer and 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.

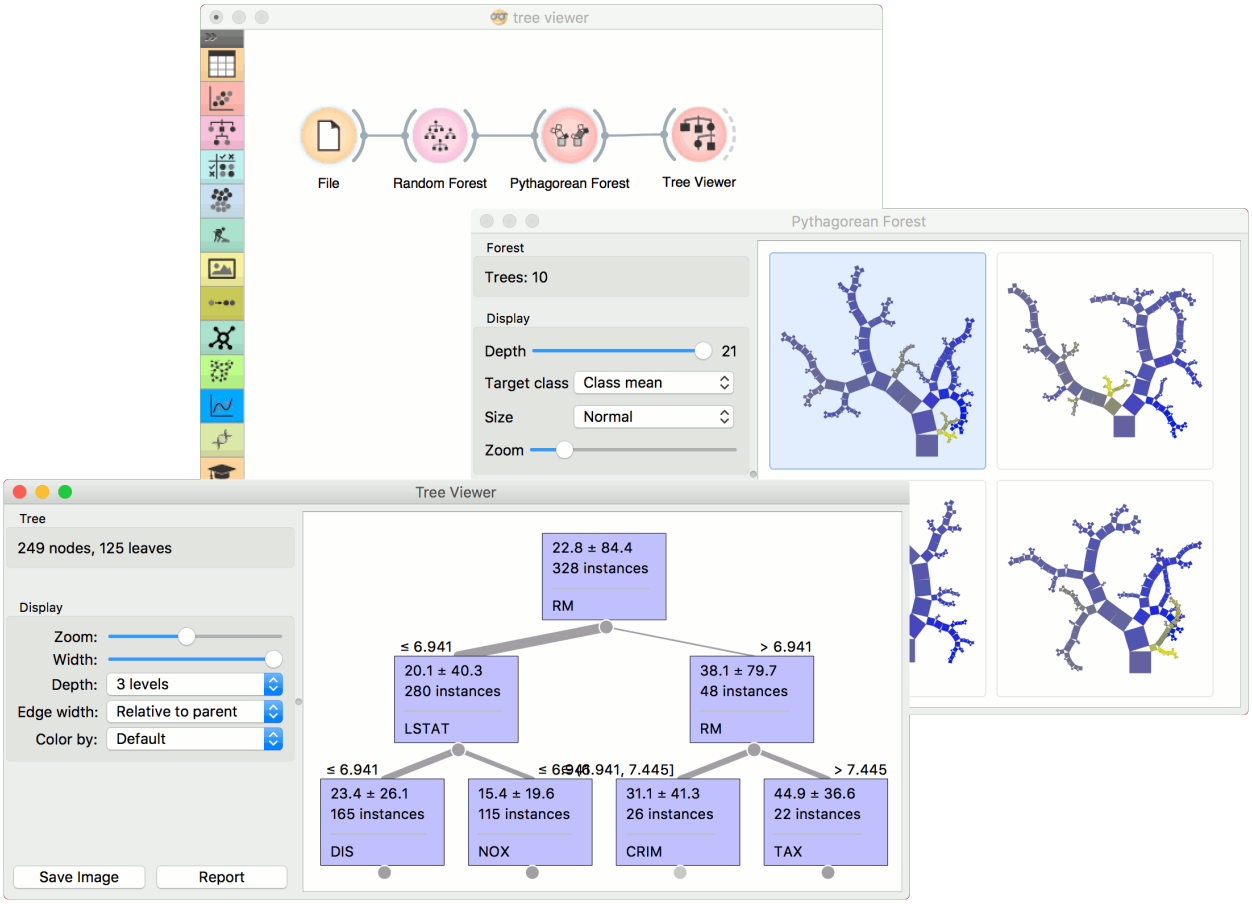


Clicking on any node will output the related data instances. This is explored in the schema below that shows the subset in the data table and in the Scatter Plot. Make sure that the tree data is passed as a data subset; this can be done by connecting the **Scatter Plot** to the File widget first, and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

Tree Viewer can also export labeled 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 labeling selected data instances (*Yes* for selected and *No* for the remaining).



Finally, **Tree Viewer** can be used also for visualizing regression trees. Connect Random Forest to File widget using *housing.tab* dataset. Then connect Pythagorean Forest to **Random Forest**. In **Pythagorean Forest** select a regression tree you wish to further analyze and pass it to the **Tree Viewer**. The widget will display the constructed tree. For visualizing larger trees, especially for regression, Pythagorean Tree could be a better option.



2.2.15 Nomogram

Nomograms for visualization of Naive Bayes and Logistic Regression classifiers.

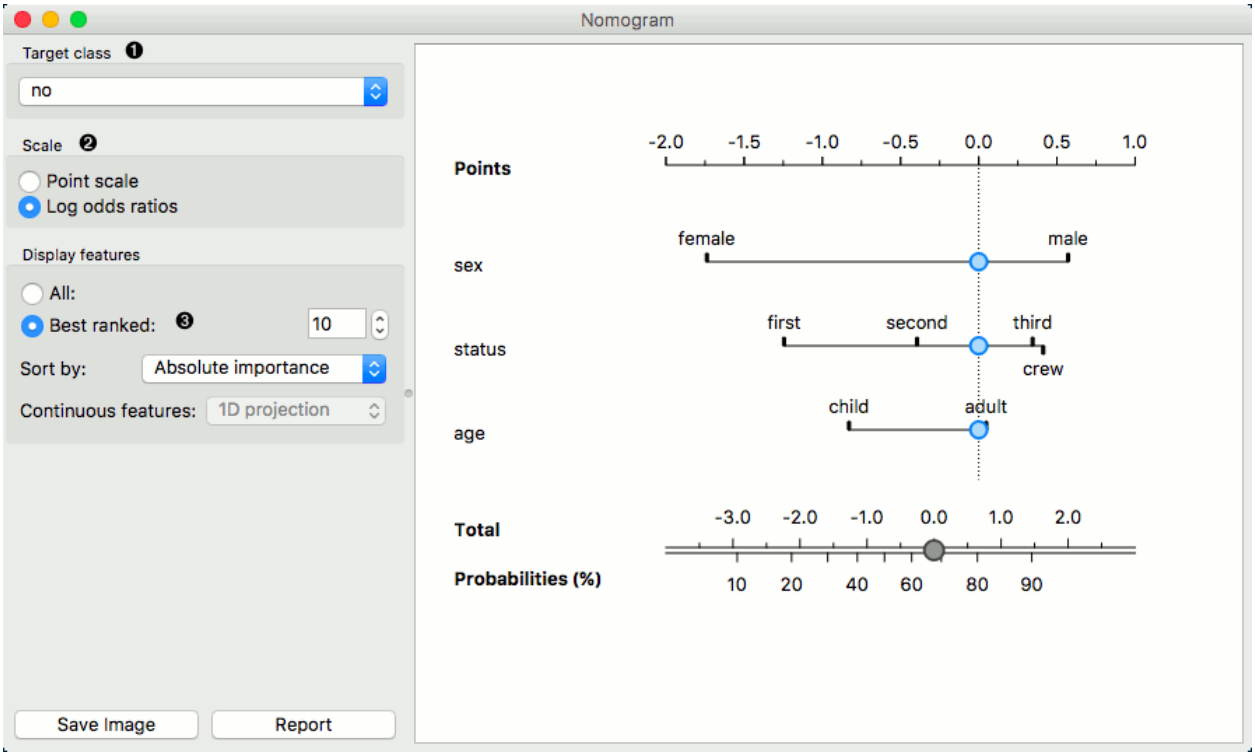
Inputs

- Classifier: trained classifier
- Data: input dataset

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 for prediction of class probabilities. A snapshot below shows the nomogram of the Titanic dataset, that models the probability for a passenger not to survive the disaster of the Titanic.

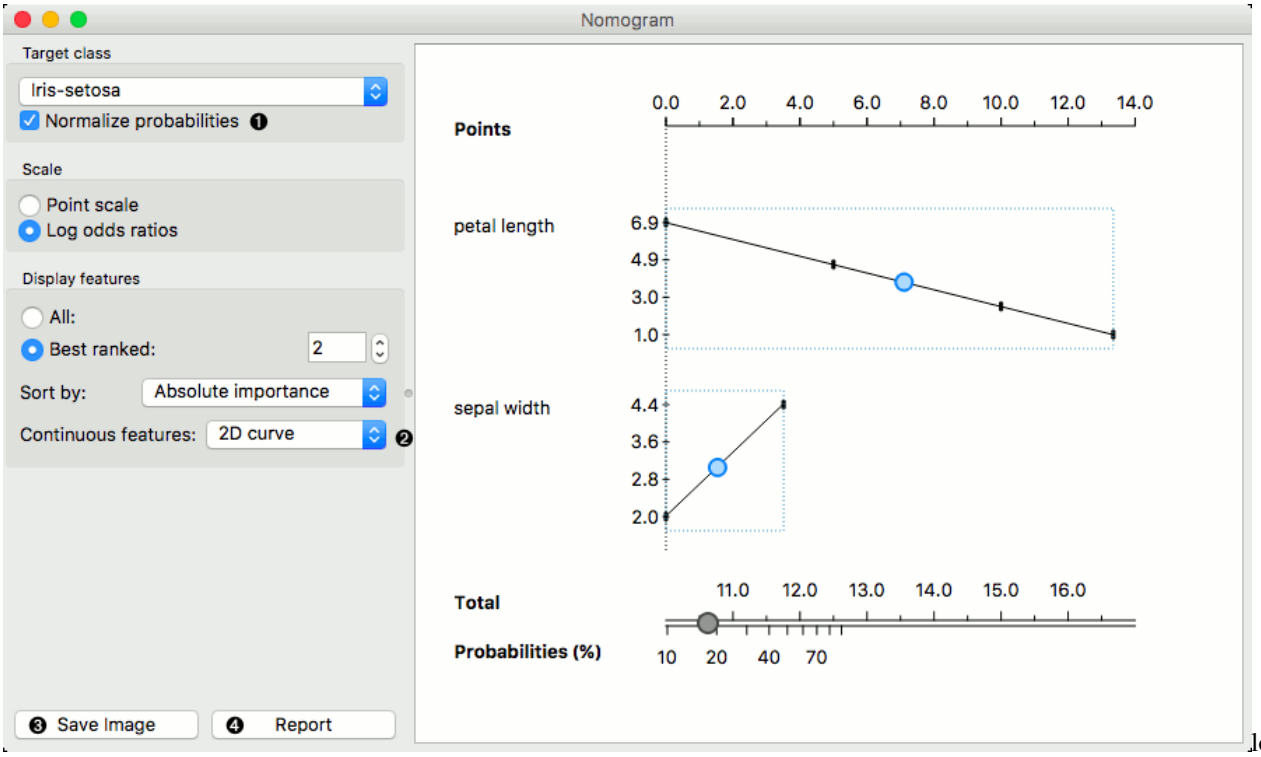
When there are too many attributes in the plotted dataset, you can choose to display only best ranked ones. It is possible to choose from 'No sorting', 'Name', 'Absolute importance', 'Positive influence' and 'Negative influence' for Naive Bayes representation and from 'No sorting', 'Name' and 'Absolute importance' for Logistic Regression representation.

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.



1. Select the target class you want to model the probability for. Select, whether you want to normalize the probabilities or not.
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. Display all attributes or only the best ranked ones. Sort them and set the projection type.

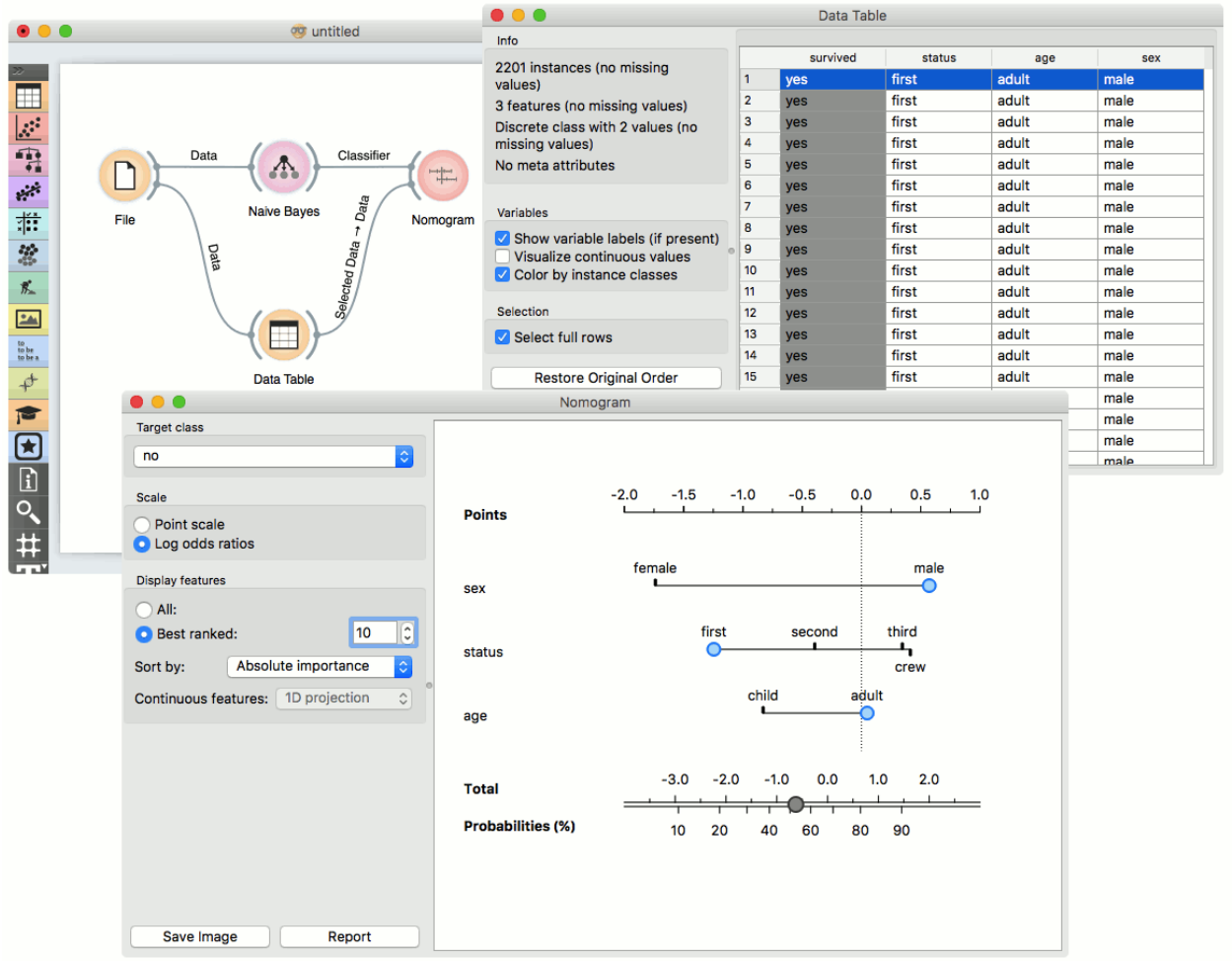
Continuous attributes can be plotted in 2D (only for Logistic Regression).



logreg

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 dataset once again, 1490 (68%) passengers on Titanic out of 2201 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 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%.

2.2.16 FreeViz

Displays FreeViz projection.

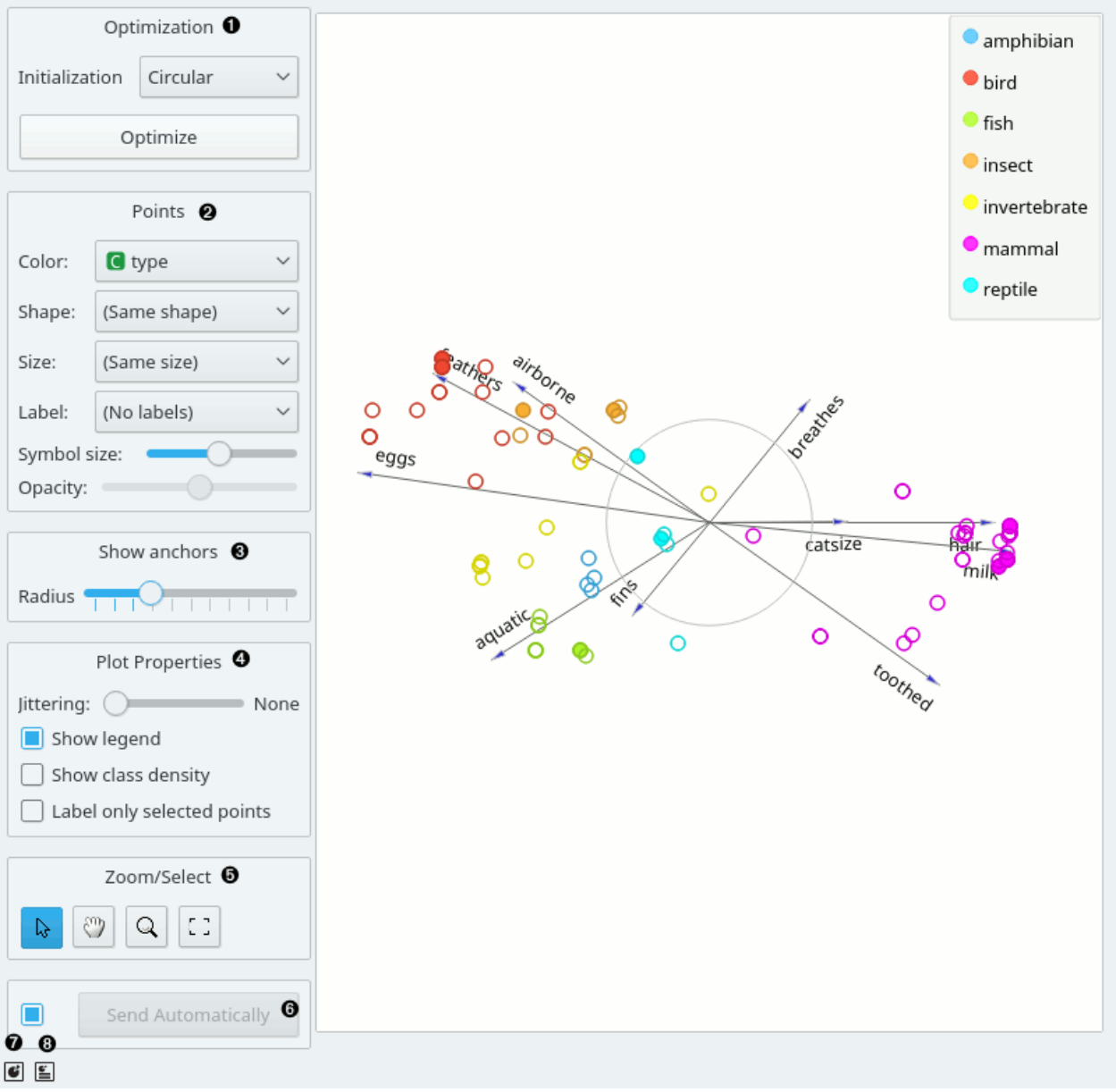
Inputs

- Data: input dataset
- Data Subset: subset of instances

Outputs

- Selected Data: instances selected from the plot
- Data: data with an additional column showing whether a point is selected
- Components: FreeViz vectors

FreeViz uses a paradigm borrowed from particle physics: points in the same class attract each other, those from different class repel each other, and the resulting forces are exerted on the anchors of the attributes, that is, on unit vectors of each of the dimensional axis. The points cannot move (are projected in the projection space), but the attribute anchors can, so the optimization process is a hill-climbing optimization where at the end the anchors are placed such that forces are in equilibrium. The button Optimize is used to invoke the optimization process. The result of the optimization may depend on the initial placement of the anchors, which can be set in a circle, arbitrary or even manually. The later also works at any stage of optimization, and we recommend to play with this option in order to understand how a change of one anchor affects the positions of the data points. In any linear projection, projections of unit vector that are very short compared to the others indicate that their associated attribute is not very informative for particular classification task. Those vectors, that is, their corresponding anchors, may be hidden from the visualization using Radius slider in Show anchors box.

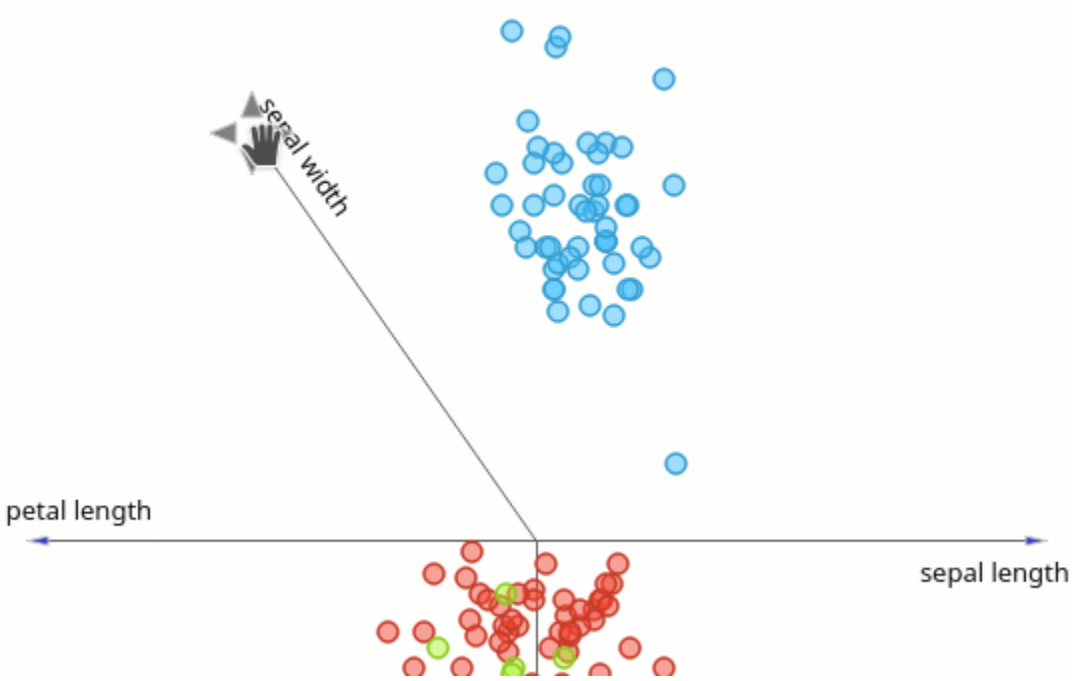


1. Two initial positions of anchors are possible: random and circular. Optimization moves anchors in an optimal position.
2. Set the color of the displayed points (you will get colors for discrete values and grey-scale points for continuous).

Set label, shape and size to differentiate between points. Set symbol size and opacity for all data points.

3. Anchors inside a circle are hidden. Circle radius can be be changed using a slider.
4. Adjust plot properties:
   - Set jittering to prevent the dots from overlapping (especially for discrete attributes).
   - *Show legend* displays a legend on the right. Click and drag the legend to move it.
   - *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.
5. *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.
6. If *Send automatically* is ticked, changes are communicated automatically. Alternatively, press *Send*.
7. *Save Image* saves the created image to your computer in a .svg or .png format.
8. Produce a report.

Manually move anchors

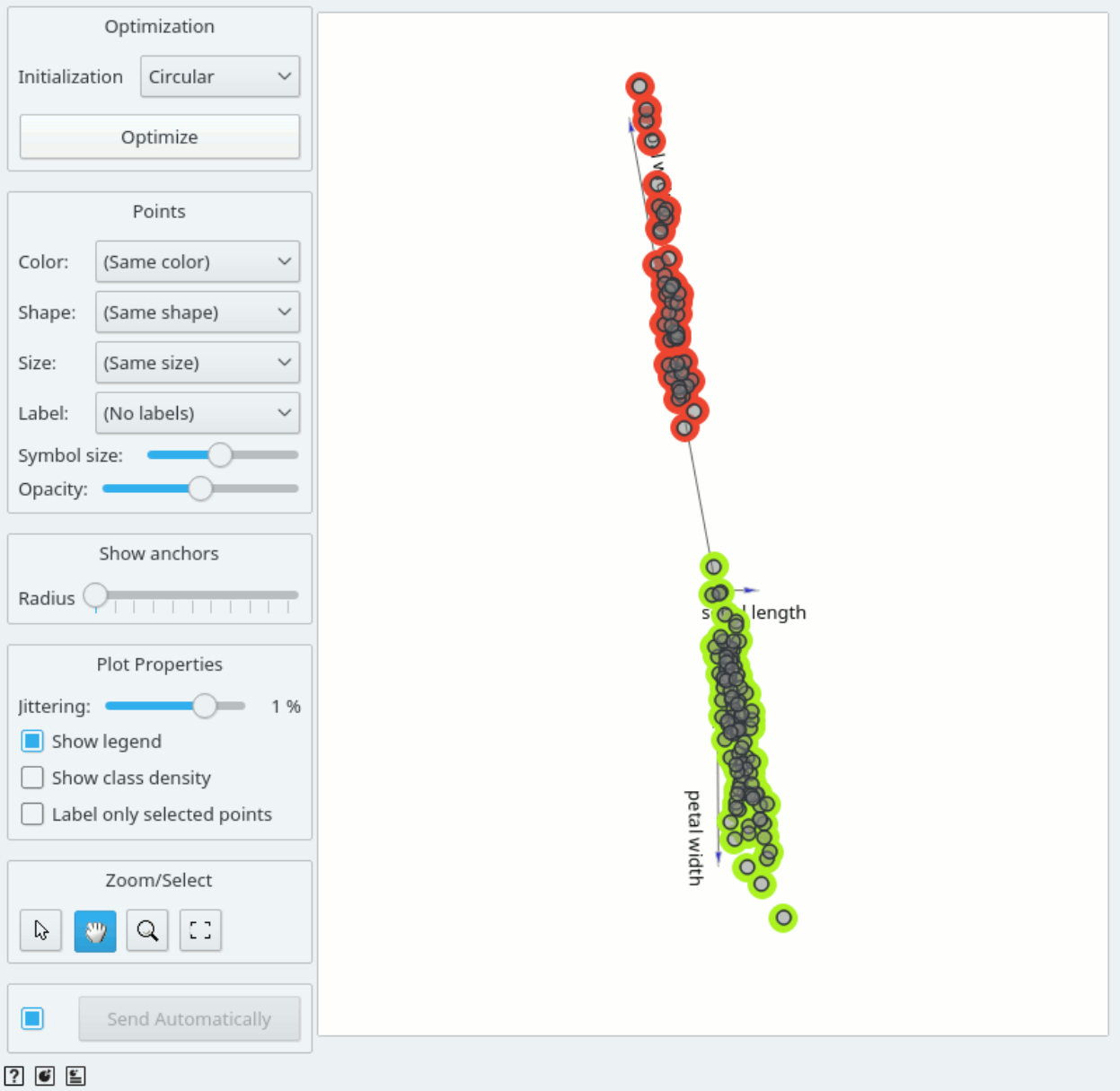


One can manually move anchors. Use a mouse pointer and hover above the end of an anchor. Click the left button and then you can move selected anchor where ever you want.

Selection

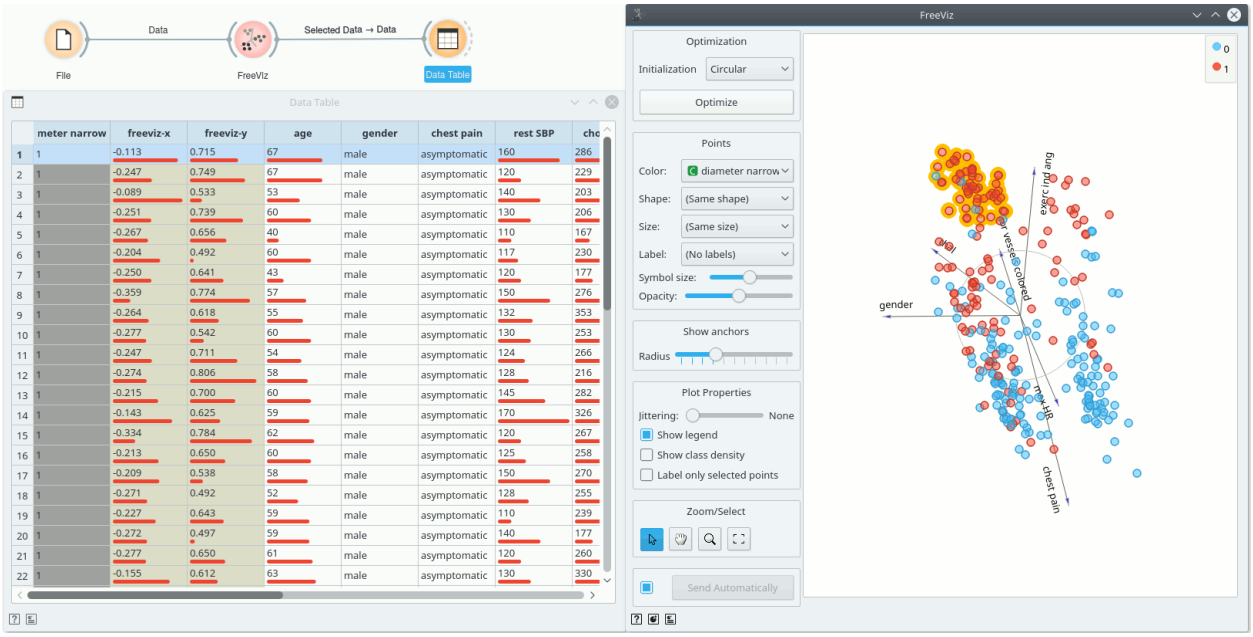
Selection can be used to manually defined subgroups in the data. Use Shift modifier when selecting data instances to put them into a new group. Shift + Ctrl (or Shift + Cmd on macOs) appends instances to the last group.

Signal data outputs a data table with an additional column that contains group indices.



Explorative Data Analysis

The **FreeViz**, 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 plot 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.



2.2.17 Radviz

Radviz vizualization with explorative data analysis and intelligent data visualization enhancements.

Inputs

- Data: input dataset
- Data Subset: subset of instances

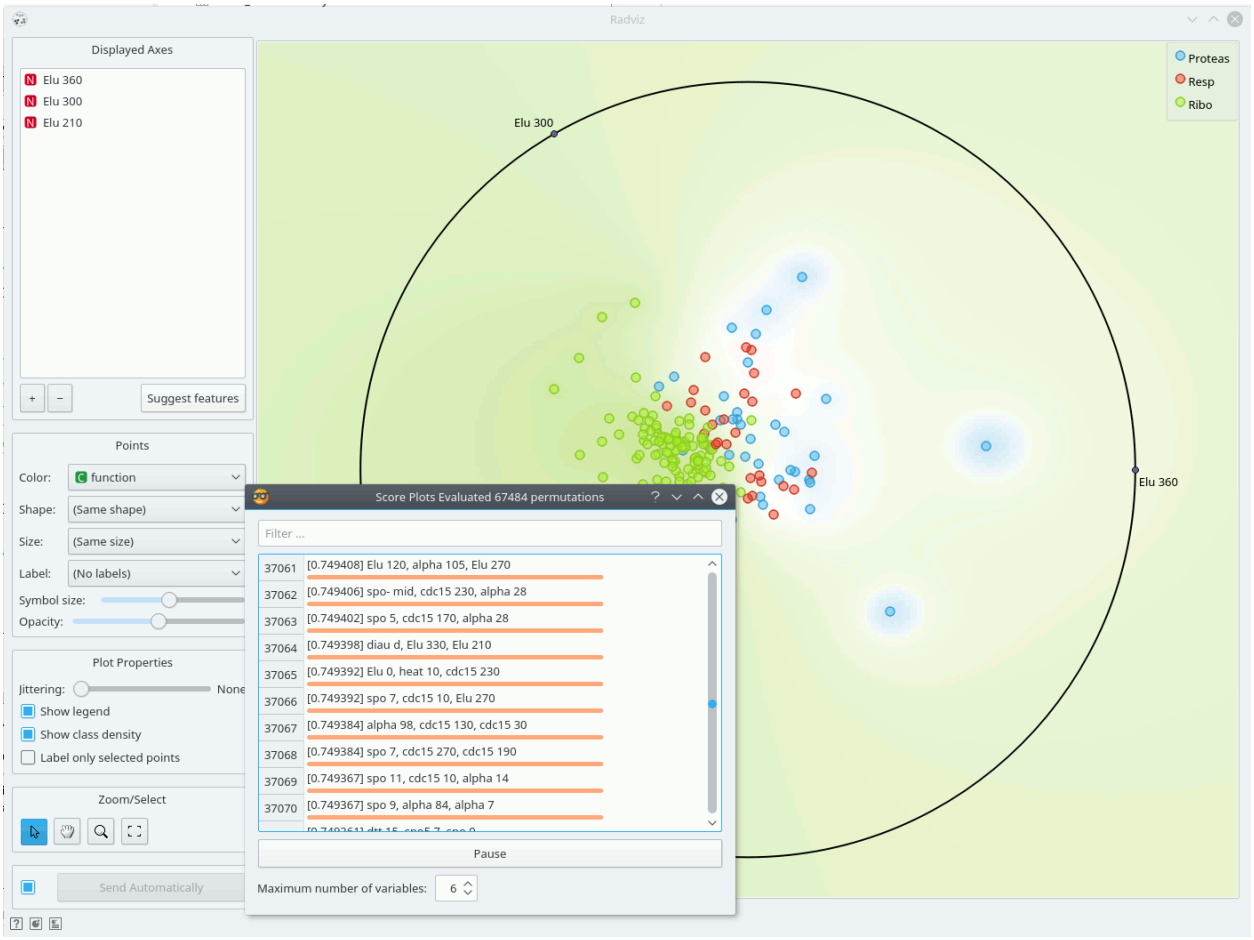
Outputs

- Selected Data: instances selected from the plot
- Data: data with an additional column showing whether a point is selected
- Components: Radviz vectors

Radviz (Hoffman et al. 1997) is a non-linear multi-dimensional visualization technique that can display data defined by three or more variables in a 2-dimensional projection. The visualized variables are presented as anchor points equally spaced around the perimeter of a unit circle. Data instances are shown as points inside the circle, with their positions determined by a metaphor from physics: each point is held in place with springs that are attached at the other end to the variable anchors. The stiffness of each spring is proportional to the value of the corresponding variable and the point ends up at the position where the spring forces are in equilibrium. Prior to visualization, variable values are scaled to lie between 0 and 1. Data instances that are close to a set of variable anchors have higher values for these variables than for the others.

The snapshot shown below shows a Radviz widget with a visualization of the dataset from functional genomics (Brown et al. 2000). In this particular visualization the data instances are colored according to the corresponding class, and the visualization space is colored according to the computed class probability. Notice that the particular visualization very nicely separates data instances of different class, making the visualization interesting and potentially informative.

Just like all point-based visualizations, this widget includes tools for intelligent data visualization (VizRank, see Leban et al. 2006) and an interface for explorative data analysis - selection of data points in visualization. Just like the Scatter Plot widget, it can be used to find a set of variables that would result in an interesting visualization. The Radviz graph above is according to this definition an example of a very good visualization, while the one below - where we show an VizRank's interface (*Suggest features* button) with a list of 3-attribute visualizations and their scores - is not.



References

Hoffman, P. E. et al. (1997) DNA visual and analytic data mining. In the Proceedings of the IEEE Visualization. Phoenix, AZ, pp. 437-441.

Brown, M. P., W. N. Grundy et al. (2000). "Knowledge-based analysis of microarray gene expression data by using support vector machines." Proc Natl Acad Sci U S A 97(1): 262-7.

Leban, G., B. Zupan et al. (2006). "VizRank: Data Visualization Guided by Machine Learning." Data Mining and Knowledge Discovery 13(2): 119-136.

Mramor, M., G. Leban, J. Demsar, and B. Zupan. Visualization-based cancer microarray data classification analysis. Bioinformatics 23(16): 2147-2154, 2007.

2.3 Model

2.3.1 Constant

Predict the most frequent class or mean value from the training set.

Inputs

- Data: input dataset

- Preprocessor: preprocessing method(s)

Outputs

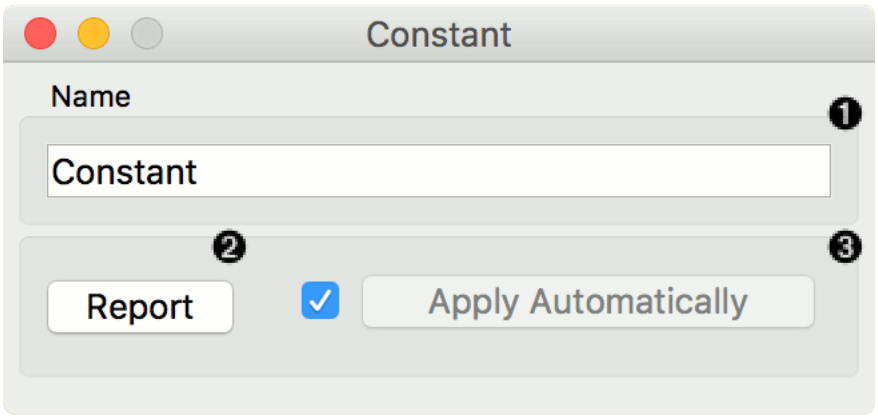
- Learner: majority/mean learning algorithm
- Model: trained model

This learner produces a model that always predicts themajority for classification tasks and mean value for regression tasks.

For classification, when predicting the class value with Predictions, the widget will return 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.

For regression, it *learns* the mean of the class variable and returns a predictor with the same mean value.

The widget is typically used as a baseline for other models.



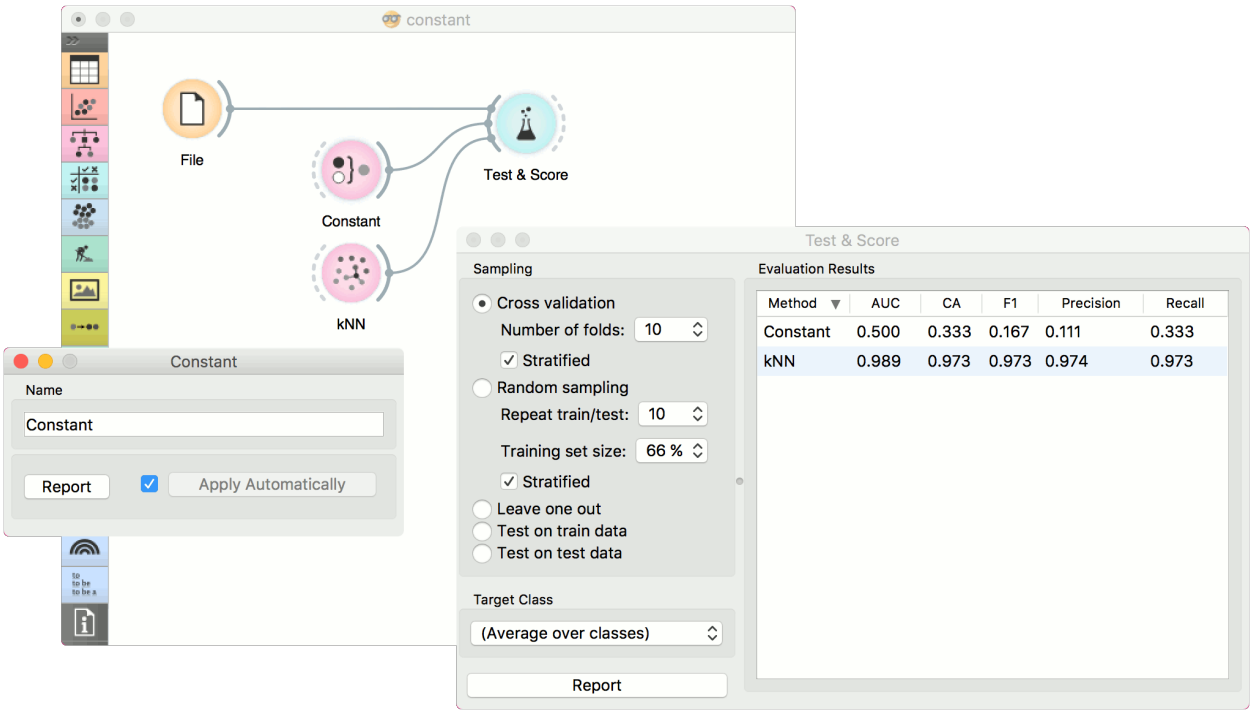
This widget provides the user with two options:

1. The name under which it will appear in other widgets. Default name is “Constant”.
2. Produce a report.

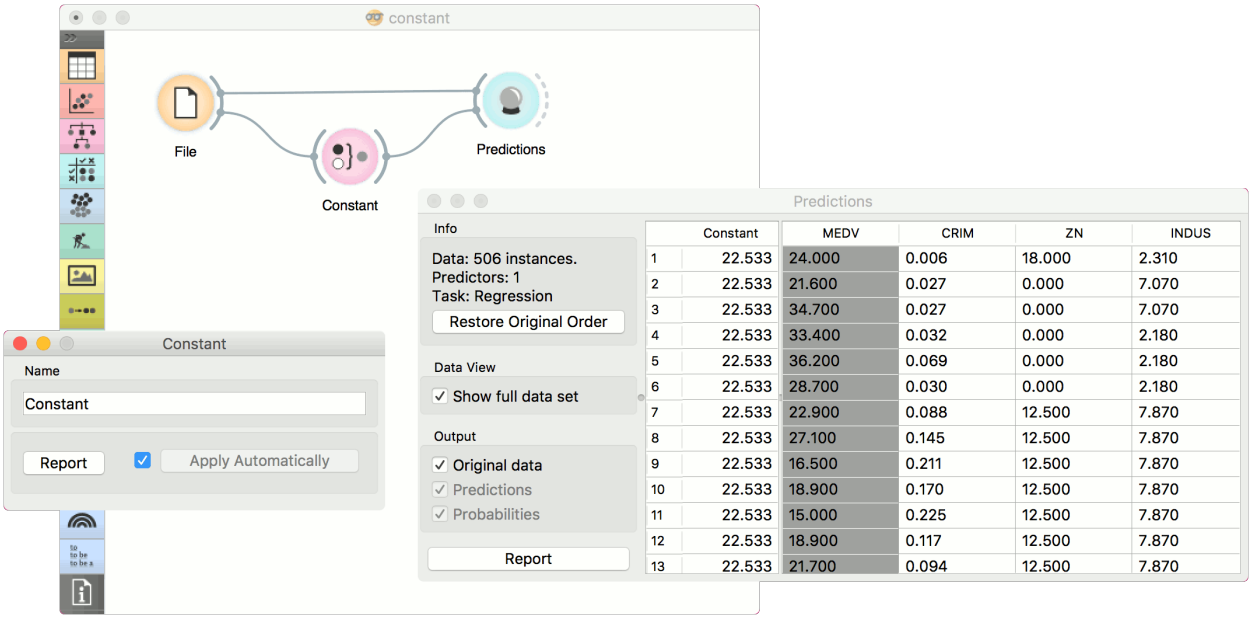
If you change the widget’s name, you need to click *Apply*. Alternatively, tick the box on the left side and changes will be communicated automatically.

Examples

In a typical classification example, we would use this widget to compare the scores of other learning algorithms (such as kNN) with the default scores. Use *iris* dataset and connect it to Test & Score. Then connect **Constant** and kNN to Test & Score and observe how well kNN performs against a constant baseline.



For regression, we use **Constant** to construct a predictor in Predictions. We used the *housing* dataset. In **Predictions**, you can see that *Mean Learner* returns one (mean) value for all instances.



2.3.2 CN2 Rule Induction

Induce rules from data using CN2 algorithm.

Inputs

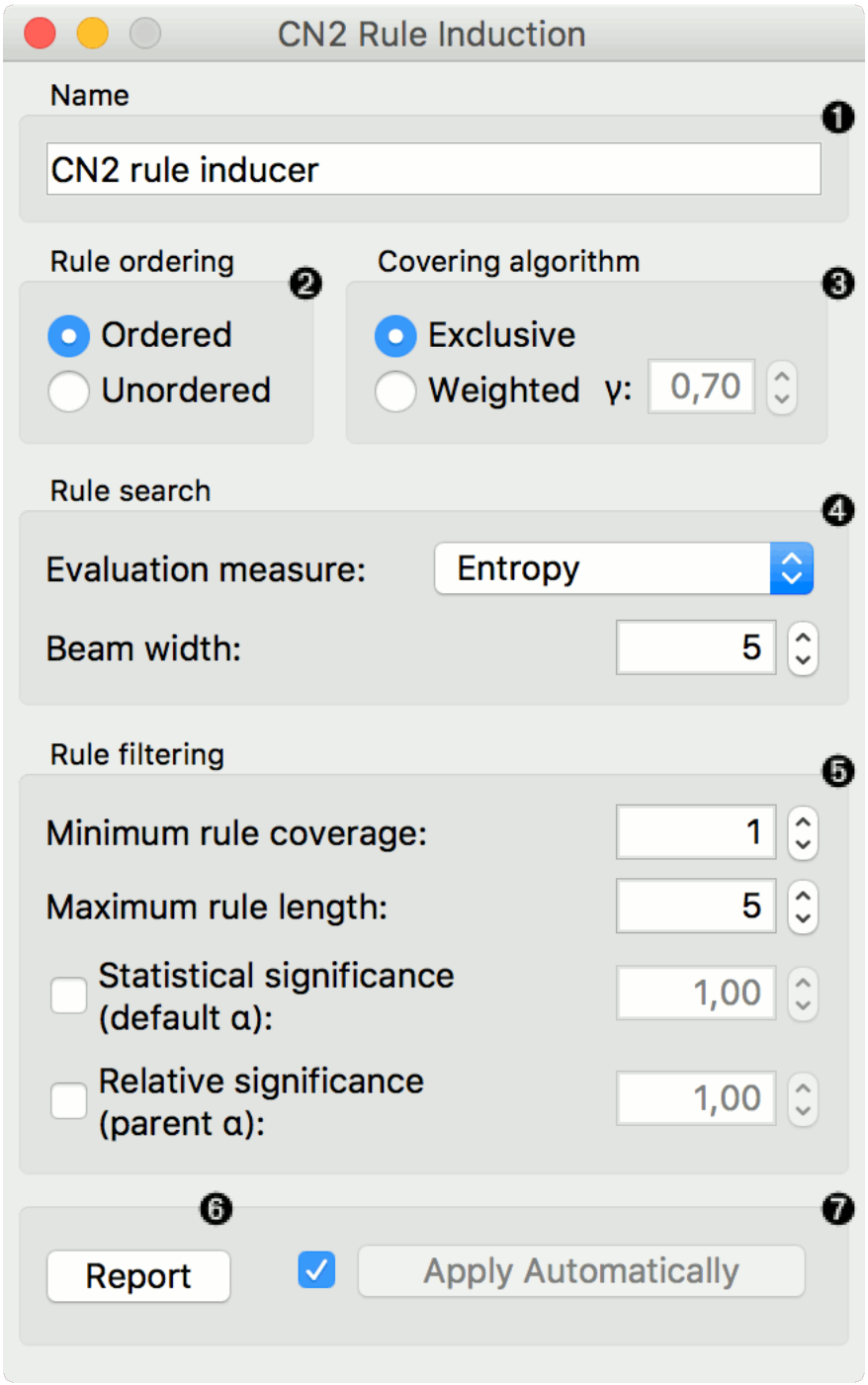
- Data: input dataset
- Preprocessor: preprocessing method(s)

Outputs

- Learner: CN2 learning algorithm
- CN2 Rule Classifier: trained model

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.

CN2 Rule Induction works only for classification.

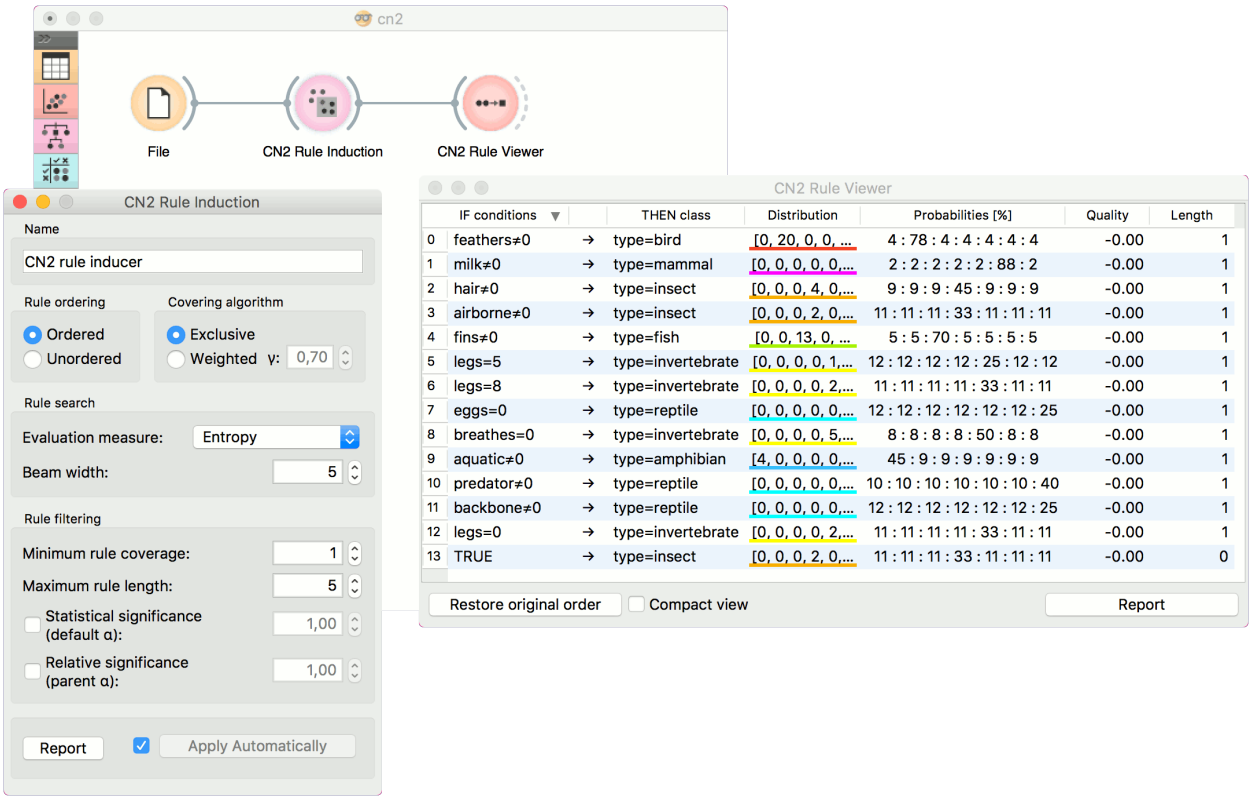


1. Name under which the learner appears in other widgets. The default name is *CN2 Rule Induction*.

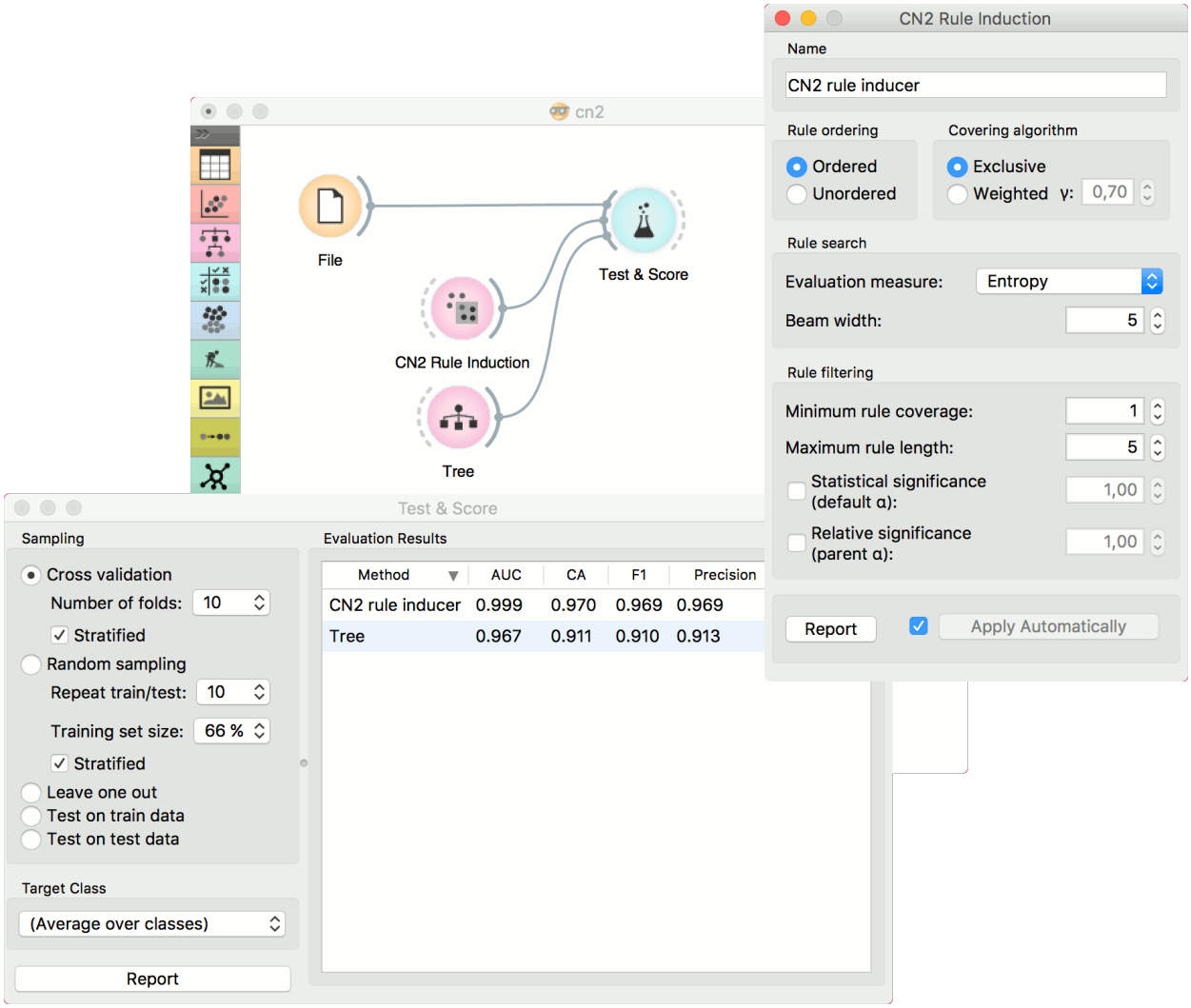
2. *Rule ordering*:
    - **Ordered**: induce ordered rules (decision list). Rule conditions are found and the majority class is assigned in the rule head.
    - **Unordered**: induce unordered rules (rule set). Learn rules for each class individually, in regard to the original learning data.
3. *Covering 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:
        - Entropy (measure of unpredictability of content)
        - Laplace Accuracy
        - 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

For the example below, we have used *zoo* dataset and passed it to **CN2 Rule Induction**. We can review and interpret the built model with CN2 Rule Viewer widget.



The second workflow tests evaluates **CN2 Rule Induction** and Tree in Test & Score.



References

1. Fürnkranz, Johannes. "Separate-and-Conquer Rule Learning", Artificial Intelligence Review 13, 3-54, 1999.
2. Clark, Peter and Tim Niblett. "The CN2 Induction Algorithm", Machine Learning Journal, 3 (4), 261-283, 1989.
3. Clark, Peter and Robin Boswell. "Rule Induction with CN2: Some Recent Improvements", Machine Learning - Proceedings of the 5th European Conference (EWSL-91),151-163, 1991.
4. Lavrač, Nada et al. "Subgroup Discovery with CN2-SD",Journal of Machine Learning Research 5, 153-188, 2004

2.3.3 kNN

Predict according to the nearest training instances.

Inputs

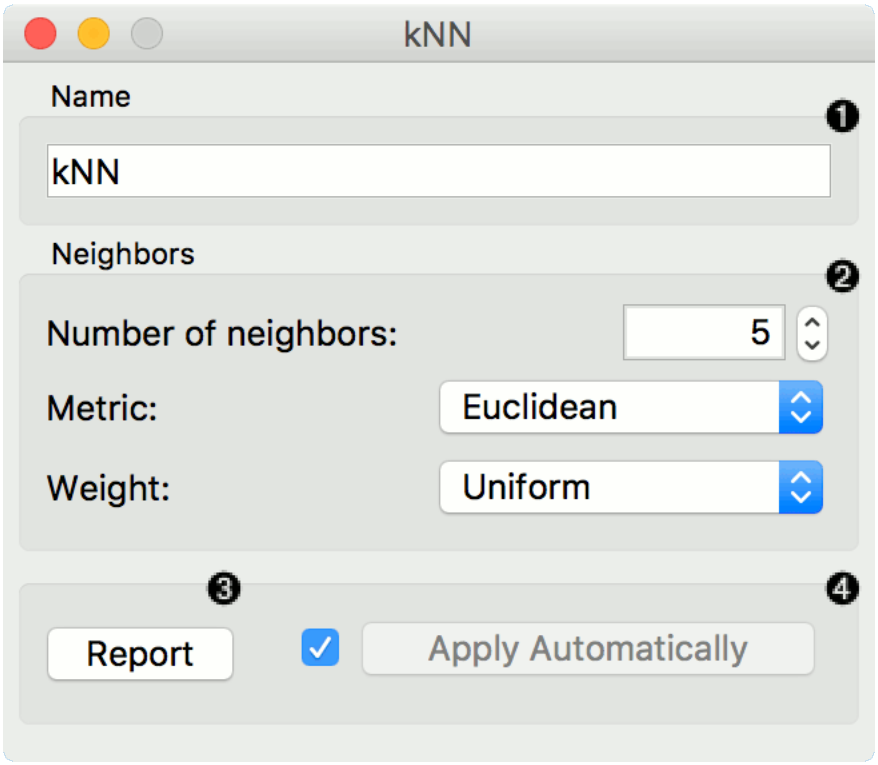
- Data: input dataset

- Preprocessor: preprocessing method(s)

Outputs

- Learner: kNN learning algorithm
- Model: trained model

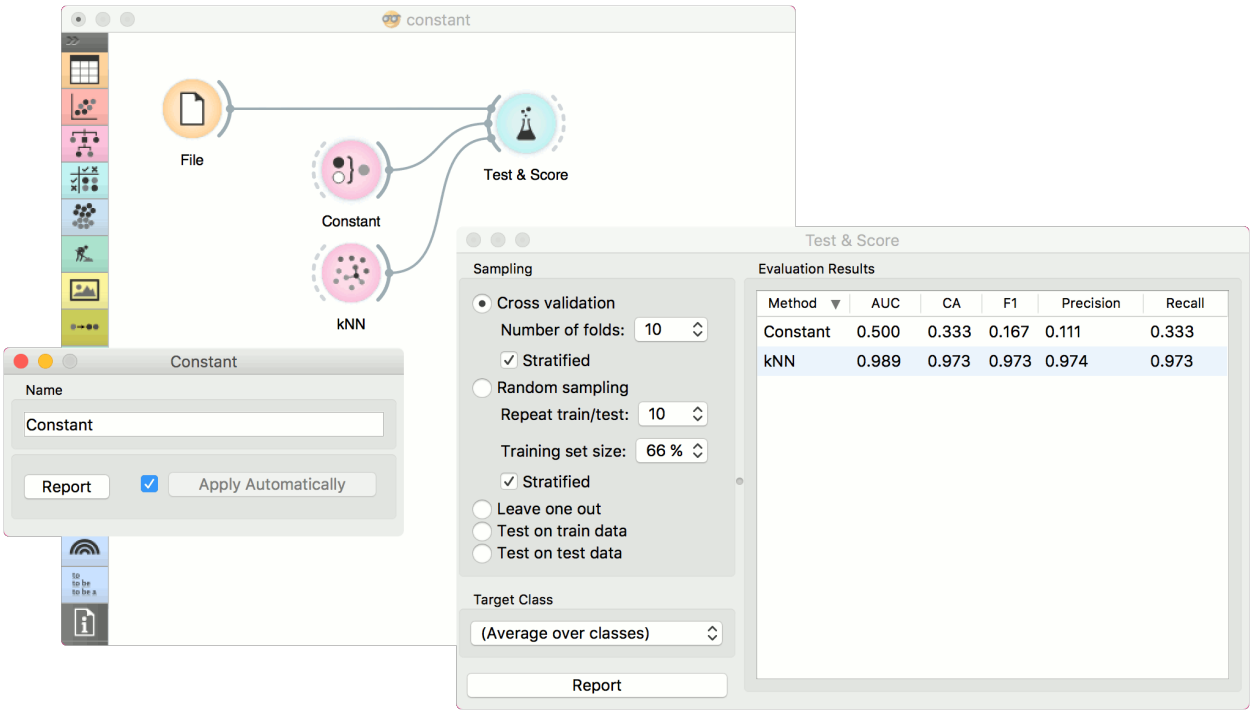
The **kNN** widget uses the kNN algorithm that searches for k closest training examples in feature space and uses their average as prediction.



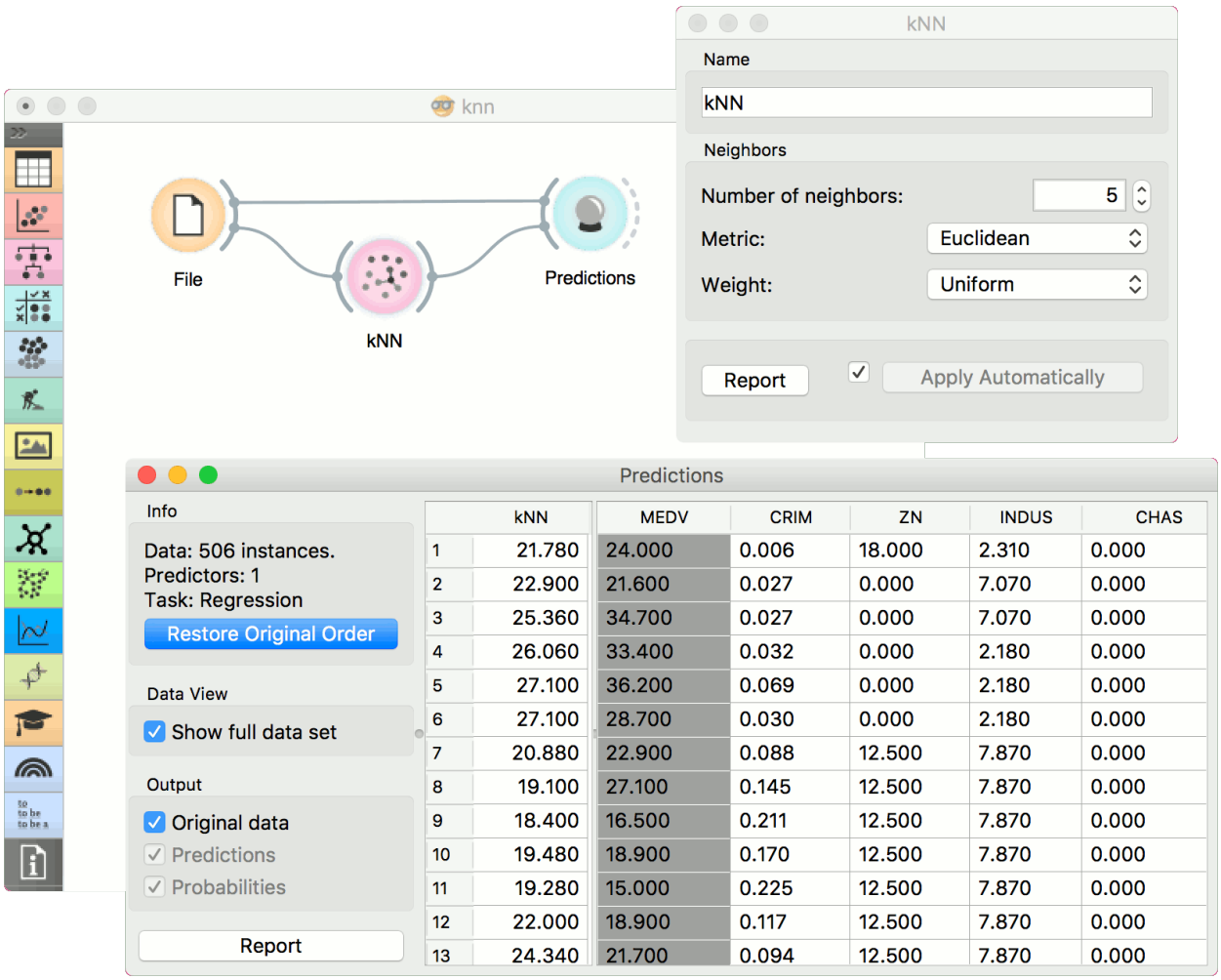
1. A name under which it will appear in other widgets. The default name is "kNN".
2. Set the number of nearest neighbors, the distance parameter (metric) and weights as model criteria.
   - Metric can be:
     - Euclidean ("straight line", distance between two points)
     - Manhattan (sum of absolute differences of all attributes)
     - Maximal (greatest of absolute differences between attributes)
     - Mahalanobis (distance between point and distribution).
   - 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.
3. Produce a report.
4. When you change one or more settings, you need to click *Apply*, which will put a new learner on the output and, if the training examples are given, construct a new model and output it as well. Changes can also be applied automatically by clicking the box on the left side of the *Apply* button.

Examples

The first example is a classification task on *iris* dataset. We compare the results of k-Nearest neighbors with the default model Constant, which always predicts the majority class.



The second example is a regression task. This workflow shows how to use the *Learner* output. For the purpose of this example, we used the *housing* dataset. We input the **kNN** prediction model into Predictions and observe the predicted values.



2.3.4 Tree

A tree algorithm with forward pruning.

Inputs

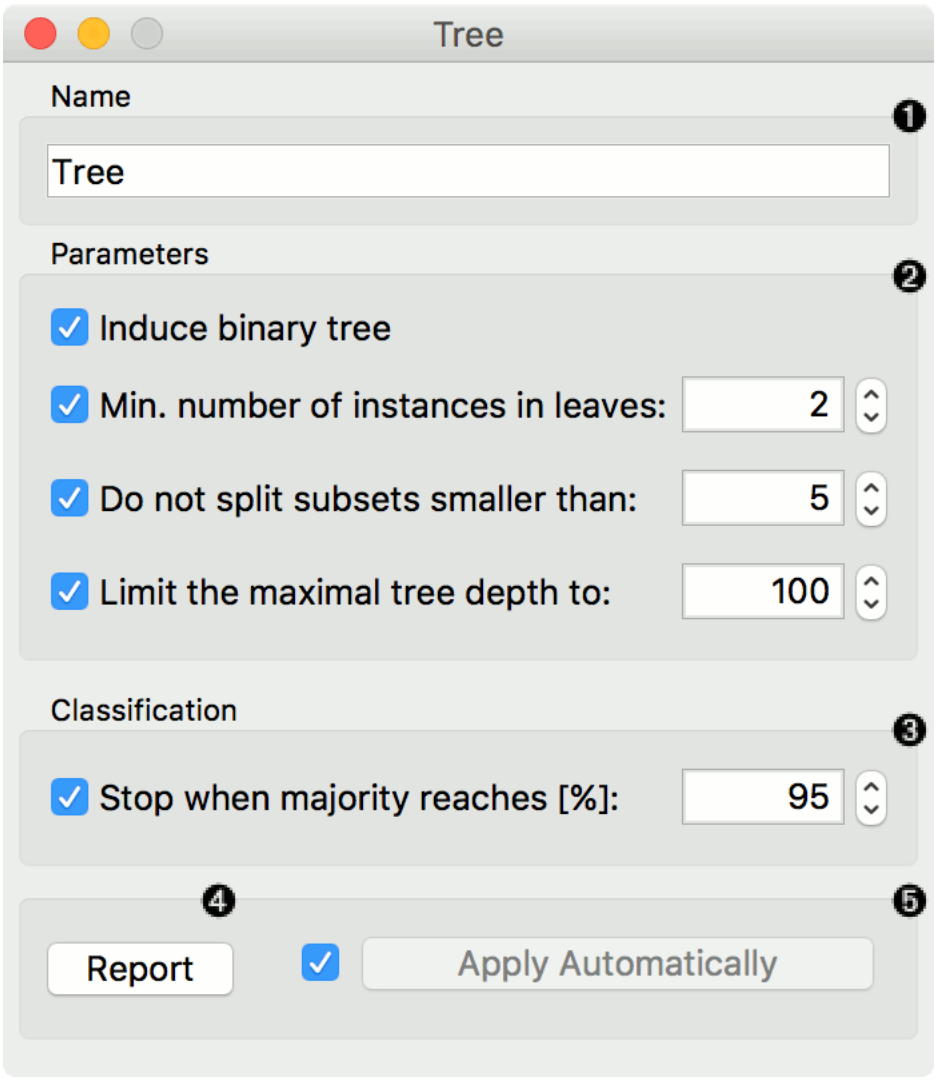
- Data: input dataset
- Preprocessor: preprocessing method(s)

Outputs

- Learner: decision tree learning algorithm
- Model: trained model

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

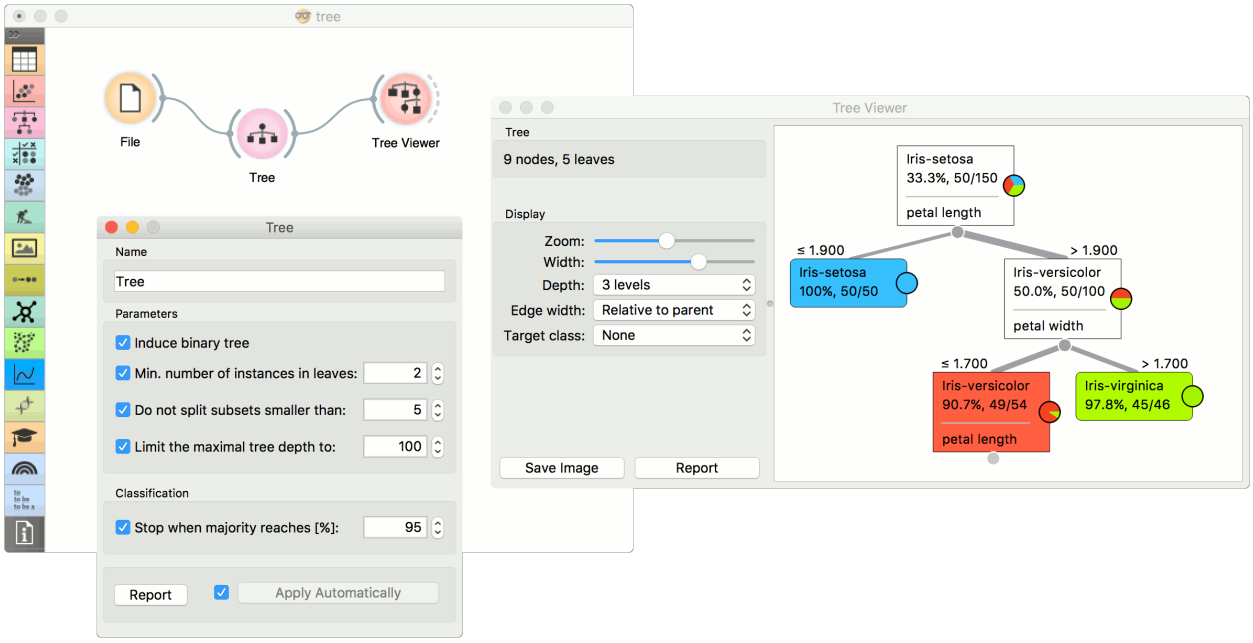
It can also be used for both classification and regression tasks.



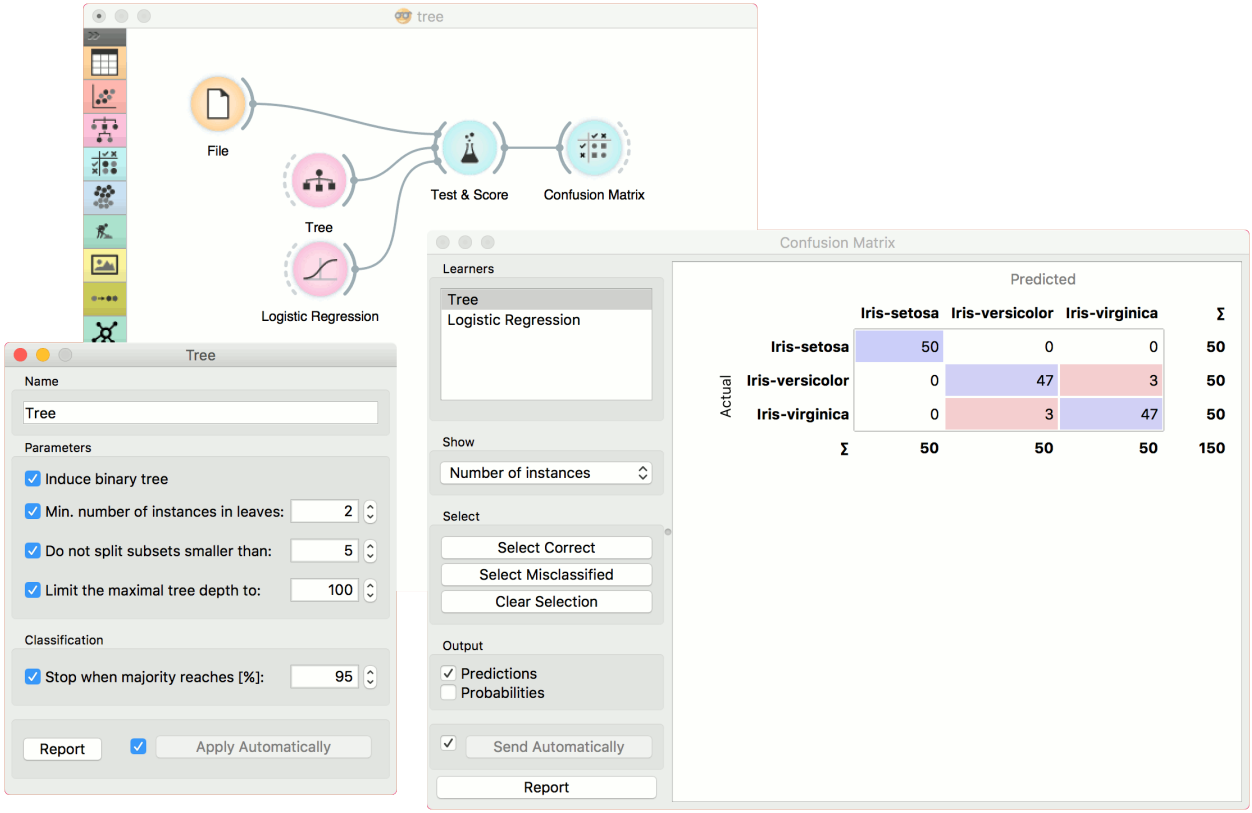
1. The learner can be given a name under which it will appear in other widgets. The default name is "Tree".
2. Tree parameters:
   - **Induce binary tree**: build a binary tree (split into two child nodes)
   - **Min. number of instances in leaves**: if checked, the algorithm will never construct a split 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.
   - **Limit the maximal tree depth**: limits the depth of the classification tree to the specified number of node levels.
3. **Stop when majority reaches [%]**: stop splitting the nodes after a specified majority threshold is reached
4. Produce a report. After changing the settings, you need to click *Apply*, which will put the new learner on 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 in Tree Viewer.

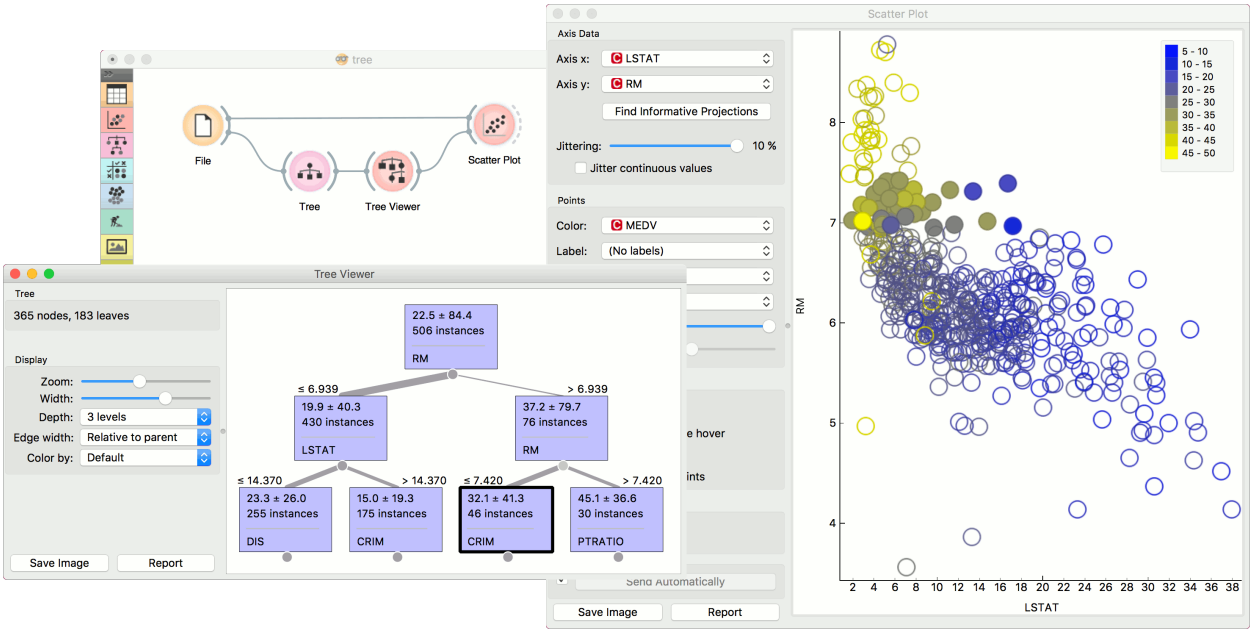


The second schema trains a model and evaluates its performance against Logistic Regression.



We used the *iris* dataset in both examples. However, **Tree** works for regression tasks as well. Use *housing* dataset

and pass it to **Tree**. The selected tree node from Tree Viewer is presented in the Scatter Plot and we can see that the selected examples exhibit the same features.



2.3.5 Random Forest

Predict using an ensemble of decision trees.

Inputs

- Data: input dataset
- Preprocessor: preprocessing method(s)

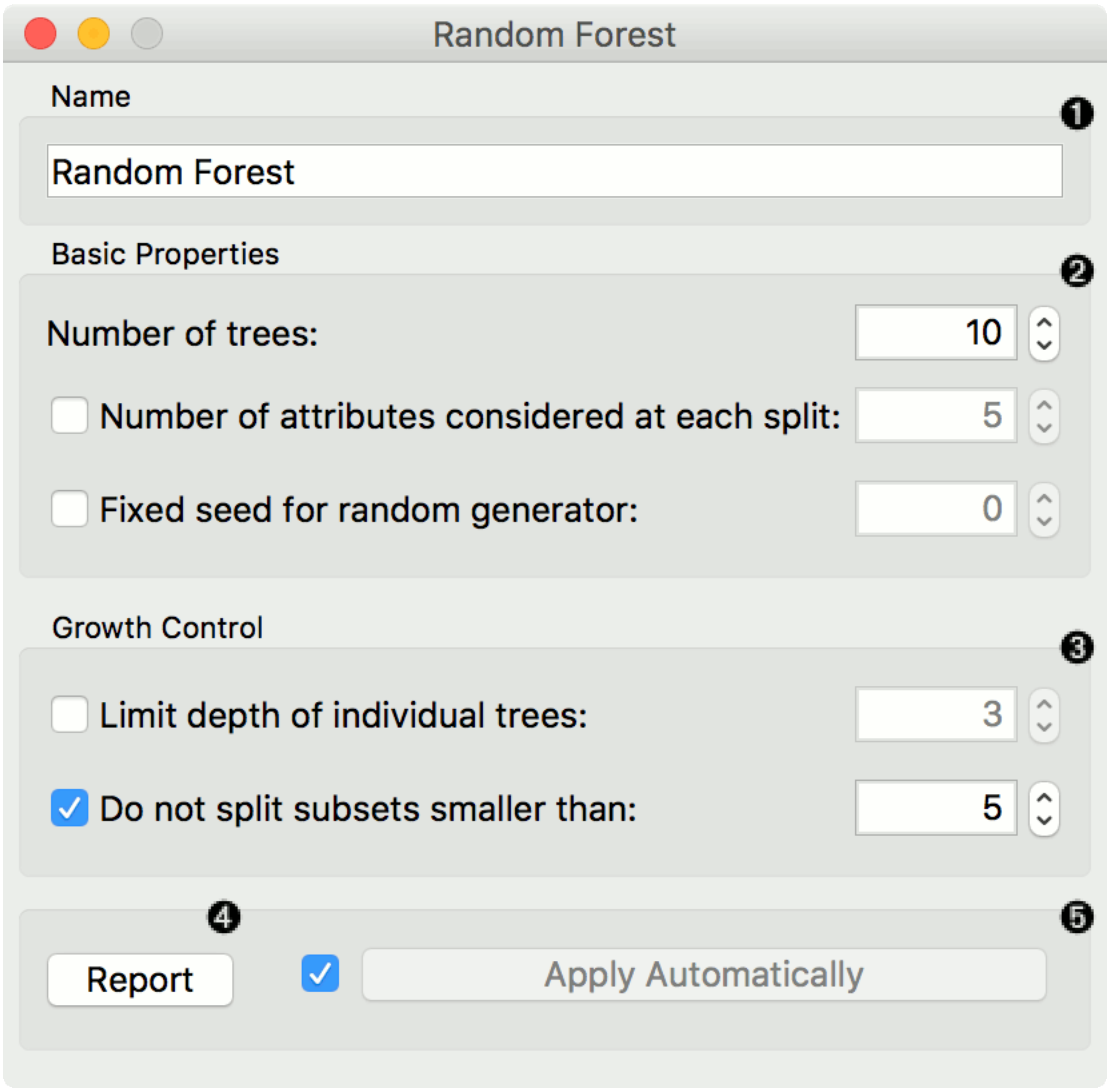
Outputs

- Learner: random forest learning algorithm
- Model: trained model

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 (Breiman, 2001) and Adele Cutler.

Random Forest builds a set of decision 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. The final model is based on the majority vote from individually developed trees in the forest.

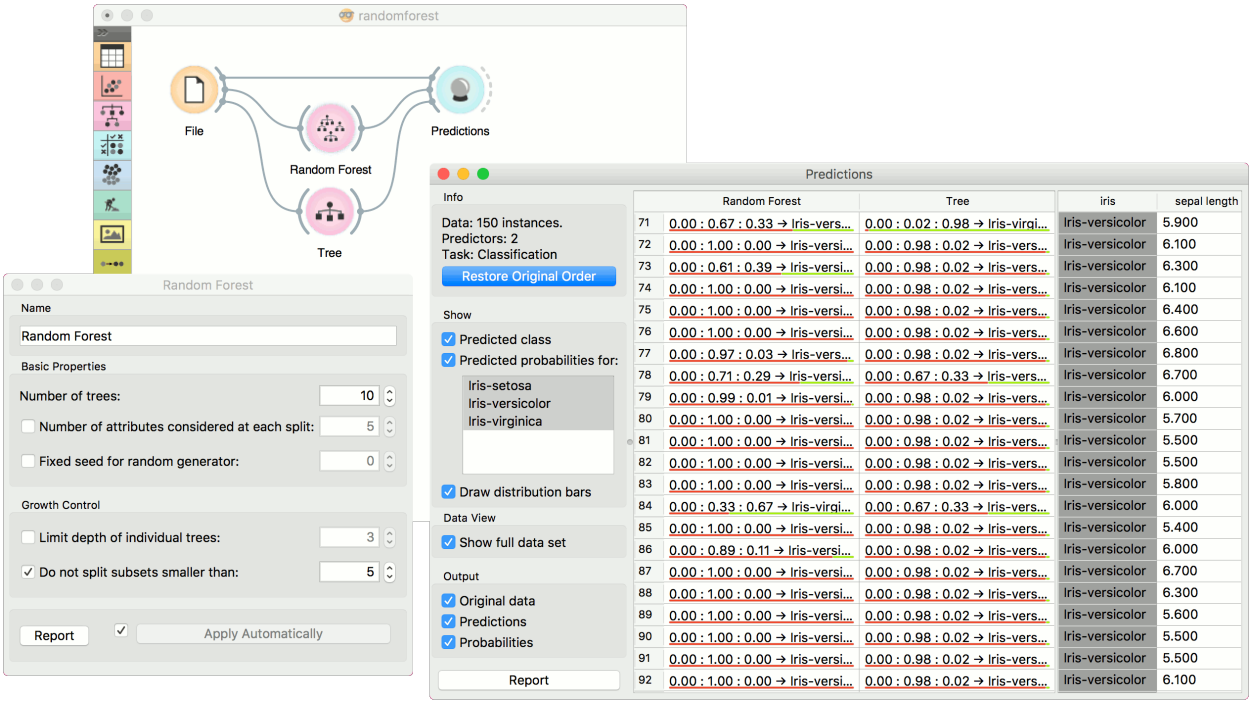
Random Forest works for both classification and regression tasks.



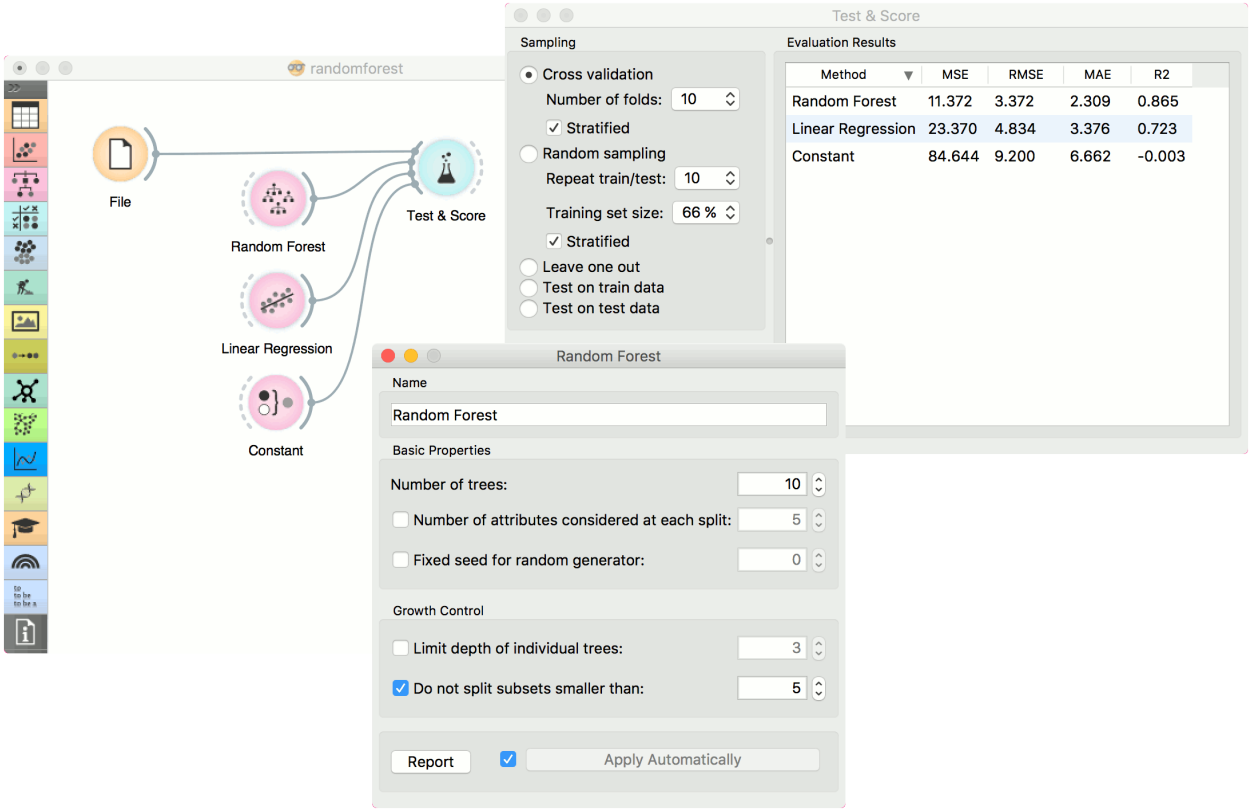
1. Specify the name of the model. The default name is "Random Forest".
2. Specify how many decision trees will be included in the forest (*Number of trees in the forest*), and how many attributes will be arbitrarily drawn for consideration at each node. If the latter is not specified (option *Number 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 fix the seed for tree generation (*Fixed seed for random generator*), which enables replicability of the results.
3. Original Breiman'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.
5. 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.

Examples

For classification tasks, we use *iris* dataset. Connect it to Predictions. Then, connect File to **Random Forest** and Tree and connect them further to Predictions. Finally, observe the predictions for the two models.



For regressions tasks, we will use *housing* data. Here, we will compare different models, namely **Random Forest**, Linear Regression and Constant, in the Test & Score widget.



References

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

2.3.6 SVM

Support Vector Machines map inputs to higher-dimensional feature spaces.

Inputs

- Data: input dataset
- Preprocessor: preprocessing method(s)

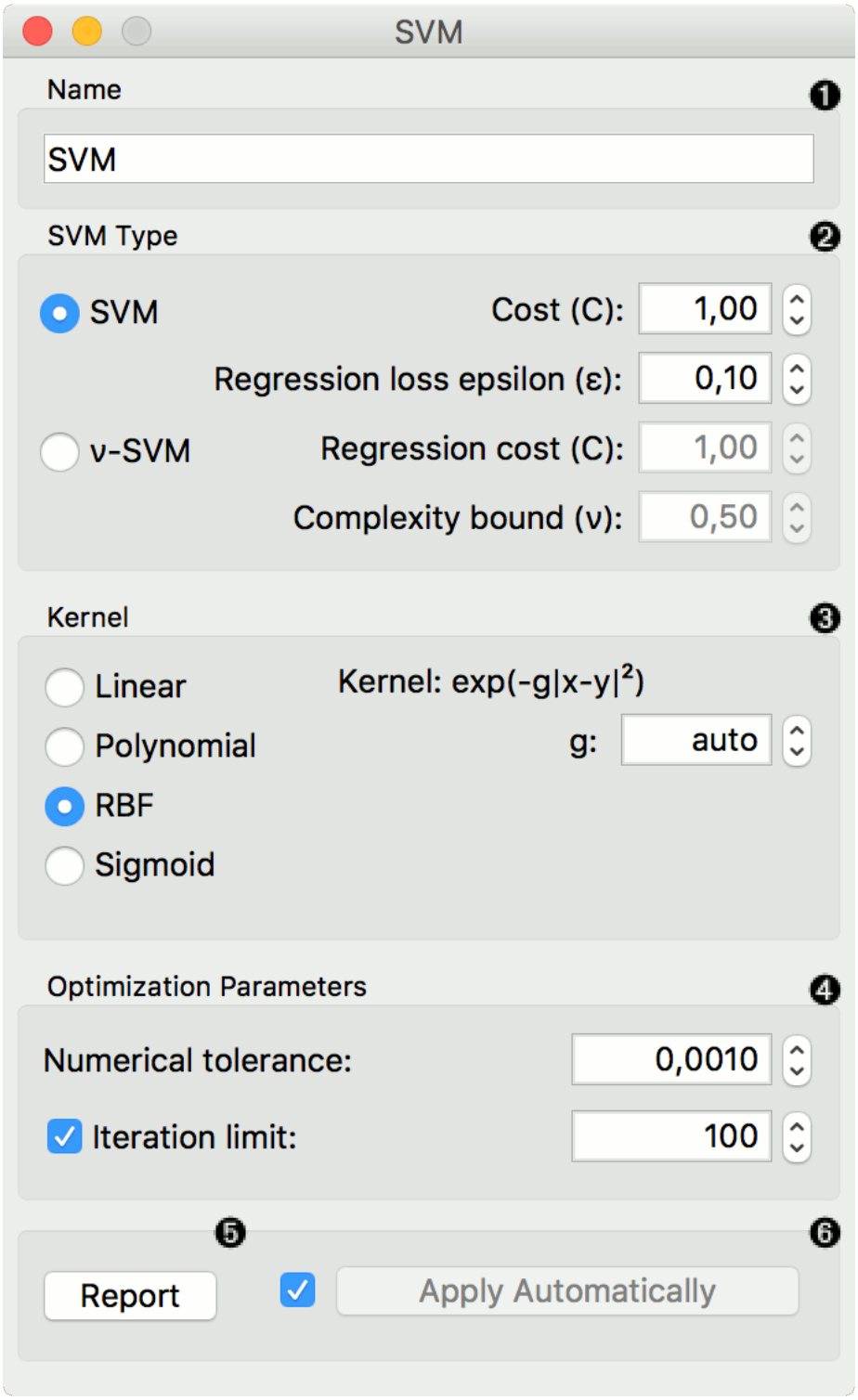
Outputs

- Learner: linear regression learning algorithm
- Model: trained model
- Support Vectors: instances used as support vectors

Support vector machine (SVM) is a machine learning technique that separates the attribute space with a hyperplane, thus maximizing the margin between the instances of different classes or class values. 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.

For regression tasks, **SVM** performs linear regression in a high dimension feature space using an ϵ-insensitive loss. Its estimation accuracy depends on a good setting of C, ϵ and kernel parameters. The widget outputs class predictions based on a SVM Regression.

The widget works for both classification and regression tasks.



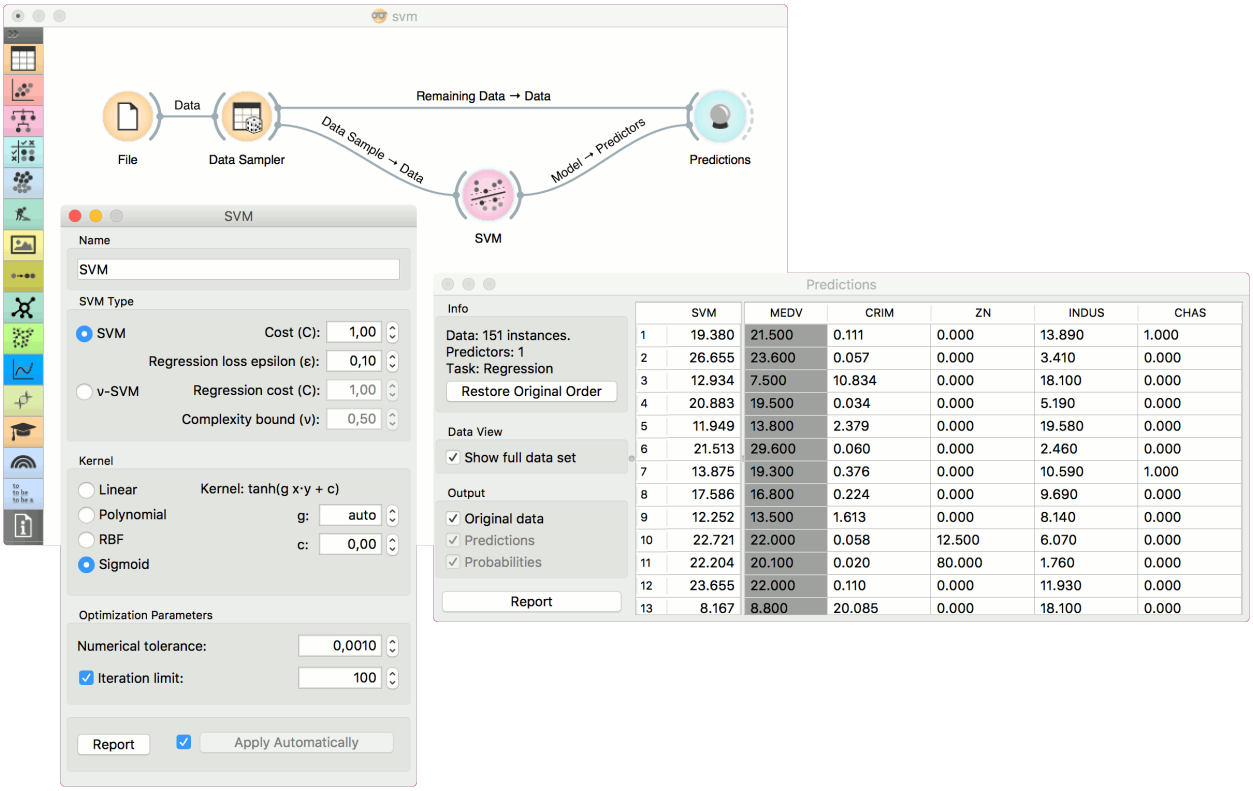
1. The learner can be given a name under which it will appear in other widgets. The default name is "SVM".
2. SVM type with test error settings. *SVM* and *ν-SVM* are based on different minimization of the error function. On the right side, you can set test error bounds:
   - SVM:

  - Cost: penalty term for loss and applies for classification and regression tasks.
  - ϵ: a parameter to the epsilon-SVR model, applies to regression tasks. Defines the distance from true values within which no penalty is associated with predicted values.
- ν-SVM:
  - Cost: penalty term for loss and applies only to regression tasks
  - ν: a parameter to the ν-SVR model, applies to classification and regression tasks. An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors.

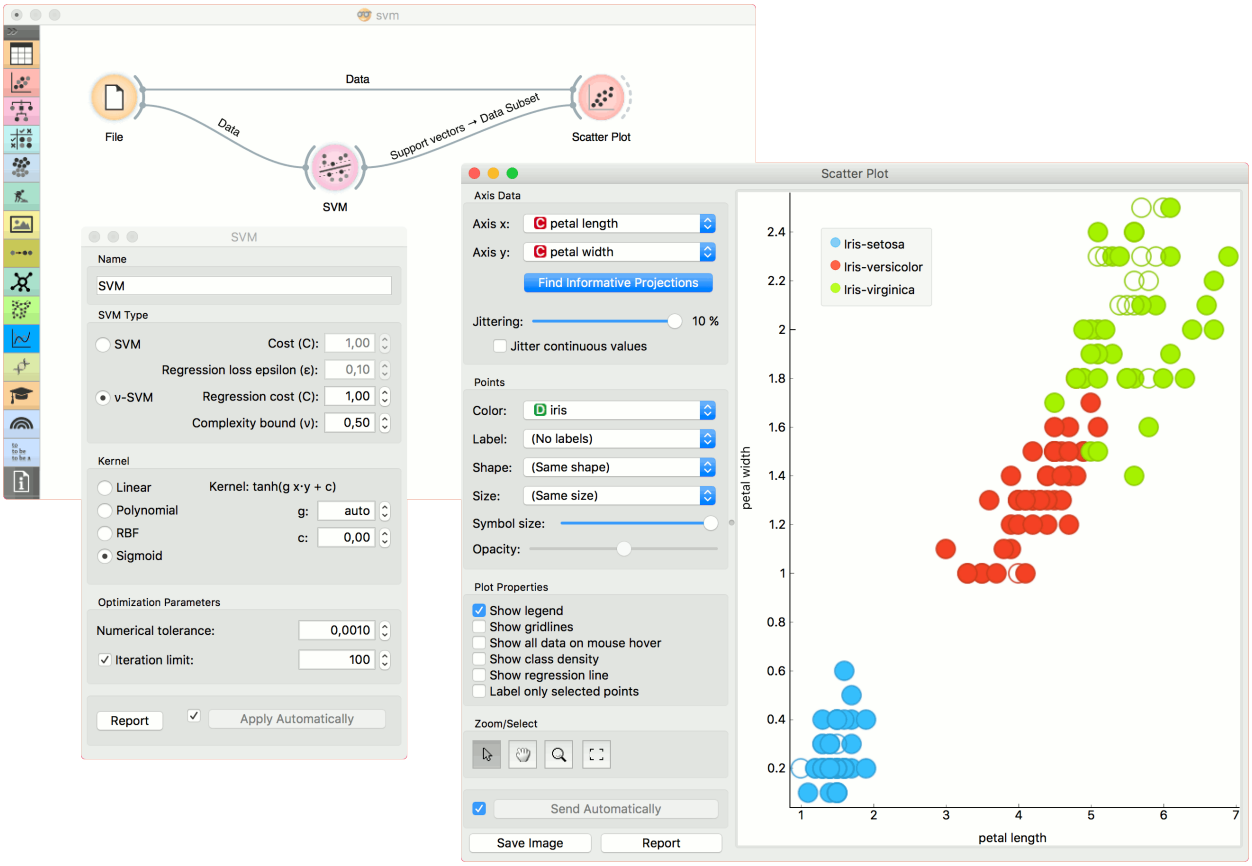
3. Kernel is a function that transforms attribute space to a new feature space to fit the maximum-margin hyperplane, thus allowing the algorithm to create the model with Linear, Polynomial, RBF and Sigmoid kernels. Functions that specify the kernel are presented upon selecting them, and the constants 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 c0 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

In the first (regression) example, we have used *housing* dataset and split the data into two data subsets (*Data Sample* and *Remaining Data*) with Data Sampler. The sample was sent to SVM which produced a *Model*, which was then used in Predictions to predict the values in *Remaining Data*. 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 example shows how to use **SVM** in combination with Scatter Plot. The following workflow trains a SVM model on *iris* data and outputs support vectors, which are those data instances that were used as support vectors in the learning phase. We can observe which are these data instances in a scatter plot visualization. Note that for the workflow to work correctly, you must set the links between widgets as demonstrated in the screenshot below.



References

Introduction to SVM on StatSoft.

2.3.7 Linear Regression

A linear regression algorithm with optional L1 (LASSO), L2 (ridge) or L1L2 (elastic net) regularization.

Inputs

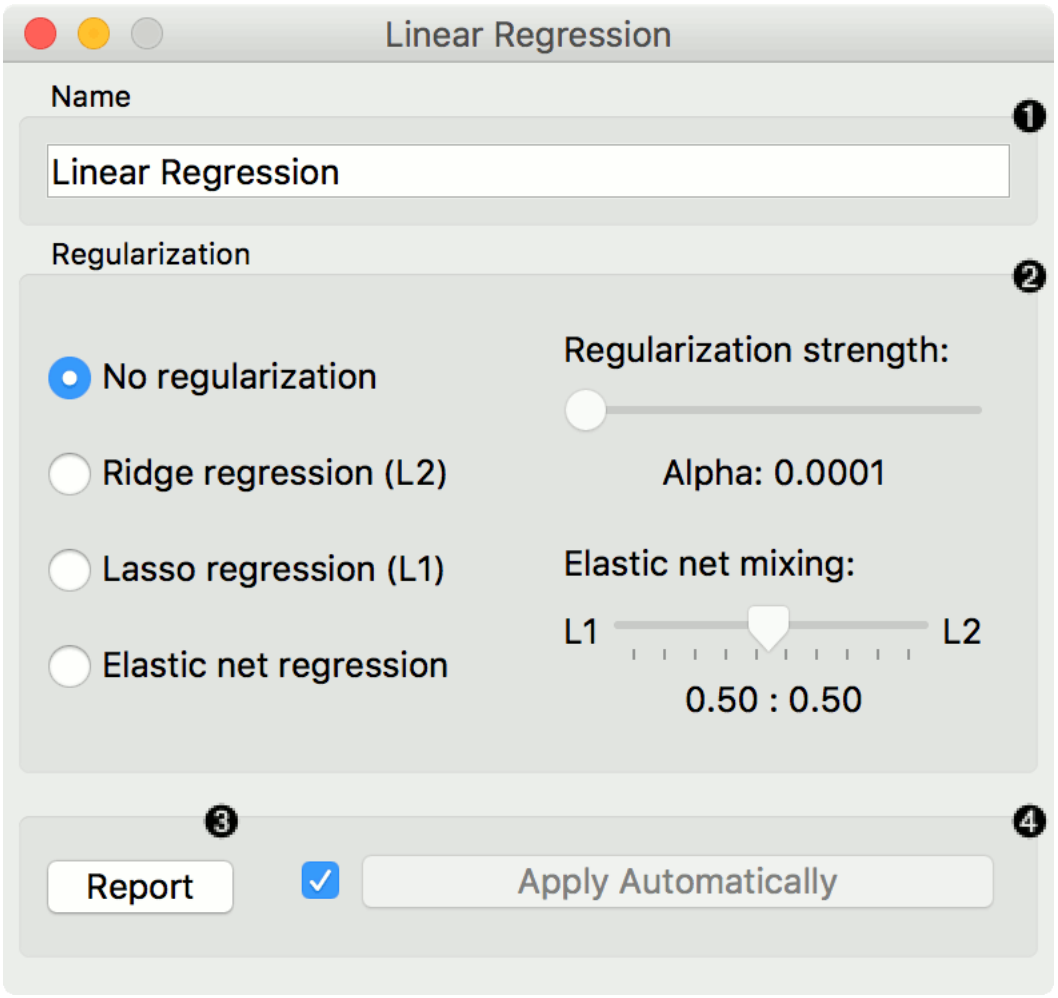
- Data: input dataset
- Preprocessor: preprocessing method(s)

Outputs

- Learner: linear regression learning algorithm
- Model: trained model
- Coefficients: linear regression coefficients

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 xi 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.

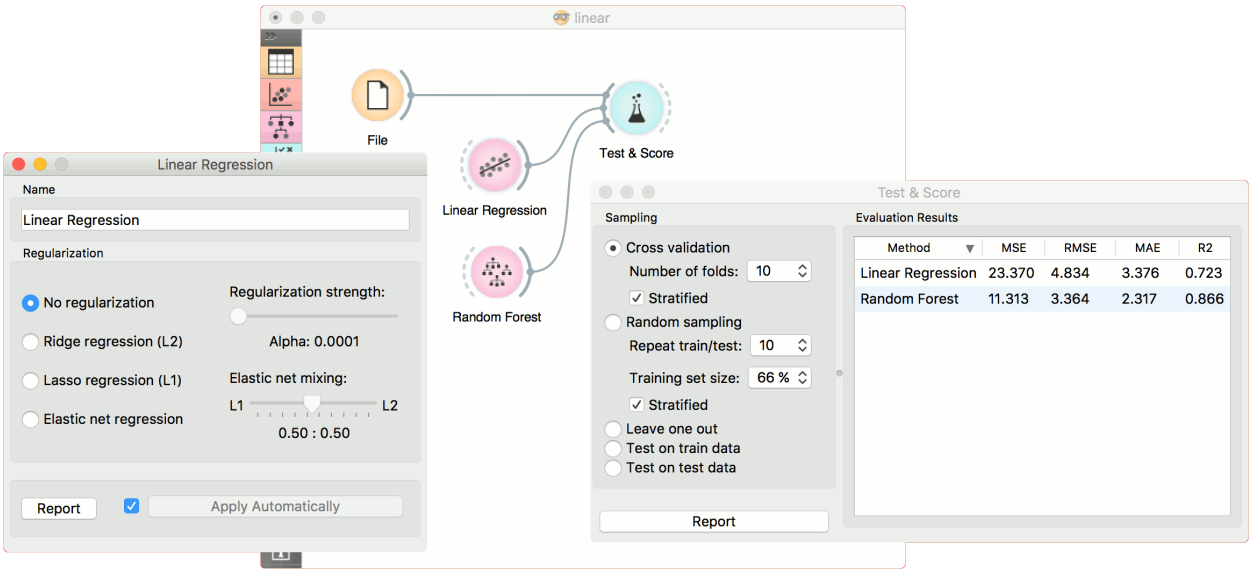
Linear regression works only on regression tasks.



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 with *housing* dataset. We trained **Linear Regression** and Random Forest and evaluated their performance in Test & Score.



2.3.8 Logistic Regression

The logistic regression classification algorithm with LASSO (L1) or ridge (L2) regularization.

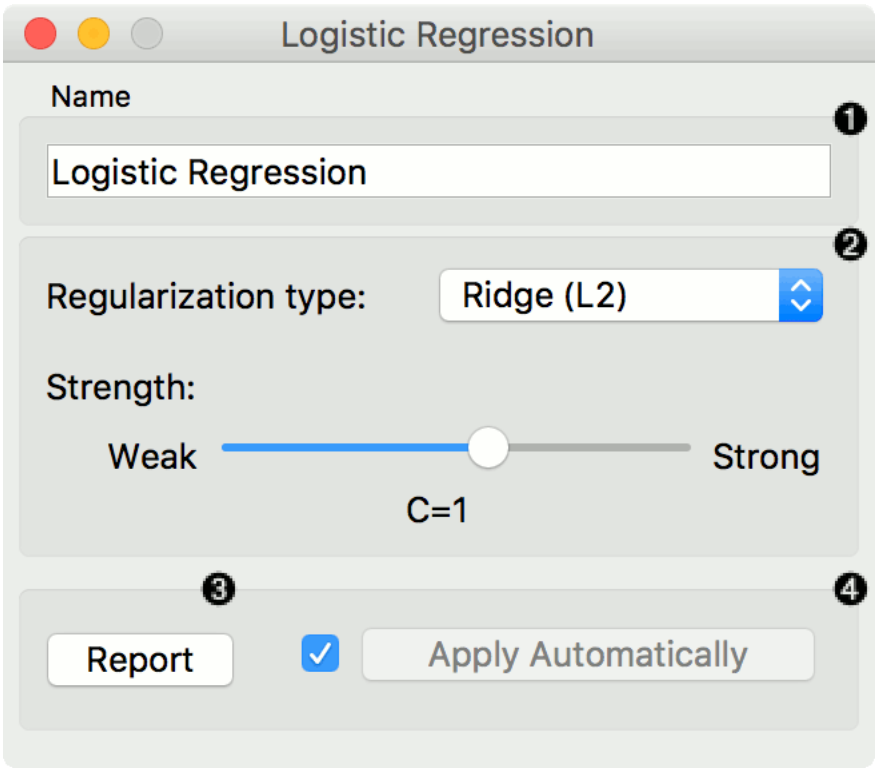
Inputs

- Data: input dataset
- Preprocessor: preprocessing method(s)

Outputs

- Learner: logistic regression learning algorithm
- Model: trained model
- Coefficients: logistic regression coefficients

Logistic Regression learns a Logistic Regression model from the data. It only works for classification tasks.

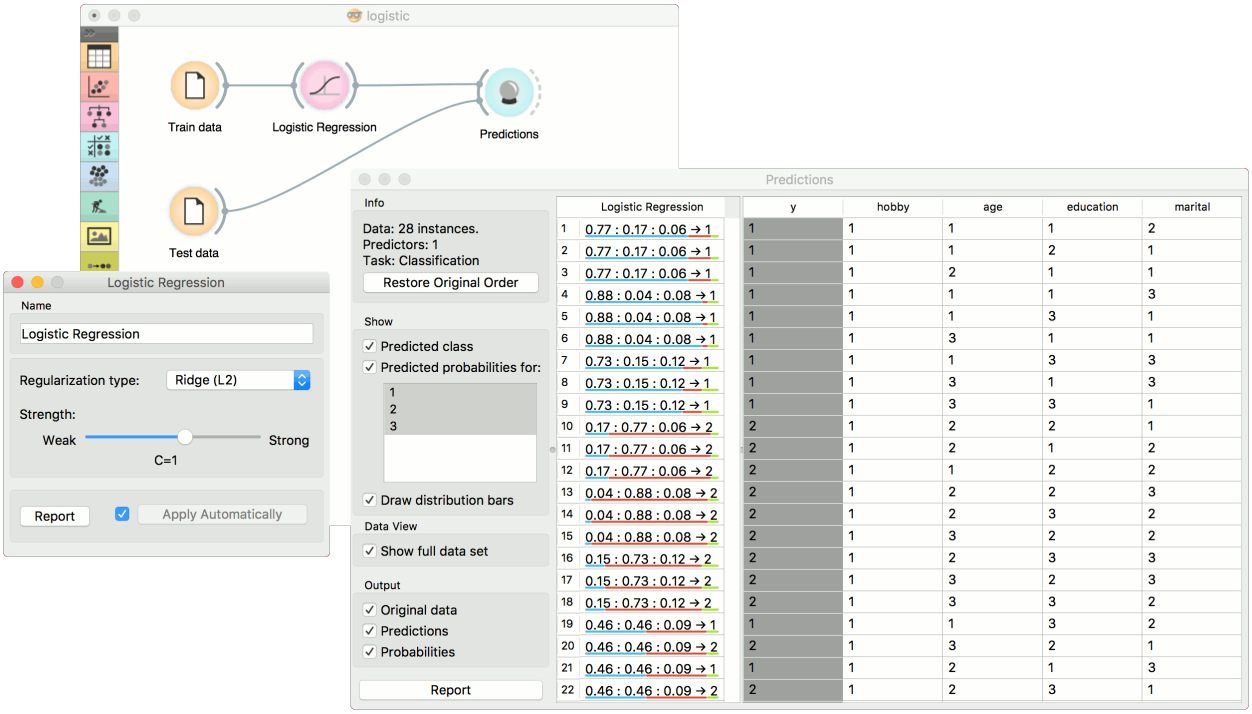


1. A name under which the learner appears in other widgets. The default name is "Logistic Regression".
2. Regularization type (either L1 or L2). Set the cost strength (default is C=1).
3. Press *Apply* to commit changes. If *Apply Automatically* is ticked, changes will be communicated automatically.

Example

The widget is used just as any other widget for inducing a classifier. This is an example demonstrating prediction results with logistic regression on the *hayes-roth* dataset. We first load *hayes-roth_learn* in the File widget and pass the data to **Logistic Regression**. Then we pass the trained model to Predictions.

Now we want to predict class value on a new dataset. We load *hayes-roth_test* in the second **File** widget and connect it to **Predictions**. We can now observe class values predicted with **Logistic Regression** directly in **Predictions**.



2.3.9 Naive Bayes

A fast and simple probabilistic classifier based on Bayes' theorem with the assumption of feature independence.

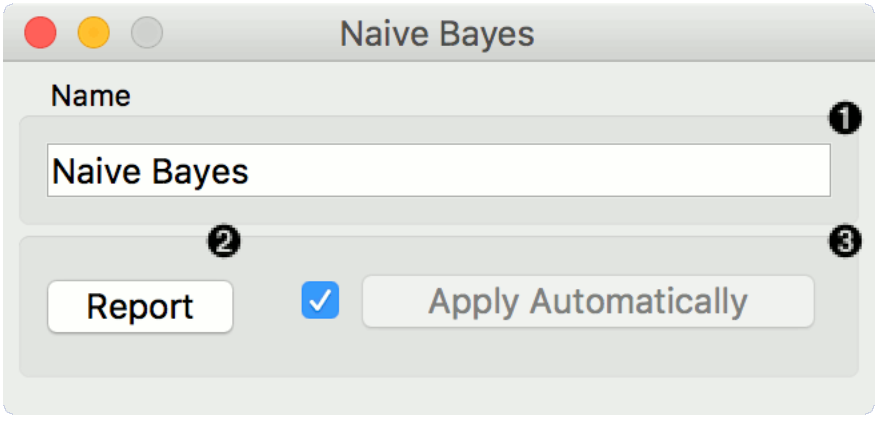
Inputs

- Data: input dataset
- Preprocessor: preprocessing method(s)

Outputs

- Learner: naive bayes learning algorithm
- Model: trained model

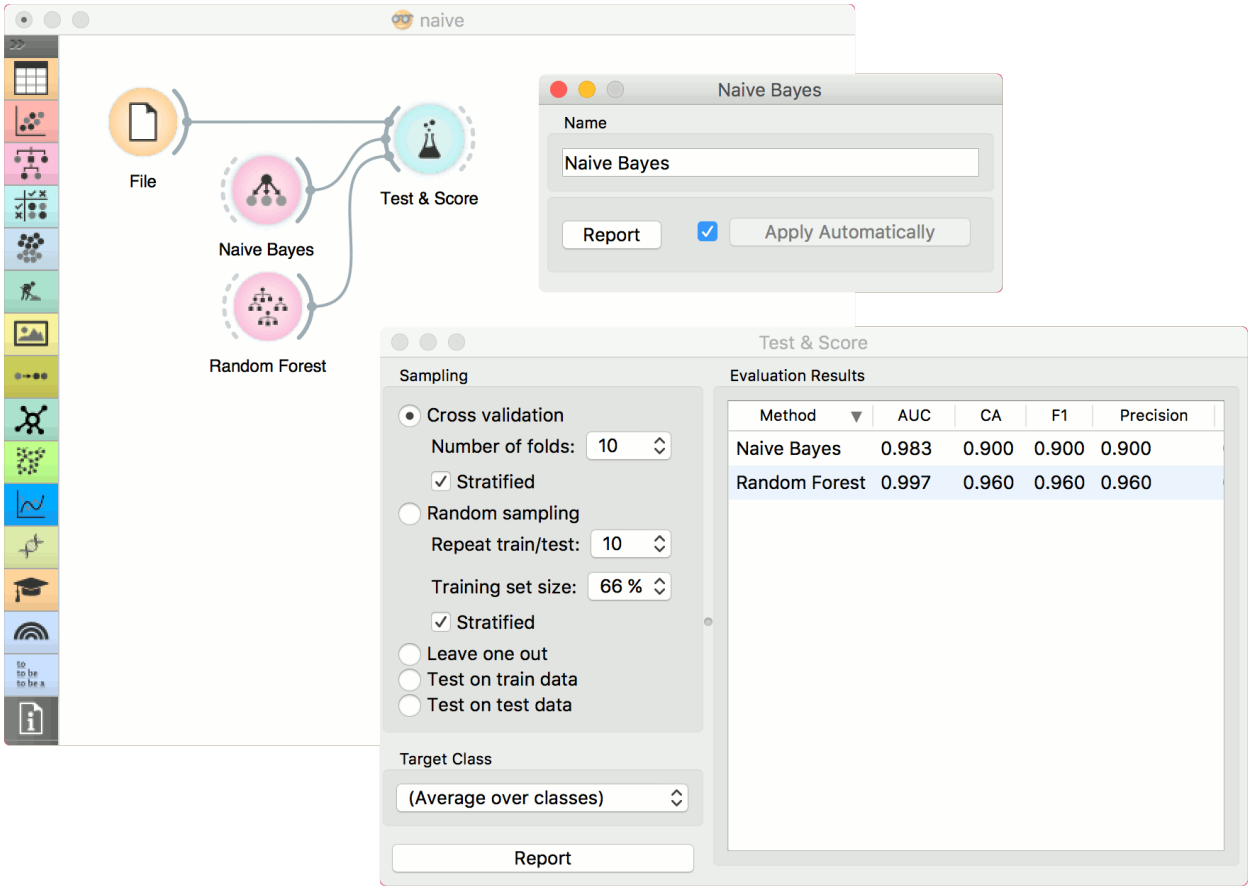
Naive Bayes learns a Naive Bayesian model from the data. It only works for classification tasks.



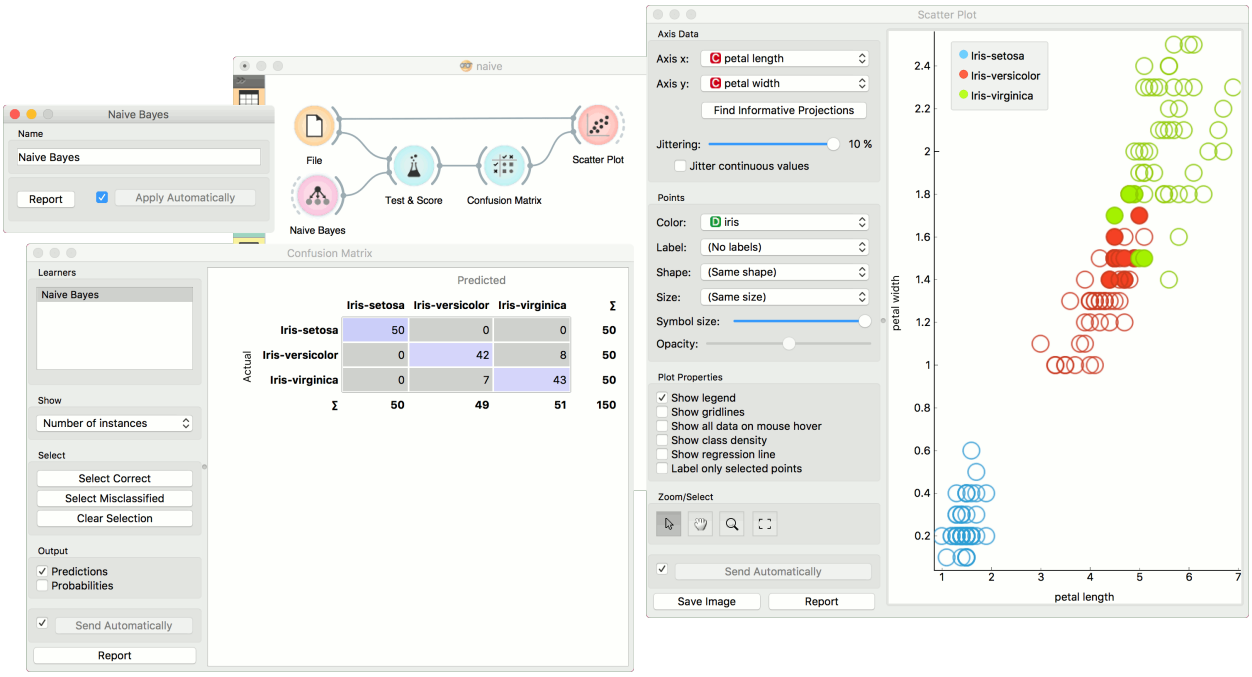
This widget has two options: the name under which it will appear in other widgets and producing a report. The default name is *Naive Bayes*. When you change it, you need to press *Apply*.

Examples

Here, we present two uses of this widget. First, we compare the results of the **Naive Bayes** with another model, the Random Forest. We connect *iris* data from File to Test & Score. We also connect **Naive Bayes** and Random Forest to **Test & Score** and observe their prediction scores.



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. We also connect Scatter Plot with **File**. Then we select the misclassified instances in the **Confusion Matrix** and show feed them to Scatter Plot. The bold dots in the scatterplot are the misclassified instances from **Naive Bayes**.



2.3.10 AdaBoost

An ensemble meta-algorithm that combines weak learners and adapts to the 'hardness' of each training sample.

Inputs

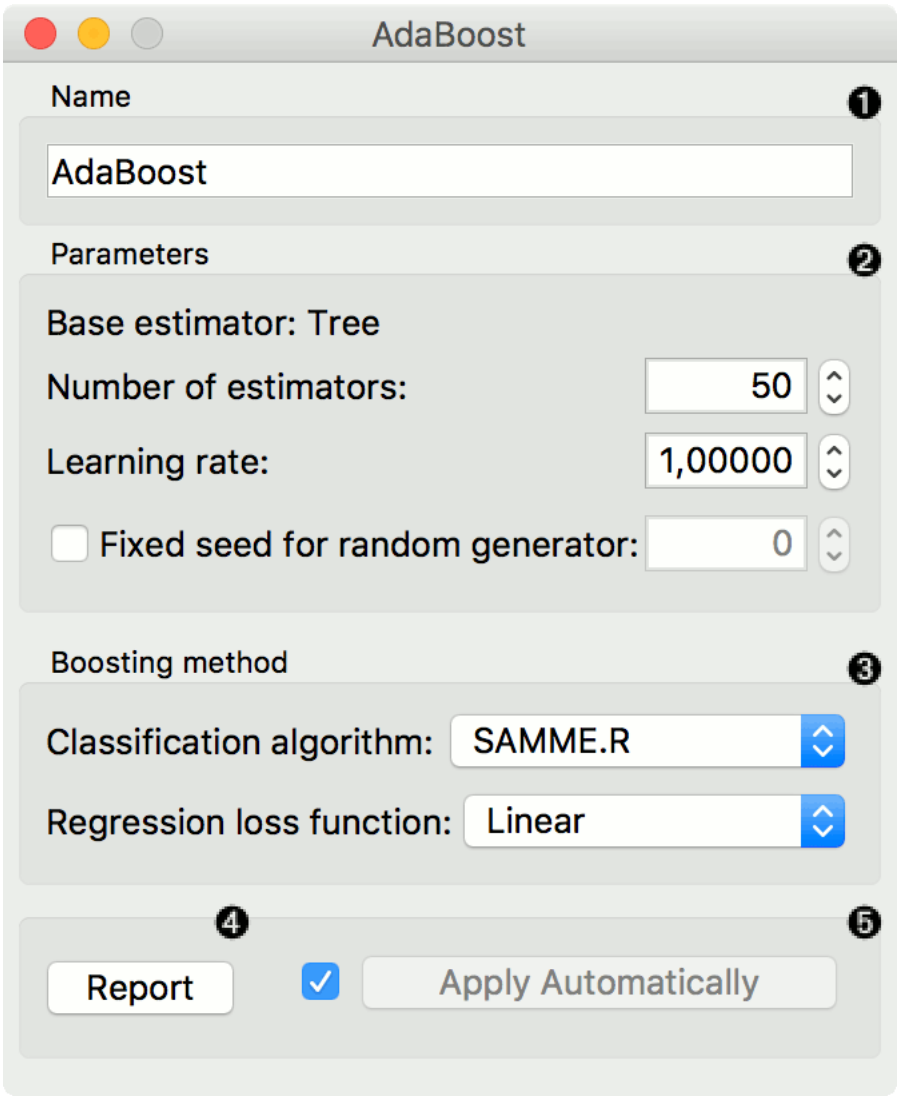
- Data: input dataset
- Preprocessor: preprocessing method(s)
- Learner: learning algorithm

Outputs

- Learner: AdaBoost learning algorithm
- Model: trained model

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.

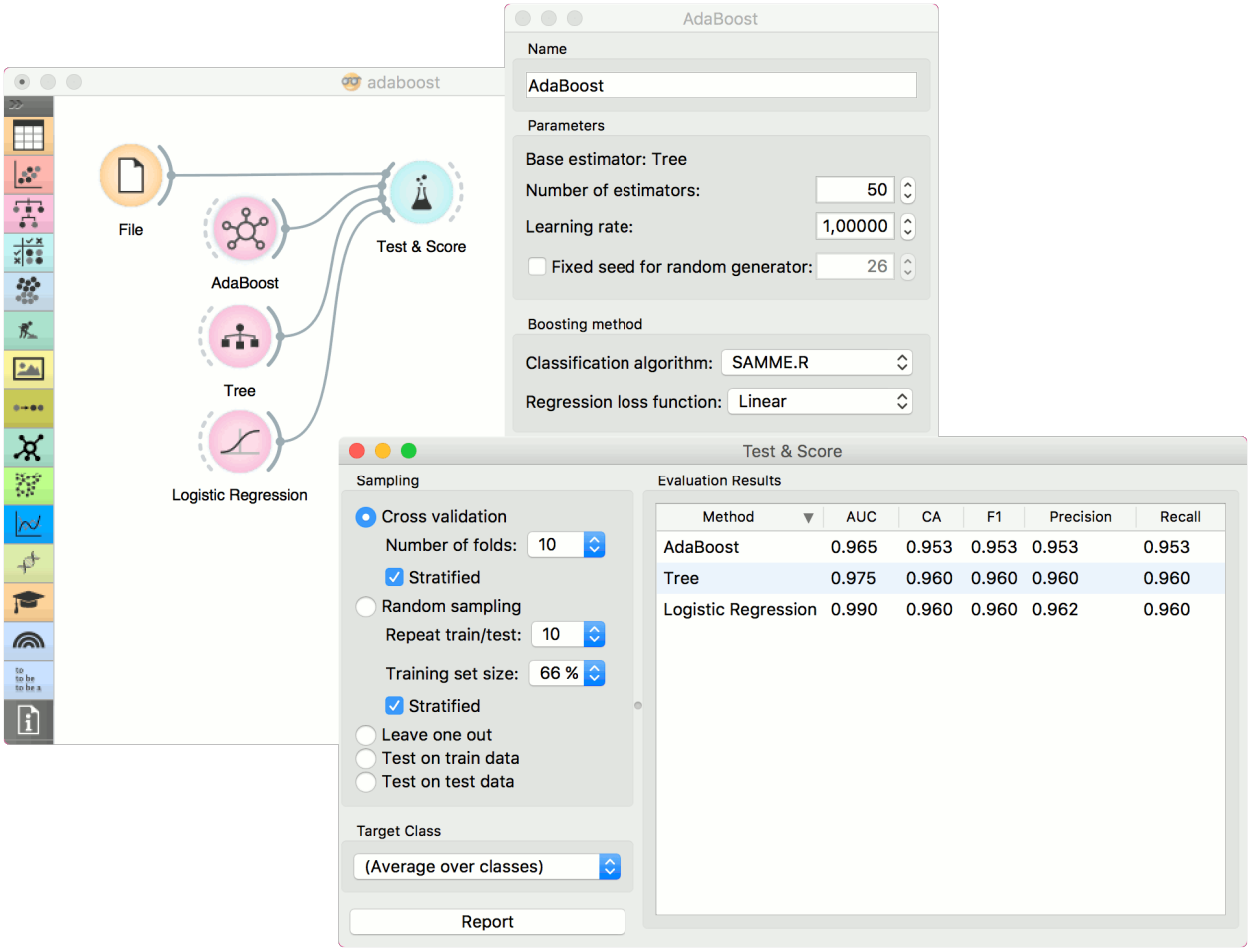
AdaBoost works for both classification and regression.



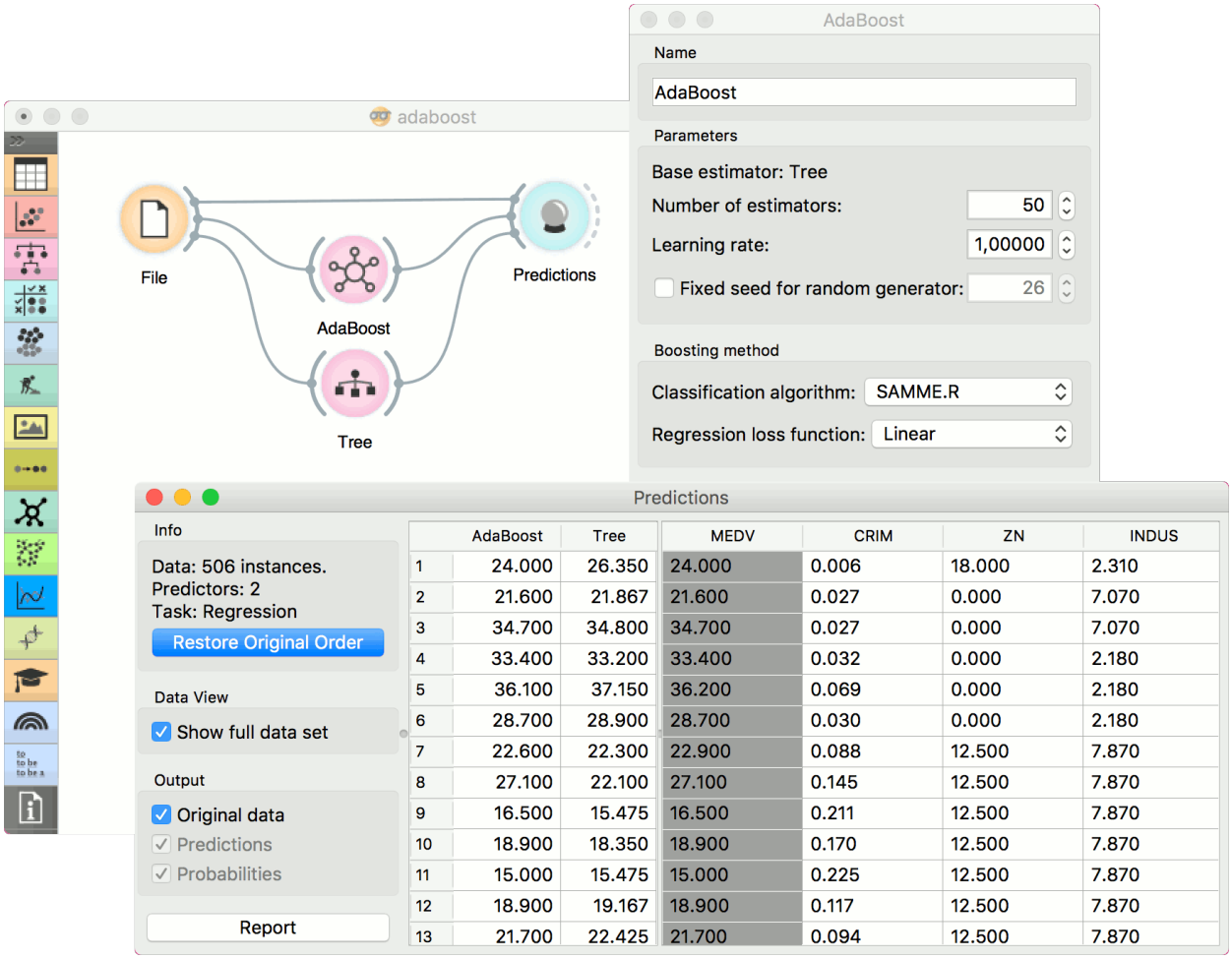
1. The learner can be given a name under which it will appear in other widgets. The default name is "AdaBoost".
2. Set the parameters. The base estimator is a tree and you can set:
   - *Number of estimators*
   - *Learning rate*: it determines to what extent the newly acquired information will override the old information (0 = the agent will not learn anything, 1 = the agent considers only the most recent information)
   - *Fixed seed for random generator*: set a fixed seed to enable reproducing the results.
3. Boosting method.
   - *Classification algorithm* (if classification on input): SAMME (updates base estimator's weights with classification results) or SAMME.R (updates base estimator's weight with probability estimates).
   - *Regression loss function* (if regression on input): Linear (), Square (), Exponential ().
4. Produce a report.
5. 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 model and output it as well. To communicate changes automatically tick *Apply Automatically*.

Examples

For classification, we loaded the *iris* dataset. We used *AdaBoost*, Tree and Logistic Regression and evaluated the models' performance in Test & Score.



For regression, we loaded the *housing* dataset, sent the data instances to two different models (**AdaBoost** and Tree) and output them to the Predictions widget.



2.3.11 Neural Network

A multi-layer perceptron (MLP) algorithm with backpropagation.

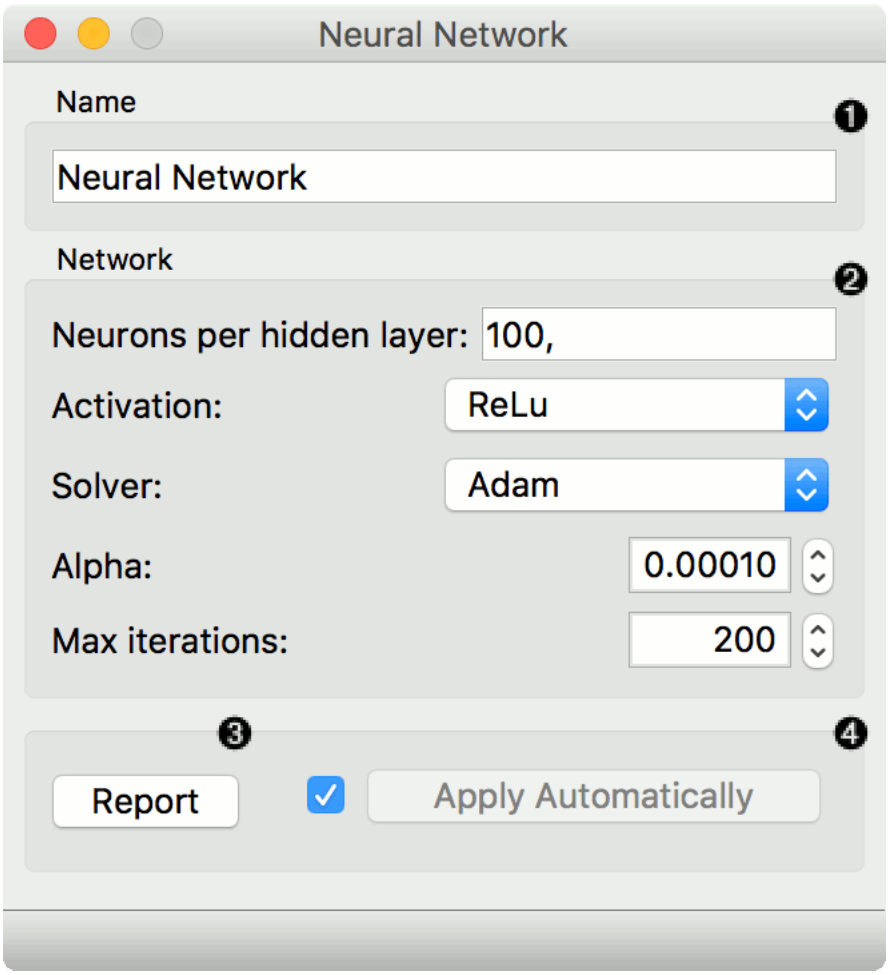
Inputs

- Data: input dataset
- Preprocessor: preprocessing method(s)

Outputs

- Learner: multi-layer perceptron learning algorithm
- Model: trained model

The **Neural Network** widget uses sklearn's Multi-layer Perceptron algorithm that can learn non-linear models as well as linear.



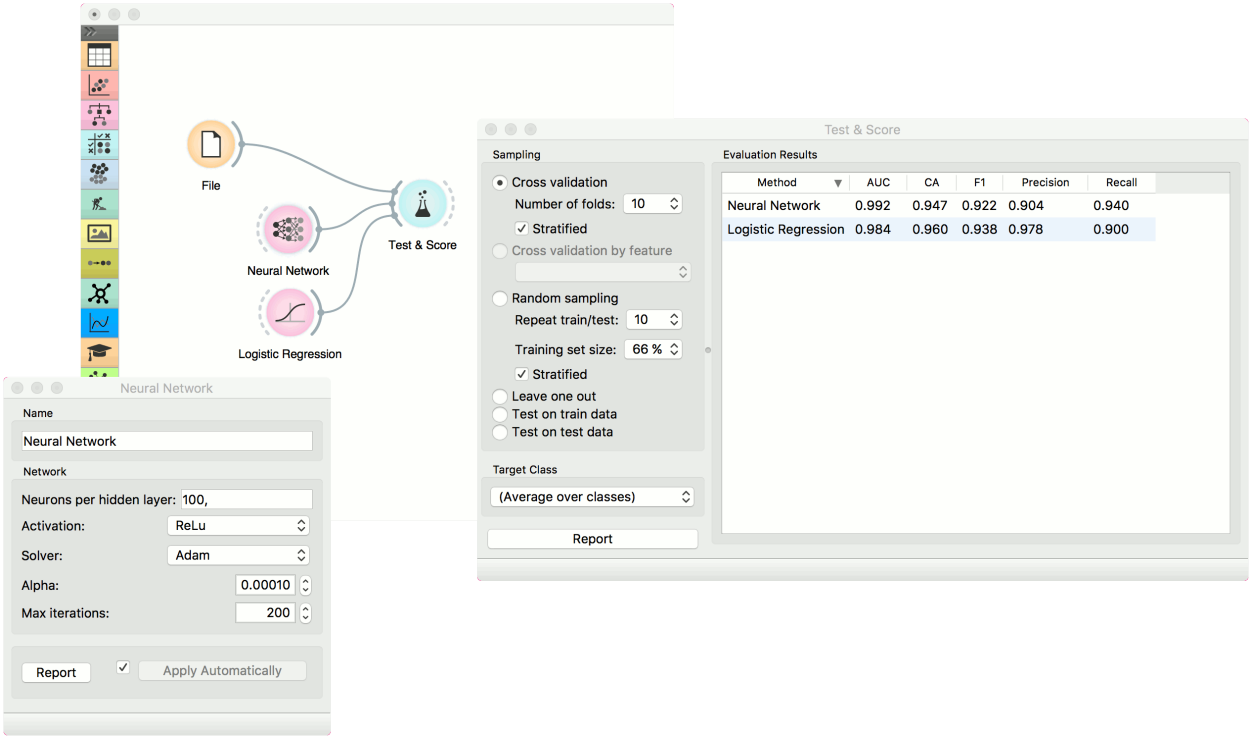
1. A name under which it will appear in other widgets. The default name is "Neural Network".
2. Set model parameters:
   - Neurons per hidden layer: defined as the ith element represents the number of neurons in the ith hidden layer. E.g. a neural network with 3 layers can be defined as 2, 3, 2.
   - Activation function for the hidden layer:
     - Identity: no-op activation, useful to implement linear bottleneck
     - Logistic: the logistic sigmoid function
     - tanh: the hyperbolic tan function
     - ReLu: the rectified linear unit function
   - Solver for weight optimization:
     - L-BFGS-B: an optimizer in the family of quasi-Newton methods
     - SGD: stochastic gradient descent
     - Adam: stochastic gradient-based optimizer
   - Alpha: L2 penalty (regularization term) parameter
   - Max iterations: maximum number of iterations

   Other parameters are set to sklearn's defaults.

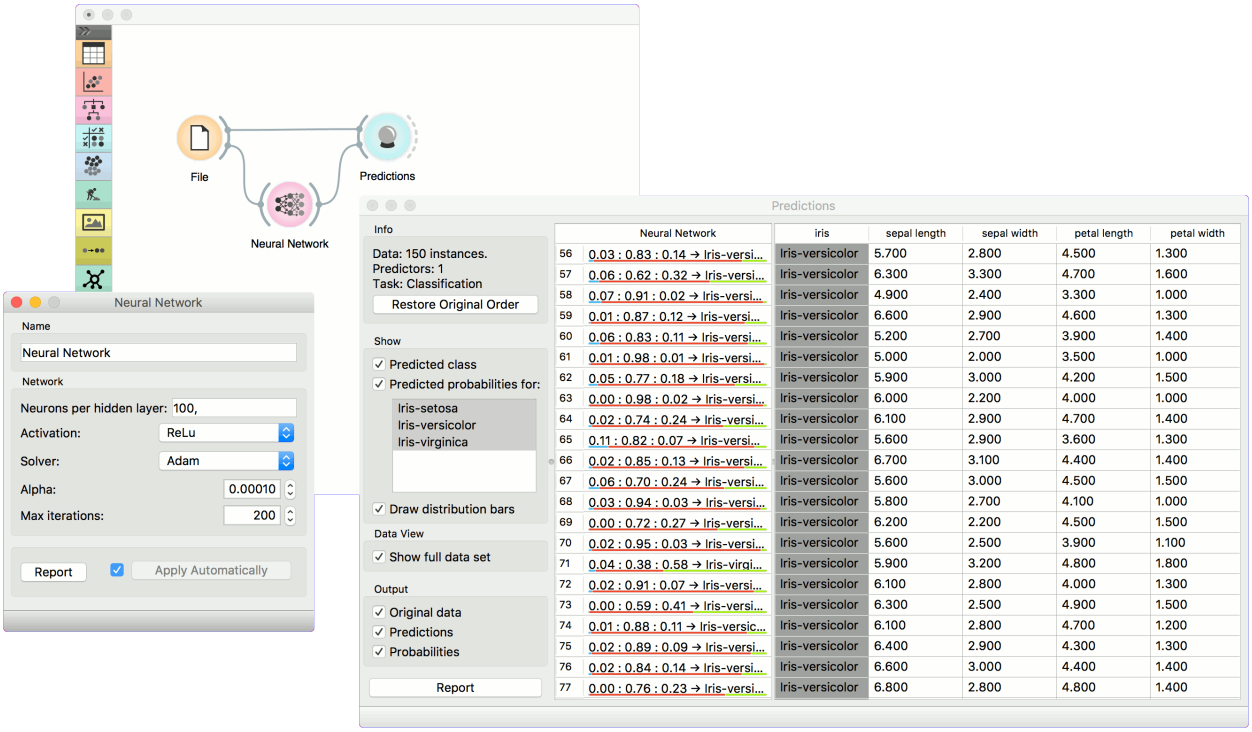
3. Produce a report.
4. When the box is ticked (*Apply Automatically*), the widget will communicate changes automatically. Alternatively, click *Apply*.

Examples

The first example is a classification task on *iris* dataset. We compare the results of **Neural Network** with the Logistic Regression.



The second example is a prediction task, still using the *iris* data. This workflow shows how to use the *Learner* output. We input the **Neural Network** prediction model into Predictions and observe the predicted values.



2.3.12 Stochastic Gradient Descent

Minimize an objective function using a stochastic approximation of gradient descent.

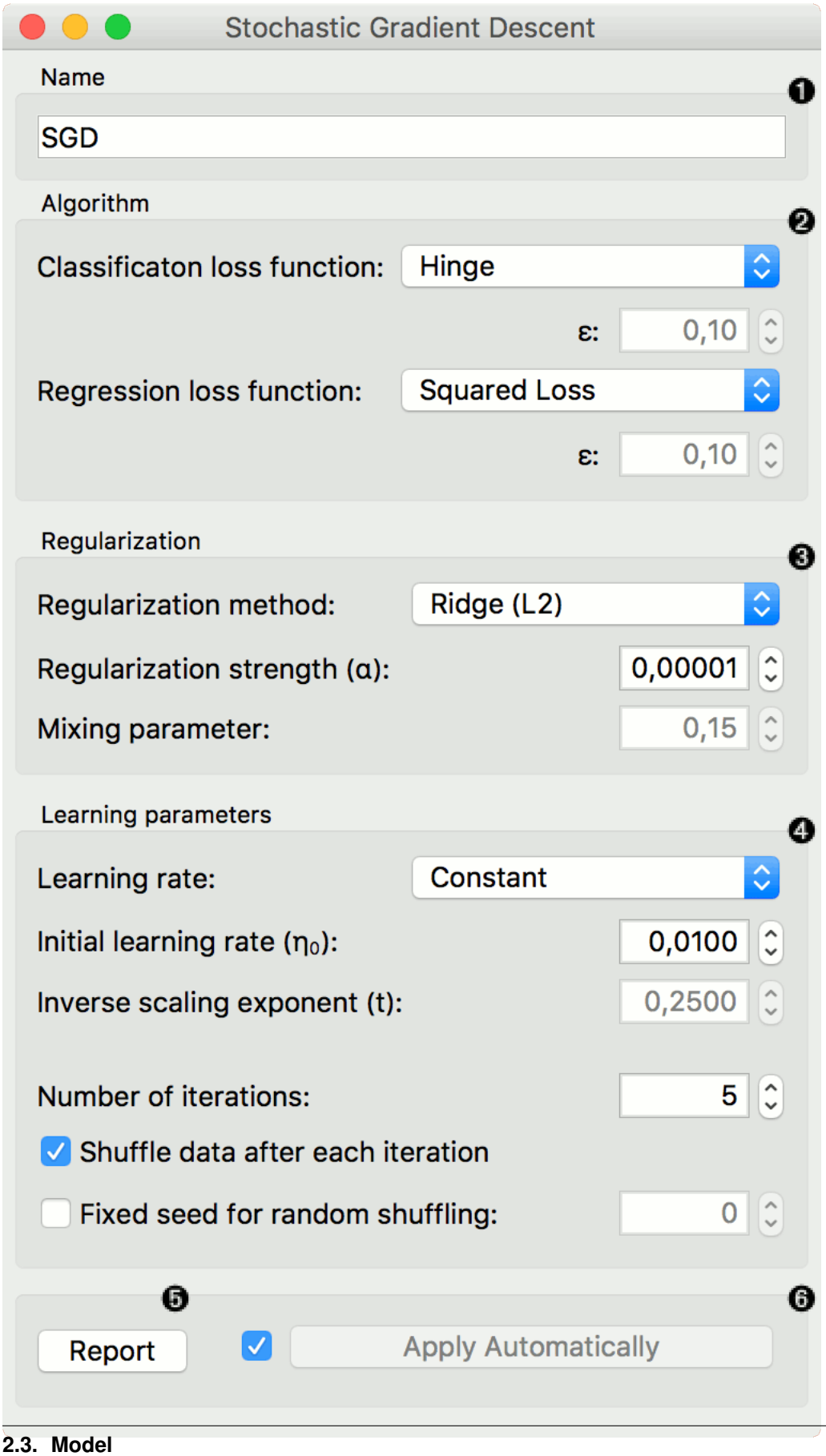
Inputs

- Data: input dataset
- Preprocessor: preprocessing method(s)

Outputs

- Learner: stochastic gradient descent learning algorithm
- Model: trained model

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. For regression, it returns predictors as minimizers of the sum, i.e. M-estimators, and is especially useful for large-scale and sparse datasets.


Stochastic Gradient Descent
Name
1
SGD
Algorithm
2
Classificaton loss function:
Hinge
ε:
0,10
Regression loss function:
Squared Loss
ε:
0,10
Regularization
3
Regularization method:
Ridge (L2)
Regularization strength (α):
0,00001
Mixing parameter:
0,15
Learning parameters
4
Learning rate:
Constant
Initial learning rate (η0):
0,0100
Inverse scaling exponent (t):
0,2500
Number of iterations:
5
Shuffle data after each iteration
Fixed seed for random shuffling:
0
5
6
Report
Apply Automatically

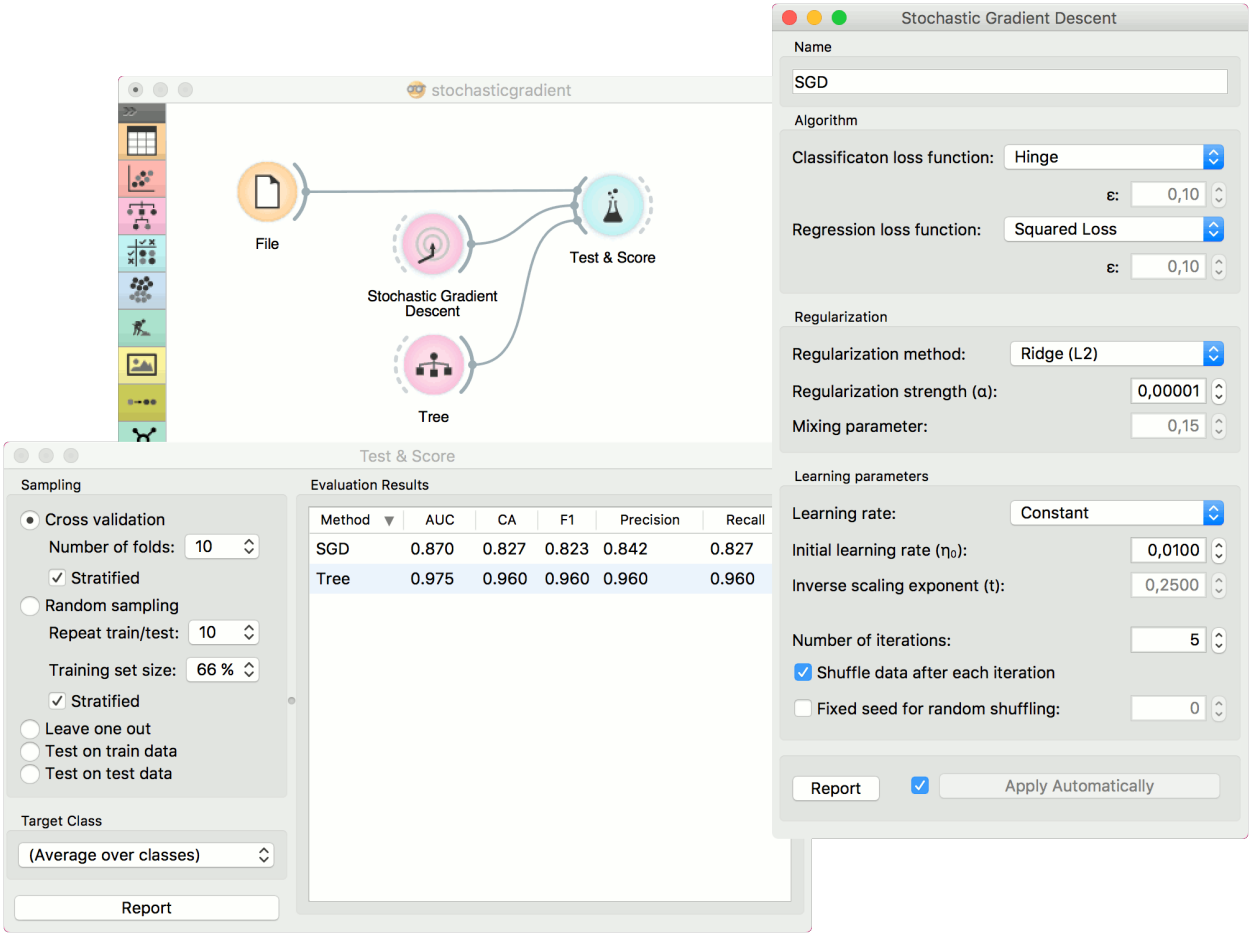
1. Specify the name of the model. The default name is "SGD".
2. Algorithm parameters:
   - Classification loss function:
     - Hinge (linear SVM)
     - Logistic Regression (logistic regression SGD)
     - Modified Huber (smooth loss that brings tolerance to outliers as well as probability estimates)
     - *Squared Hinge* (quadratically penalized hinge)
     - Perceptron (linear loss used by the perceptron algorithm)
     - Squared Loss (fitted to ordinary least-squares)
     - Huber (switches to linear loss beyond ϵ)
     - Epsilon insensitive (ignores errors within ϵ, linear beyond it)
     - *Squared epsilon insensitive* (loss is squared beyond ϵ-region).
   - Regression loss function:
     - Squared Loss (fitted to ordinary least-squares)
     - Huber (switches to linear loss beyond ϵ)
     - Epsilon insensitive (ignores errors within ϵ, linear beyond it)
     - *Squared epsilon insensitive* (loss is squared beyond ϵ-region).
3. Regularization norms to prevent overfitting:
   - None.
   - Lasso (L1) (L1 leading to sparse solutions)
   - Ridge (L2) (L2, standard regularizer)
   - Elastic net (mixing both penalty norms).

   Regularization strength defines how much regularization will be applied (the less we regularize, the more we allow the model to fit the data) and the mixing parameter what the ratio between L1 and L2 loss will be (if set to 0 then the loss is L2, if set to 1 then it is L1).
4. Learning parameters.
   - Learning rate:
     - *Constant*: learning rate stays the same through all epochs (passes)
     - Optimal: a heuristic proposed by Leon Bottou
     - Inverse scaling: earning rate is inversely related to the number of iterations
   - Initial learning rate.
   - Inverse scaling exponent: learning rate decay.
   - Number of iterations: the number of passes through the training data.
   - If *Shuffle data after each iteration* is on, the order of data instances is mixed after each pass.
   - If *Fixed seed for random shuffling* is on, the algorithm will use a fixed random seed and enable replicating the results.
5. Produce a report.

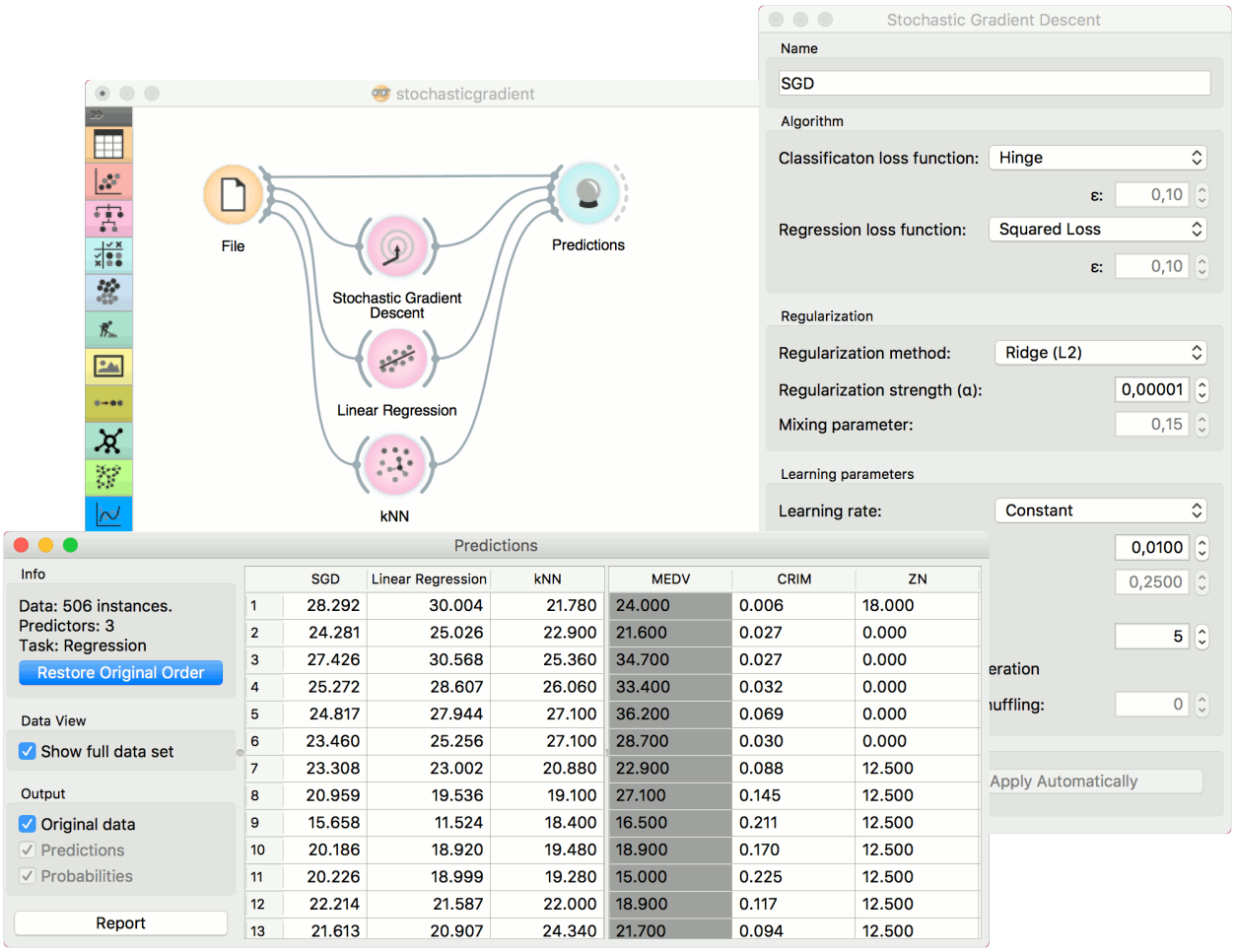
6. Press *Apply* to commit changes. Alternatively, tick the box on the left side of the *Apply* button and changes will be communicated automatically.

Examples

For the classification task, we will use *iris* dataset and test two models on it. We connected Stochastic Gradient Descent and Tree to Test & Score. We also connected File to **Test & Score** and observed model performance in the widget.



For the regression task, we will compare three different models to see which predict what kind of results. For the purpose of this example, the *housing* dataset is used. We connect the File widget to **Stochastic Gradient Descent**, Linear Regression and kNN widget and all four to the Predictions widget.



	SGD	Linear Regression	kNN	MEDV	CRIM	ZN
1	28.292	30.004	21.780	24.000	0.006	18.000
2	24.281	25.026	22.900	21.600	0.027	0.000
3	27.426	30.568	25.360	34.700	0.027	0.000
4	25.272	28.607	26.060	33.400	0.032	0.000
5	24.817	27.944	27.100	36.200	0.069	0.000
6	23.460	25.256	27.100	28.700	0.030	0.000
7	23.308	23.002	20.880	22.900	0.088	12.500
8	20.959	19.536	19.100	27.100	0.145	12.500
9	15.658	11.524	18.400	16.500	0.211	12.500
10	20.186	18.920	19.480	18.900	0.170	12.500
11	20.226	18.999	19.280	15.000	0.225	12.500
12	22.214	21.587	22.000	18.900	0.117	12.500
13	21.613	20.907	24.340	21.700	0.094	12.500

2.3.13 Stacking

Stack multiple models.

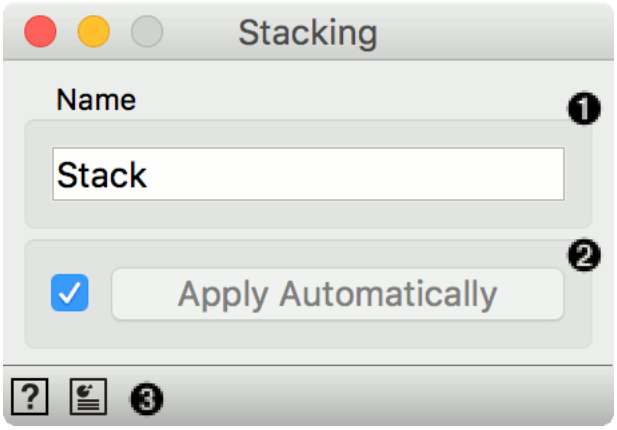
Inputs

- Data: input dataset
- Preprocessor: preprocessing method(s)
- Learners: learning algorithm
- Aggregate: model aggregation method

Outputs

- Learner: aggregated (stacked) learning algorithm
- Model: trained model

Stacking is an ensemble method that computes a meta model from several base models. The **Stacking** widget has the **Aggregate** input, which provides a method for aggregating the input models. If no aggregation input is given the default methods are used. Those are **Logistic Regression** for classification and **Ridge Regression** for regression problems.

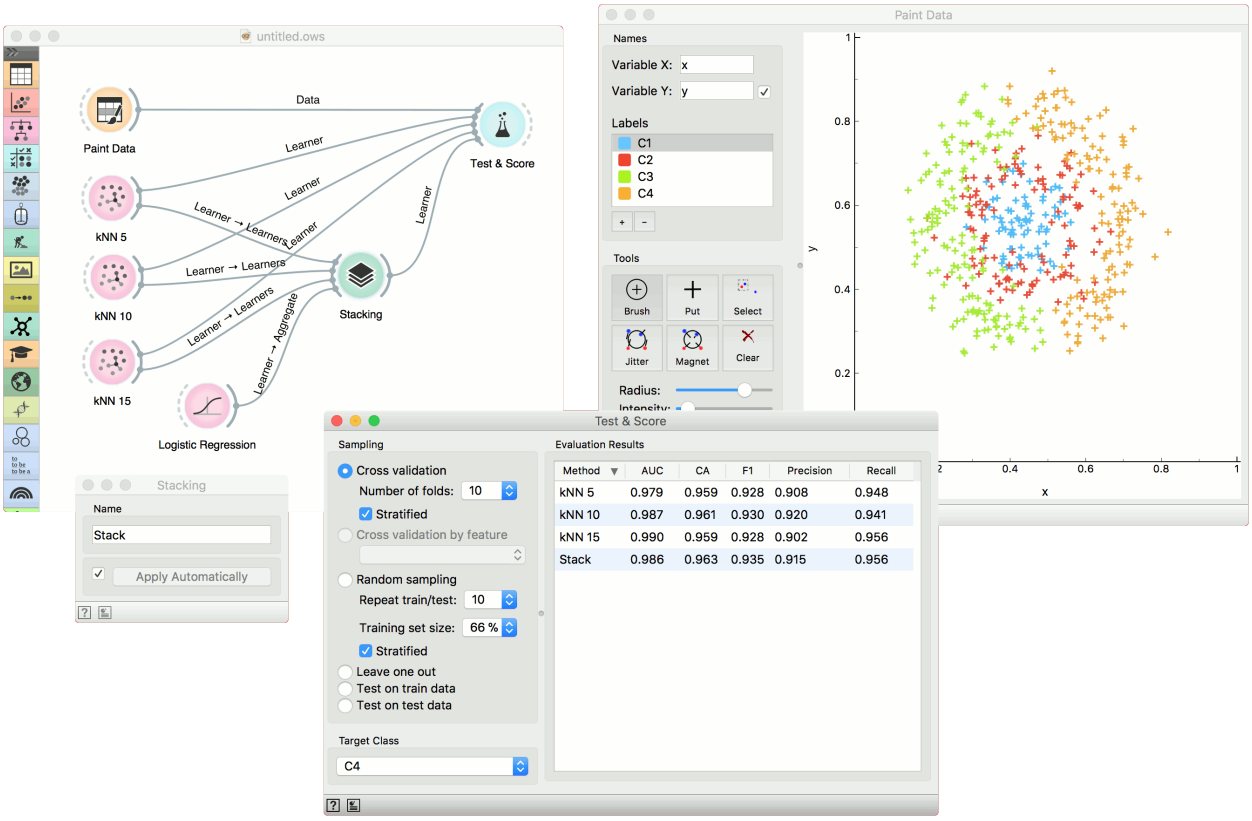


1. The meta learner can be given a name under which it will appear in other widgets. The default name is "Stack".
2. Click *Apply* to commit the aggregated model. That will put the new learner in the output and, if the training examples are given, construct a new model and output it as well. To communicate changes automatically tick *Apply Automatically*.
3. Access help and produce a report.

Example

We will use Paint Data to demonstrate how the widget is used. We painted a complex dataset with 4 class labels and sent it to Test & Score. We also provided three kNN learners, each with a different parameters (number of neighbors is 5, 10 or 15). Evaluation results are good, but can we do better?

Let's use **Stacking**. **Stacking** requires several learners on the input and an aggregation method. In our case, this is Logistic Regression. A constructed meta learner is then sent to **Test & Score**. Results have improved, even if only marginally. **Stacking** normally works well on complex data sets.

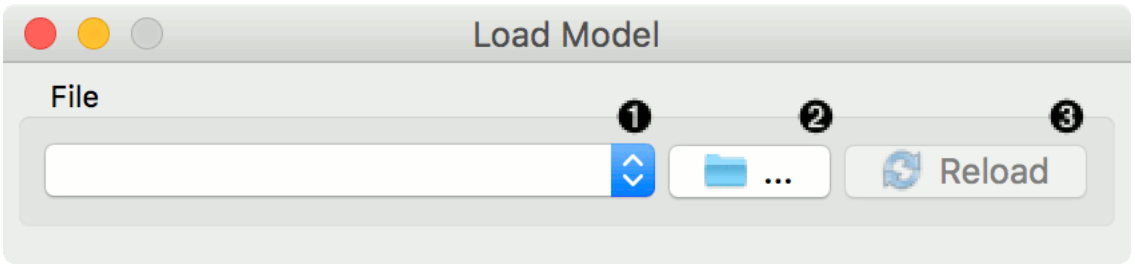


2.3.14 Load Model

Load a model from an input file.

Outputs

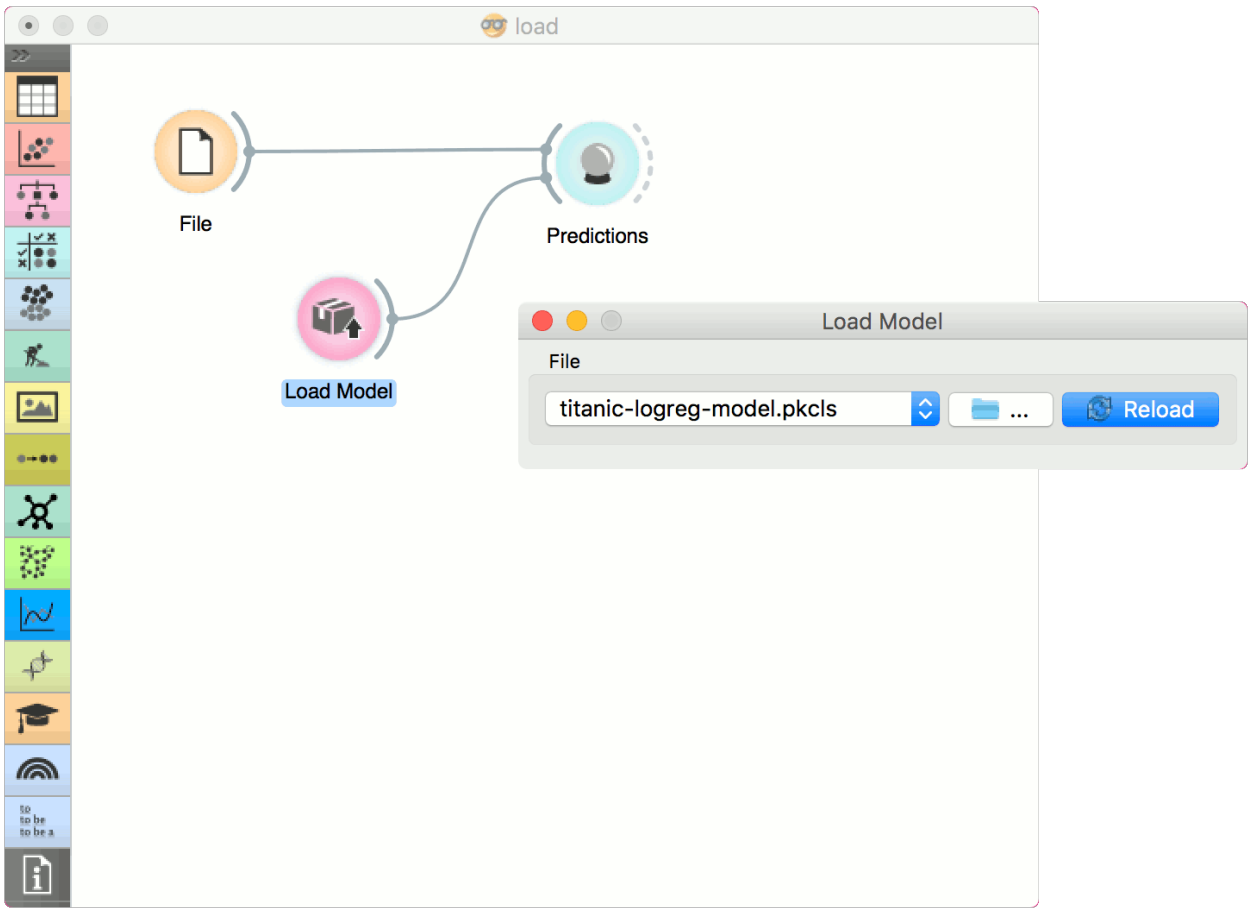
- Model: trained model



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

Example

When you want to use a custom-set model that you've saved before, open the **Load Model** widget and select the desired file with the *Browse* icon. This widget loads the existing model into Predictions widget. Datasets used with **Load Model** have to contain compatible attributes!

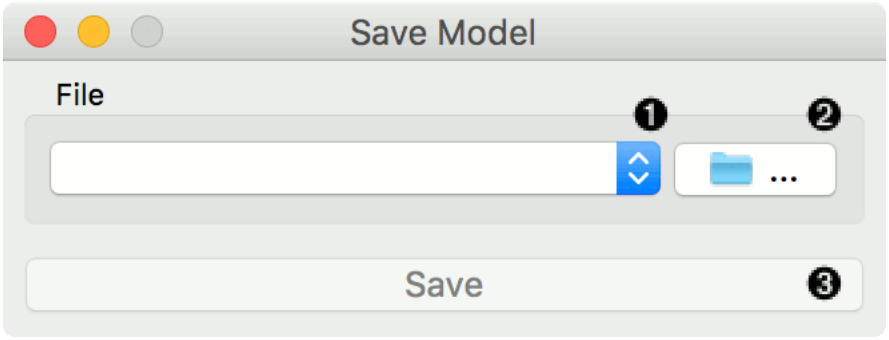


2.3.15 Save Model

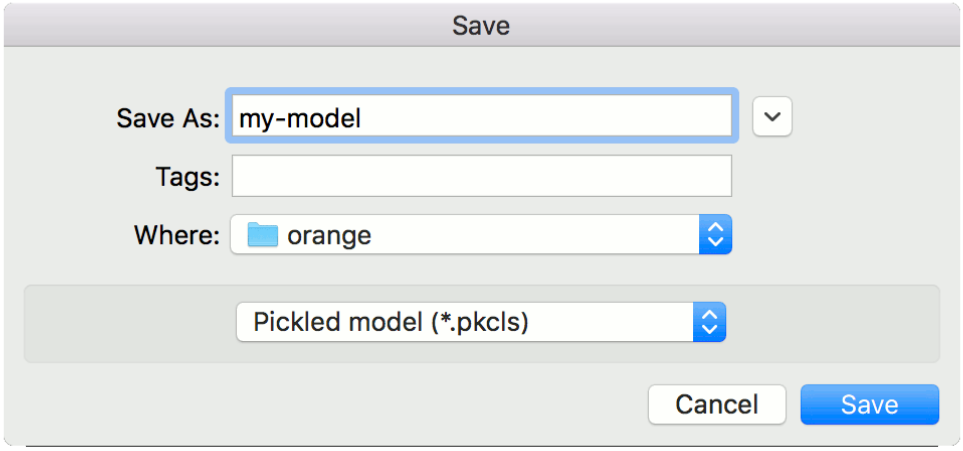
Save a trained model to an output file.

Inputs

- Model: trained model



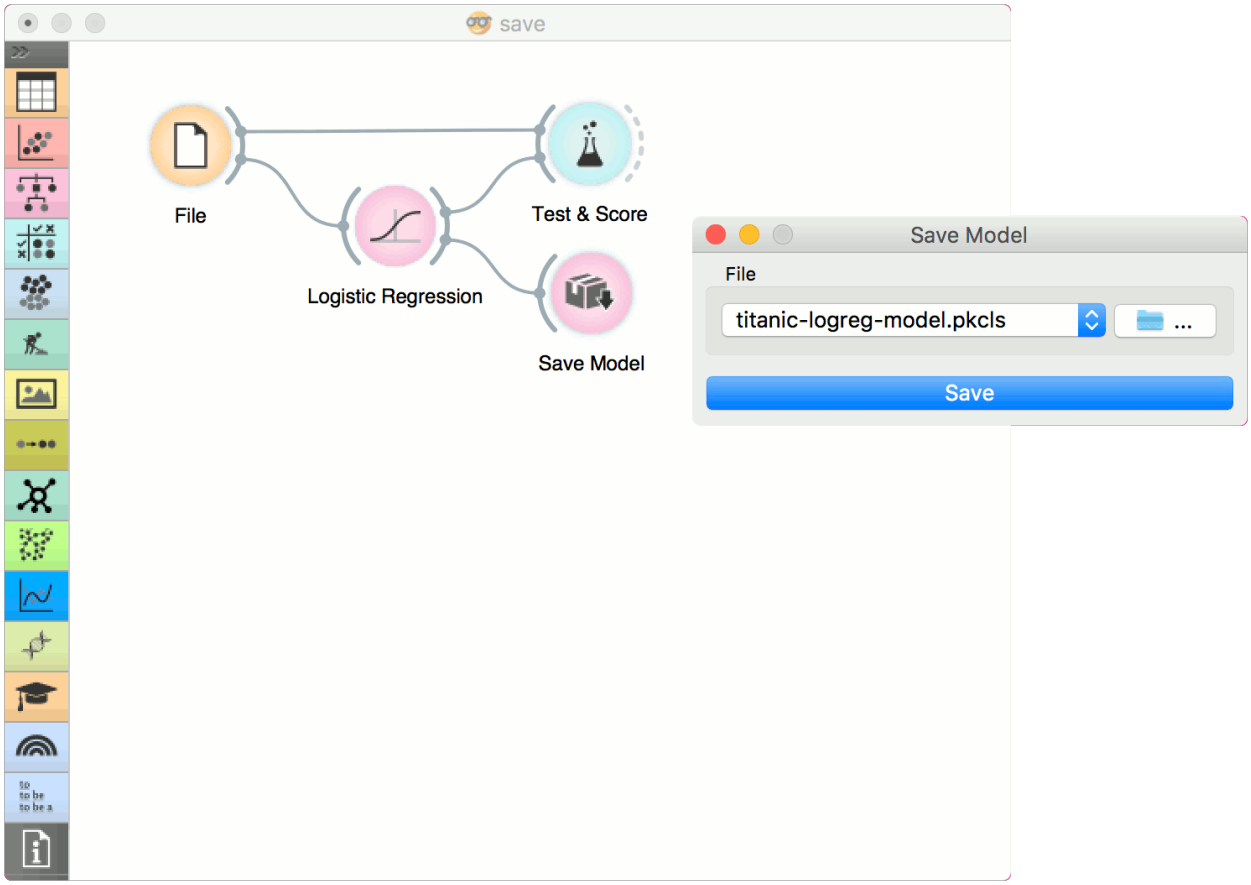
1. Choose from previously saved models.
2. Save the created model with the *Browse* icon. Click on the icon and enter the name of the file. The model will be saved to a pickled file.



3. Save the model.

Example

When you want to save a custom-set model, feed the data to the model (e.g. Logistic Regression) and connect it to **Save Model**. Name the model; load it later into workflows with Load Model. Datasets used with **Load Model** have to contain compatible attributes.



2.4 Unsupervised

2.4.1 PCA

PCA linear transformation of input data.

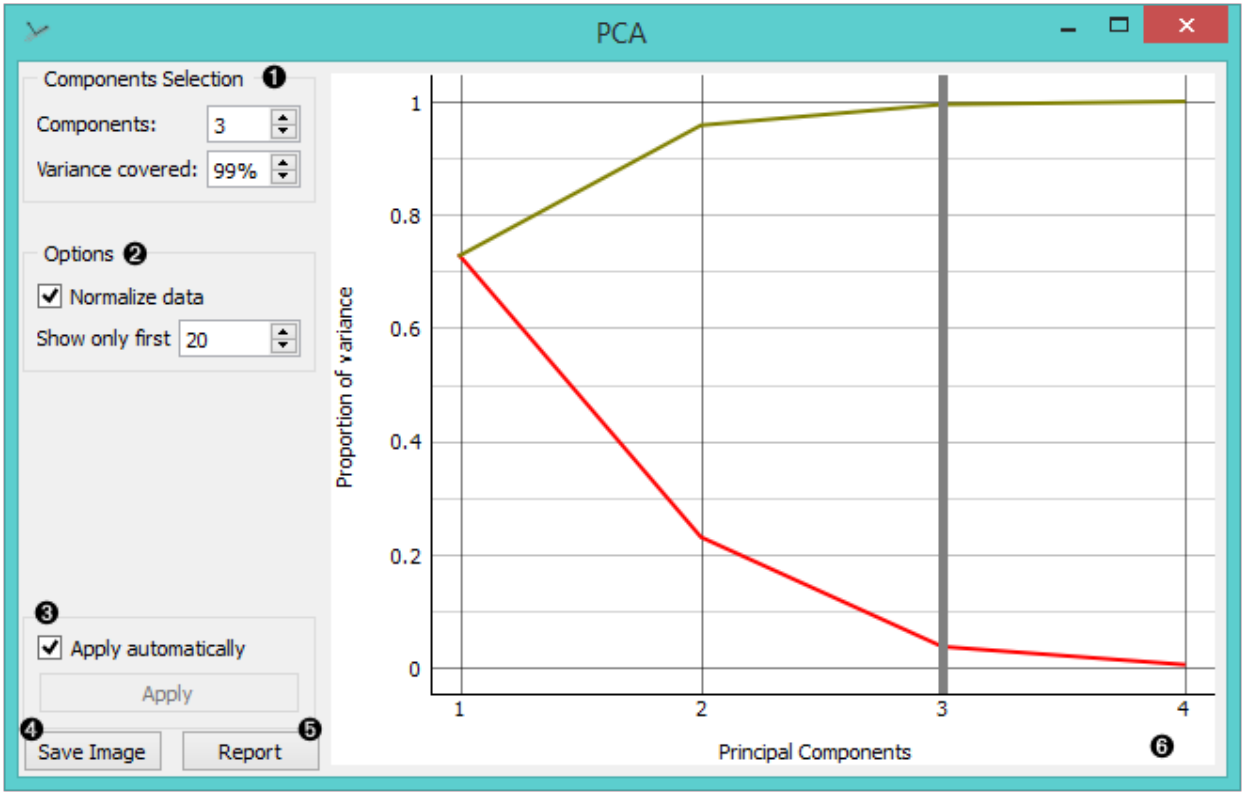
Inputs

- Data: input dataset

Outputs

- Transformed Data: PCA transformed data
- Components: Eigenvectors.

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

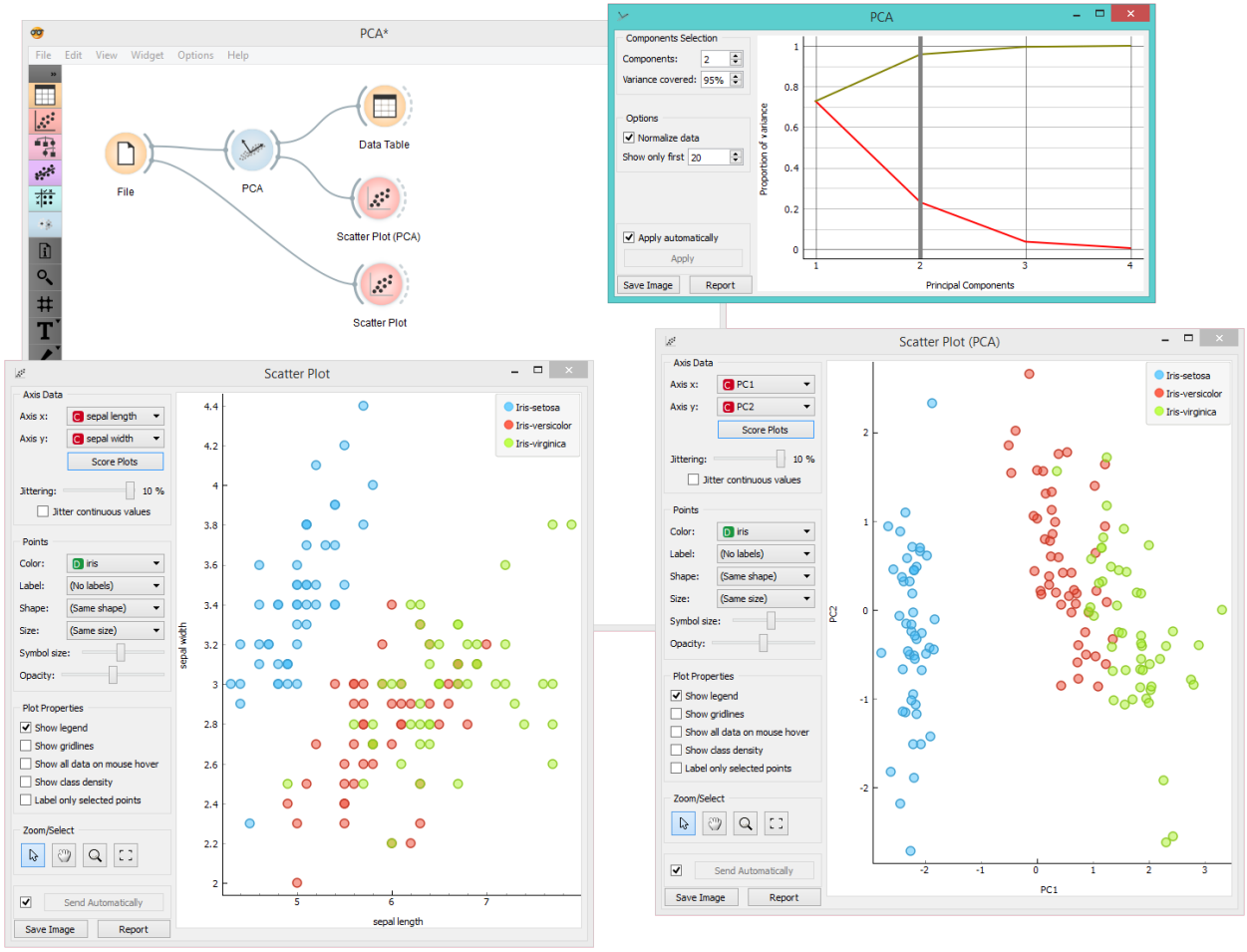


1. Select how many principal components you wish in your output. It is best to choose as few as possible with variance covered as high as possible. You can also set how much variance you wish to cover with your principal components.
2. You can normalize data to adjust the values to common scale.
3. When *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.
6. 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.

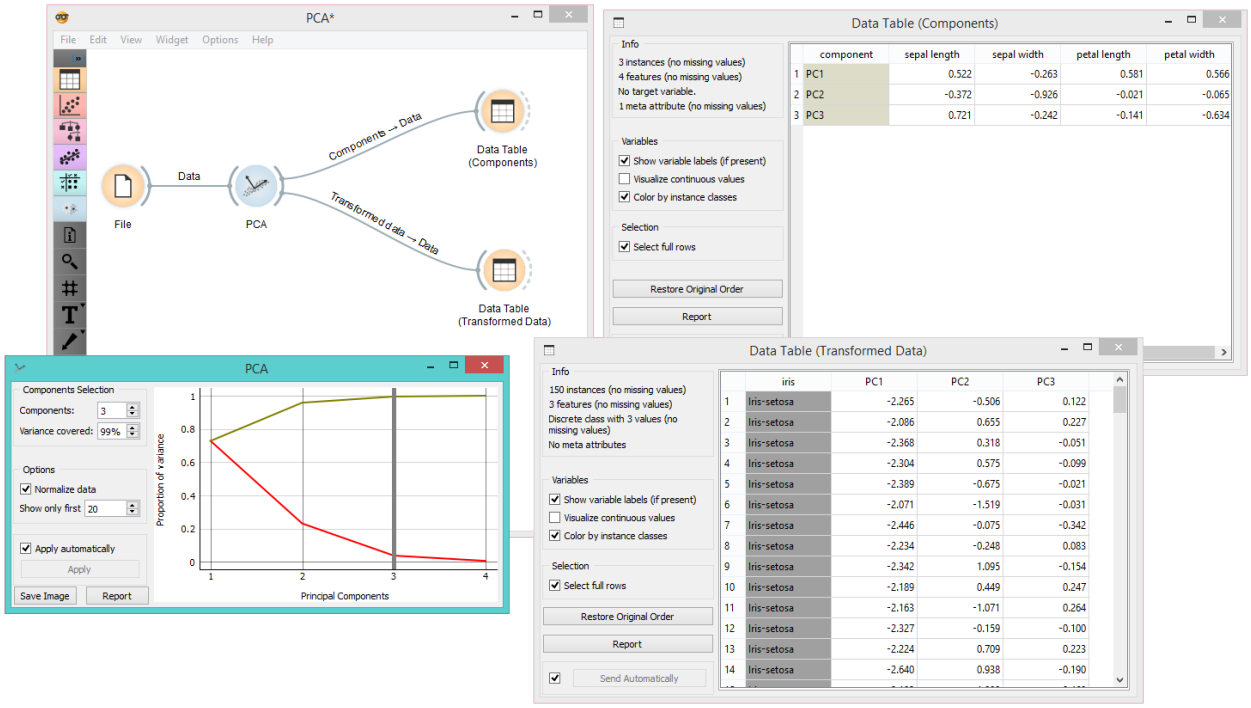
The number of components of the transformation can be selected either in the *Components Selection* input box or by dragging the vertical cutoff line in the graph.

Examples

PCA can be used to simplify visualizations of large datasets. Below, we used the *Iris* dataset to show how we can improve the visualization of the dataset 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.



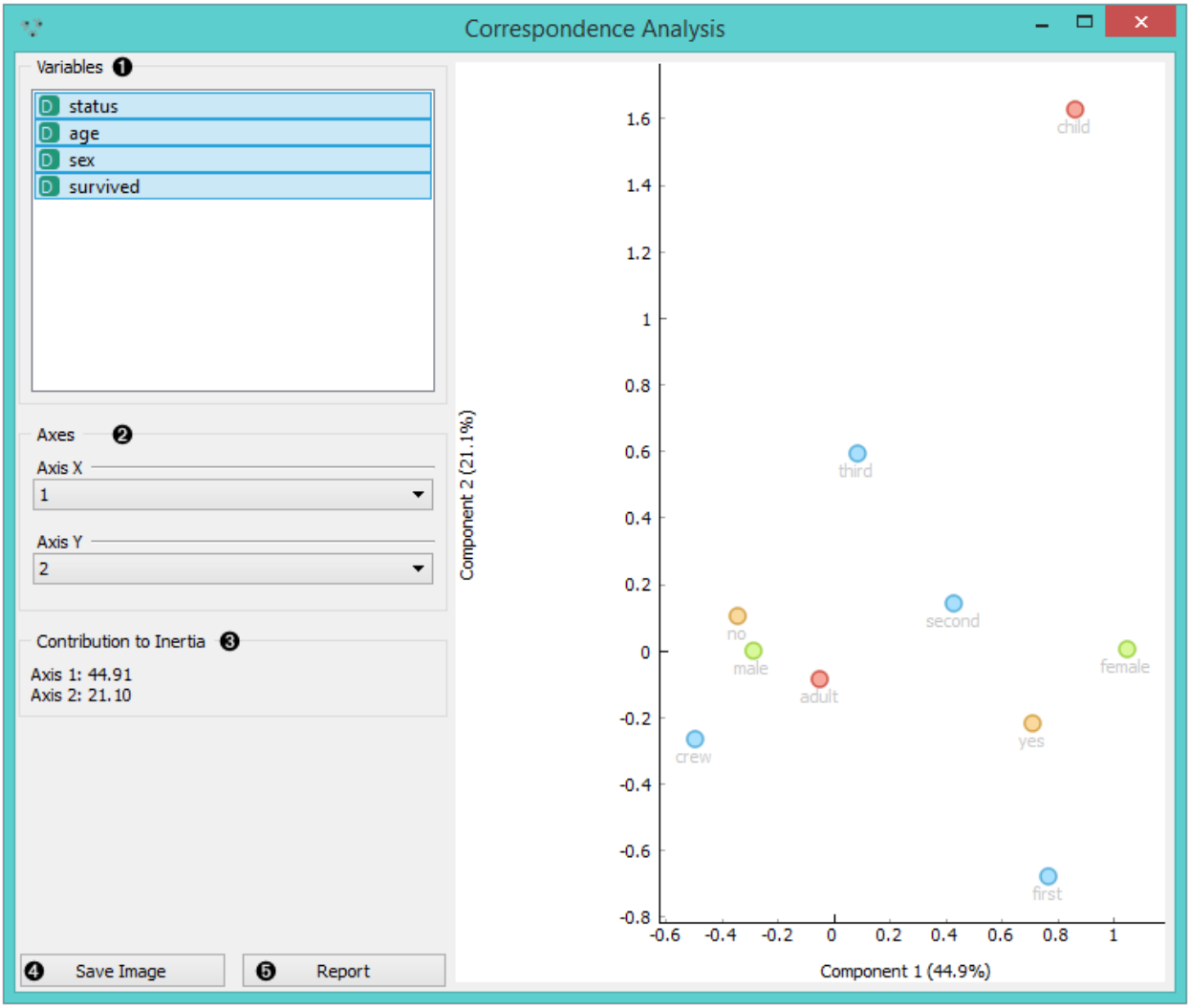
2.4.2 Correspondence Analysis

Correspondence analysis for categorical multivariate data.

Inputs

- Data: input dataset

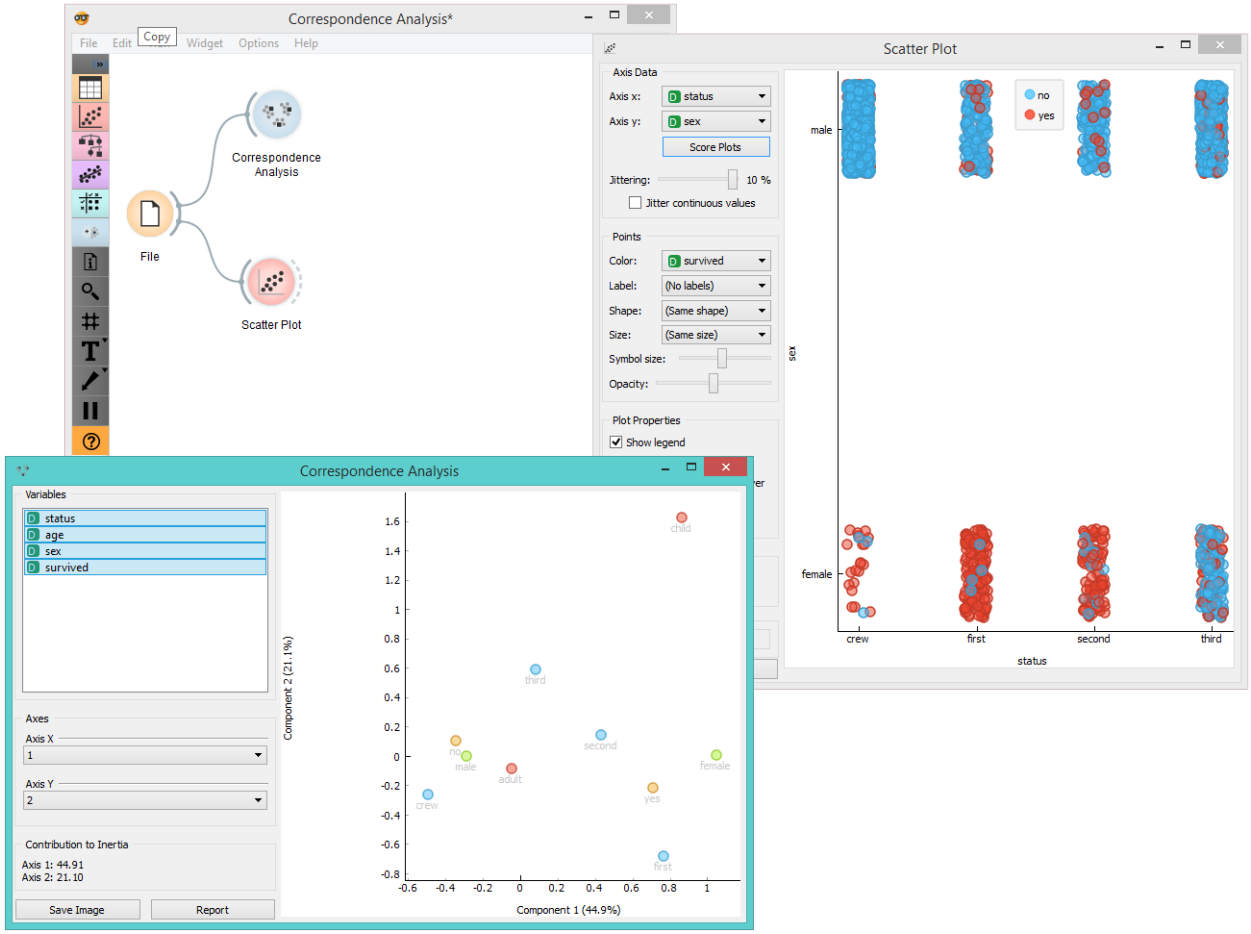
Correspondence Analysis (CA) computes the CA linear transformation of the input data. While it is similar to PCA, CA computes linear transformation on discrete rather than on continuous data.



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* dataset. 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".



2.4.3 Distance Map

Visualizes distances between items.

Inputs

- Distances: distance matrix

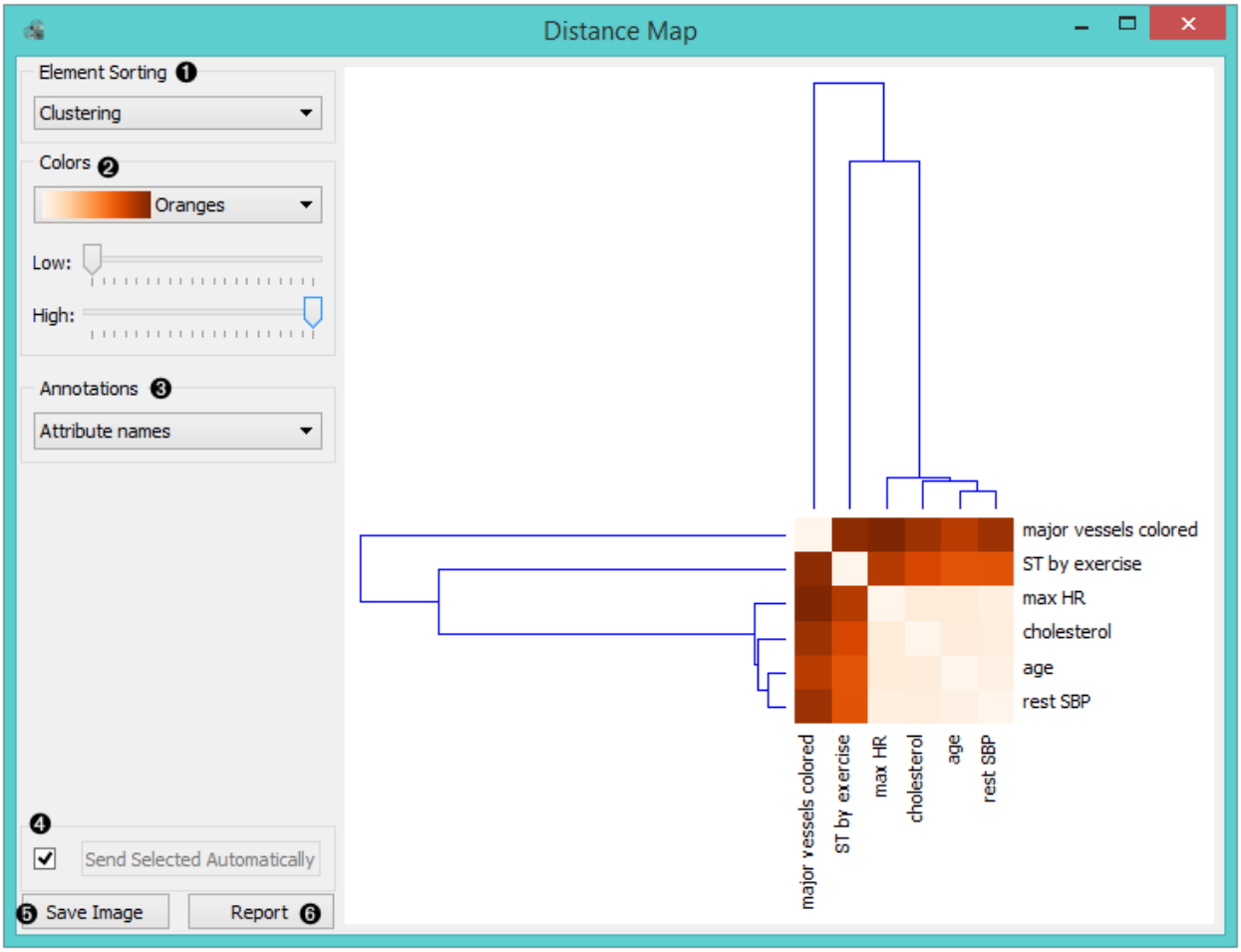
Outputs

- Data: instances selected from the matrix
- Features: attributes selected from the matrix

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 Continuize them first.

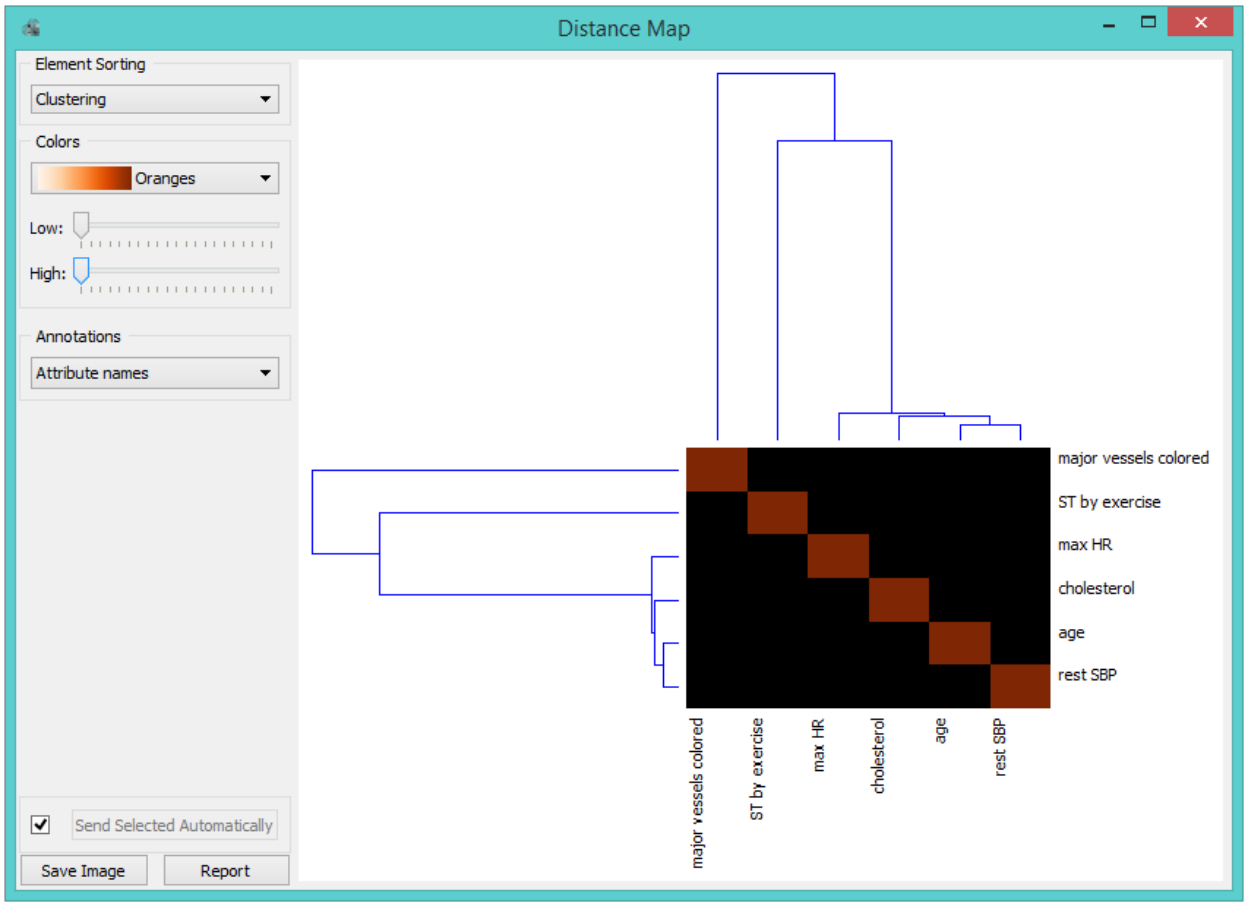
The snapshot shows distances between columns in the *heart disease* data, where smaller distances are represented with light and larger with dark orange. The matrix is symmetric and the diagonal is a light shade of orange - no attribute is different from itself. Symmetricity is always assumed, while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



1. *Element sorting* arranges elements in the map by
   - None (lists instances as found in the dataset)
   - **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.

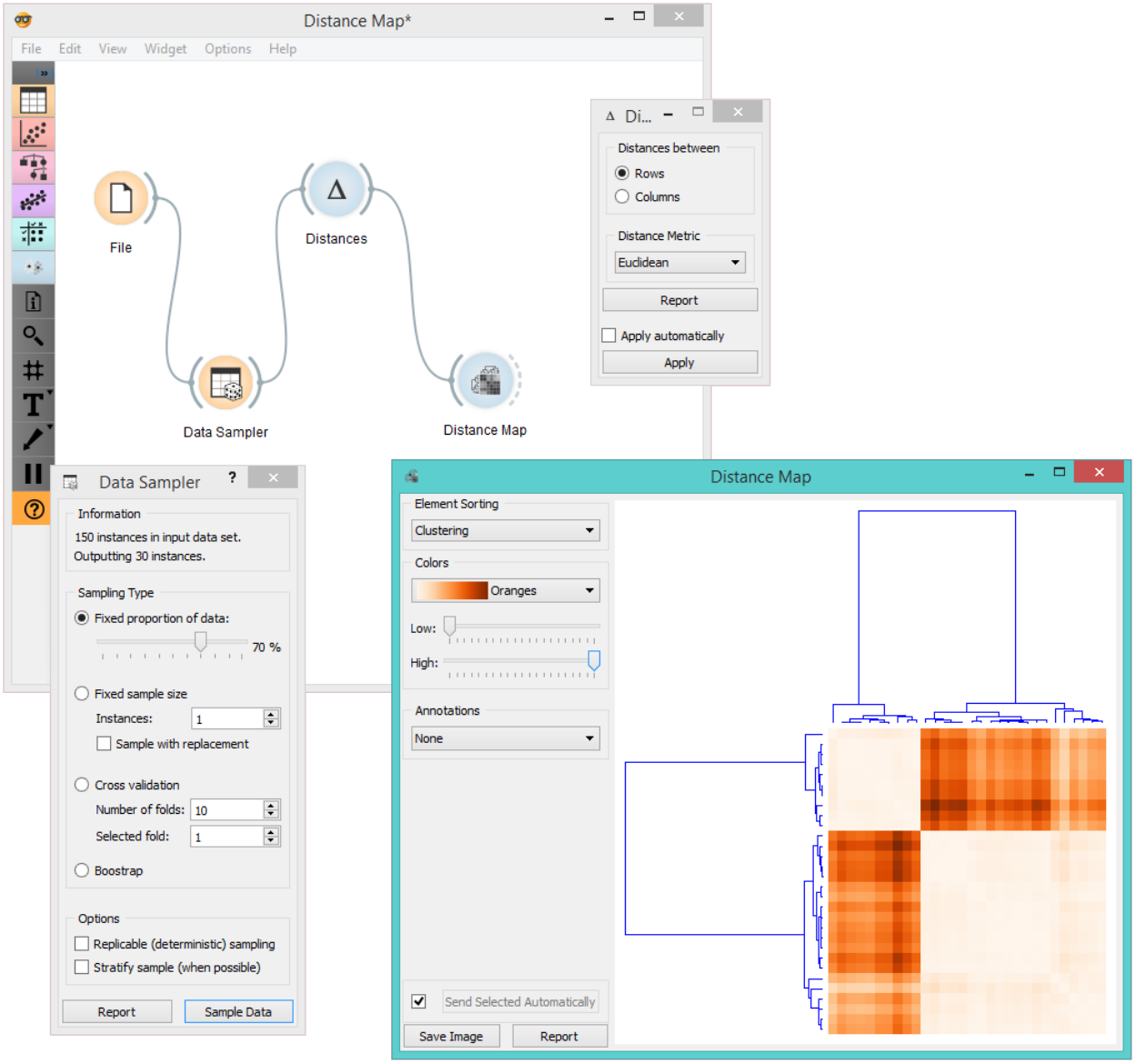
Below, we visualized the most correlated attributes (distances by columns) in the *heart disease* dataset by setting the color threshold for high distances to the minimum. We get a predominantly black square, where attributes with the lowest distance scores are represented by a lighter shade of the selected 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.



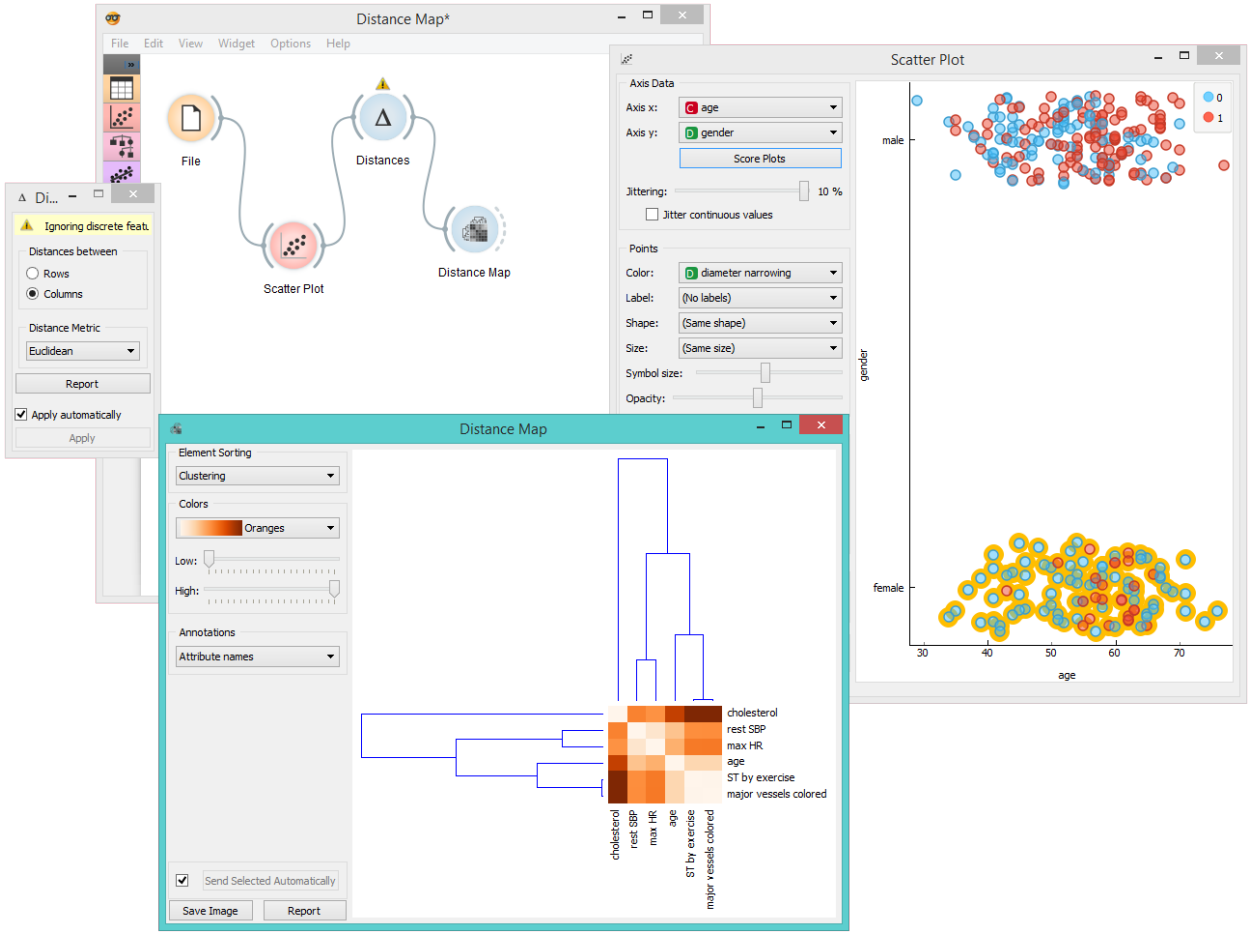
The user can select a region in the map with the usual click-and-drag of the cursor. When a part of the map is selected, the widget outputs all items from the selected cells.

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.



2.4.4 Distances

Computes distances between rows/columns in a dataset.

Inputs

- Data: input dataset

Outputs

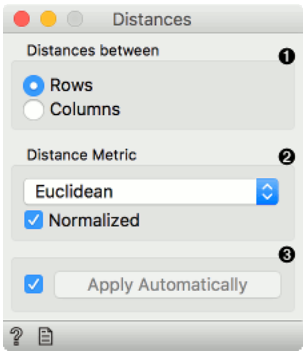
- Distances: distance matrix

The **Distances** widget computes distances between rows or columns in a dataset. By default, the data will be normalized to ensure equal treatment of individual features. Normalization is always done column-wise.

Sparse data can only be used with Euclidean, Manhattan and Cosine metric.

The resulting distance matrix can be fed further to Hierarchical Clustering for uncovering groups in the data, to Distance Map or Distance Matrix for visualizing the distances (Distance Matrix can be quite slow for larger data sets), to MDS for mapping the data instances using the distance matrix and finally, saved with Save Distance Matrix. Distance file can be loaded with Distance File.

Distances work well with Orange add-ons, too. The distance matrix can be fed to Network from Distances (Network add-on) to convert the matrix into a graph and to Duplicate Detection (Text add-on) to find duplicate documents in the corpus.

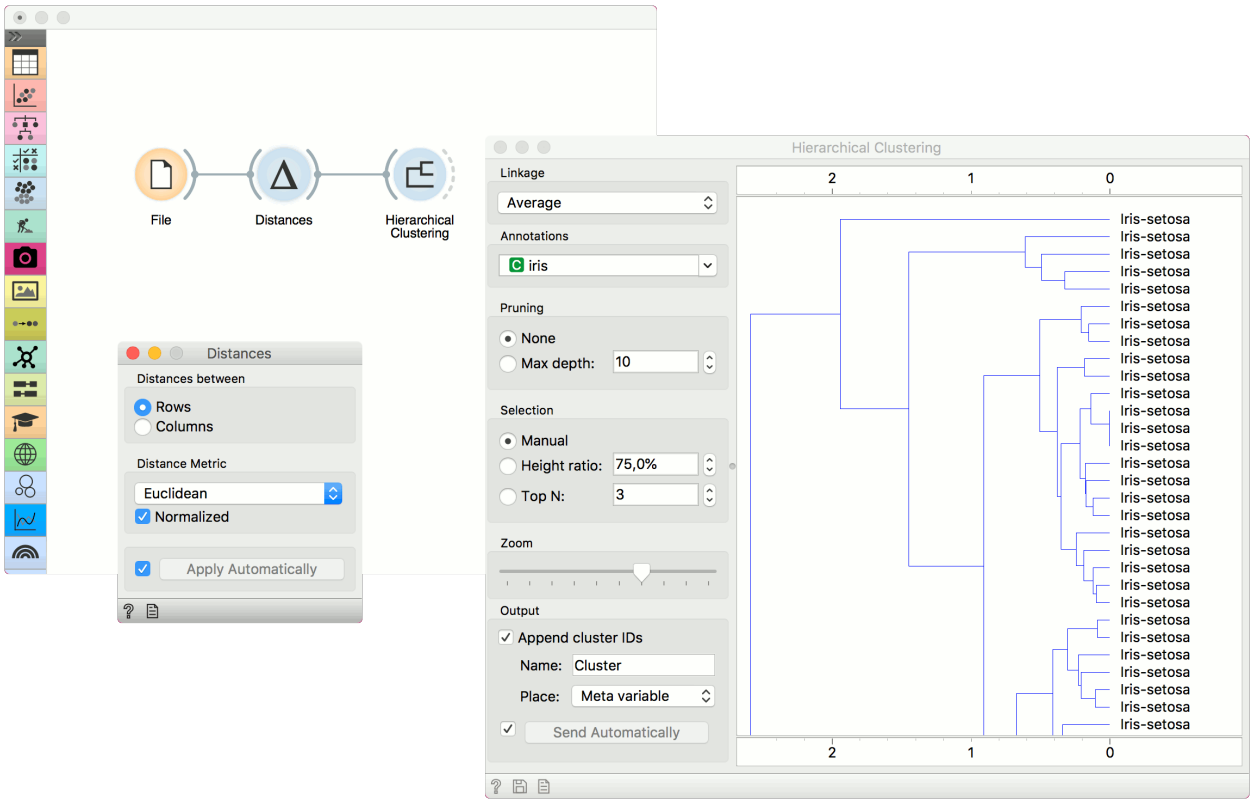


1. Choose whether to measure distances between rows or columns.
2. Choose the *Distance Metric*:
   - Euclidean ("straight line", distance between two points)
   - Manhattan (the sum of absolute differences for all attributes)
   - Cosine (the cosine of the angle between two vectors of an inner product 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)
   - Hamming (the number of features at which the corresponding values are different)

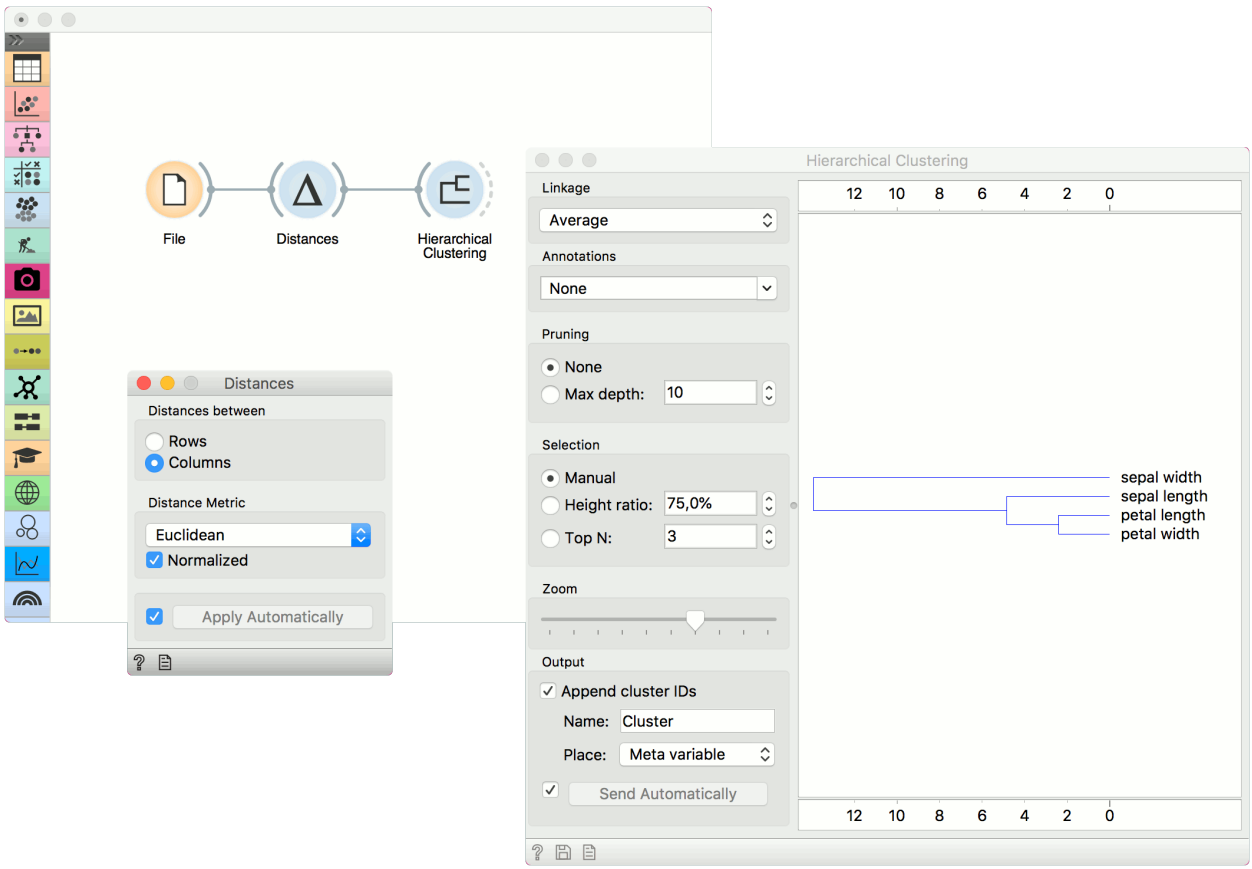
   Normalize the features. Normalization is always done column-wise. In case of missing values, the widget automatically imputes the average value of the row or the column. The widget works for both numeric and categorical data. In case of categorical data, the distance is 0 if the two values are the same ('green' and 'green') and 1 if they are not ('green' and 'blue').
3. Tick *Apply Automatically* to automatically commit changes to other widgets. Alternatively, press '*Apply*'.

Examples

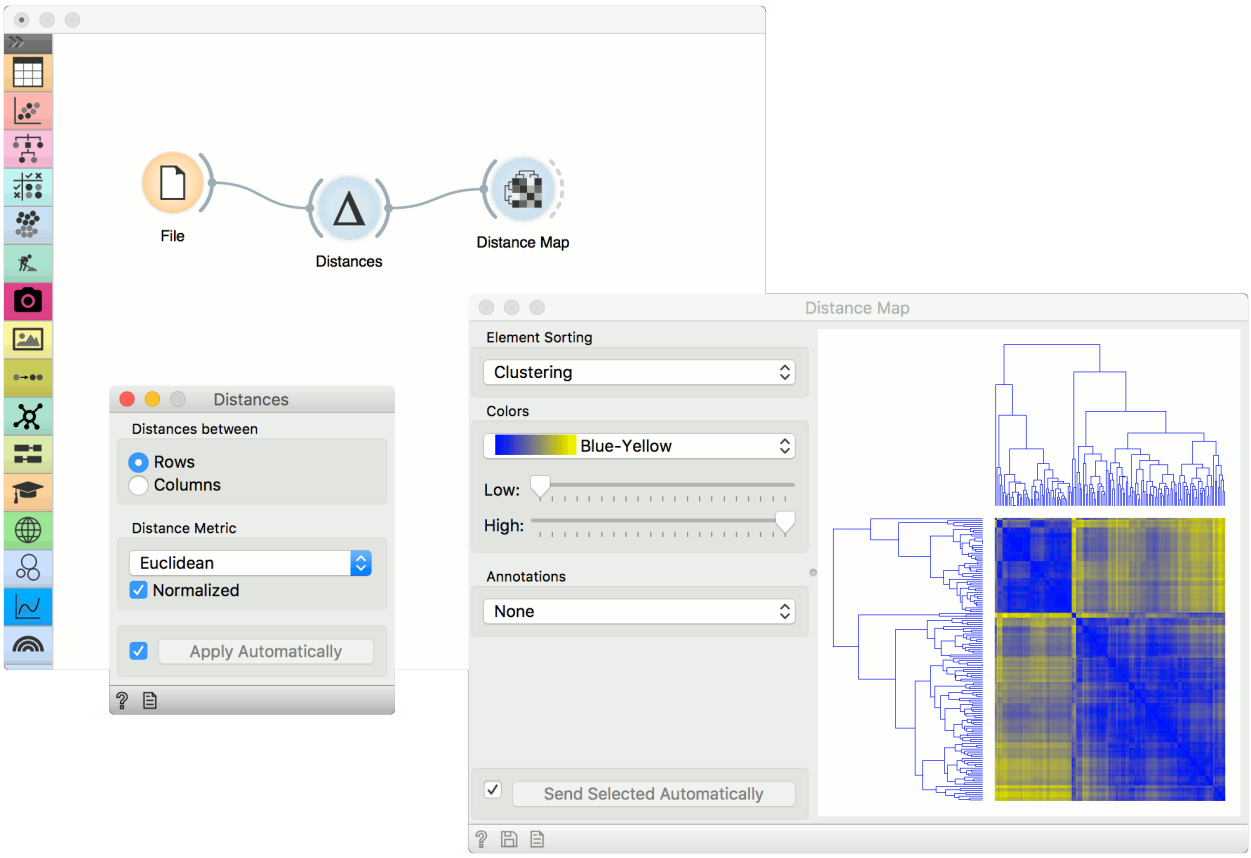
The first example shows a typical use of the **Distances** widget. We are using the *iris.tab* data from the File widget. We compute distances between data instances (rows) and pass the result to the Hierarchical Clustering. This is a simple workflow to find groups of data instances.



Alternatively, we can compute distance between columns and find how similar our features are.



The second example shows how to visualize the resulting distance matrix. A nice way to observe data similarity is in a Distance Map or in MDS.



2.4.5 Distance Matrix

Visualizes distance measures in a distance matrix.

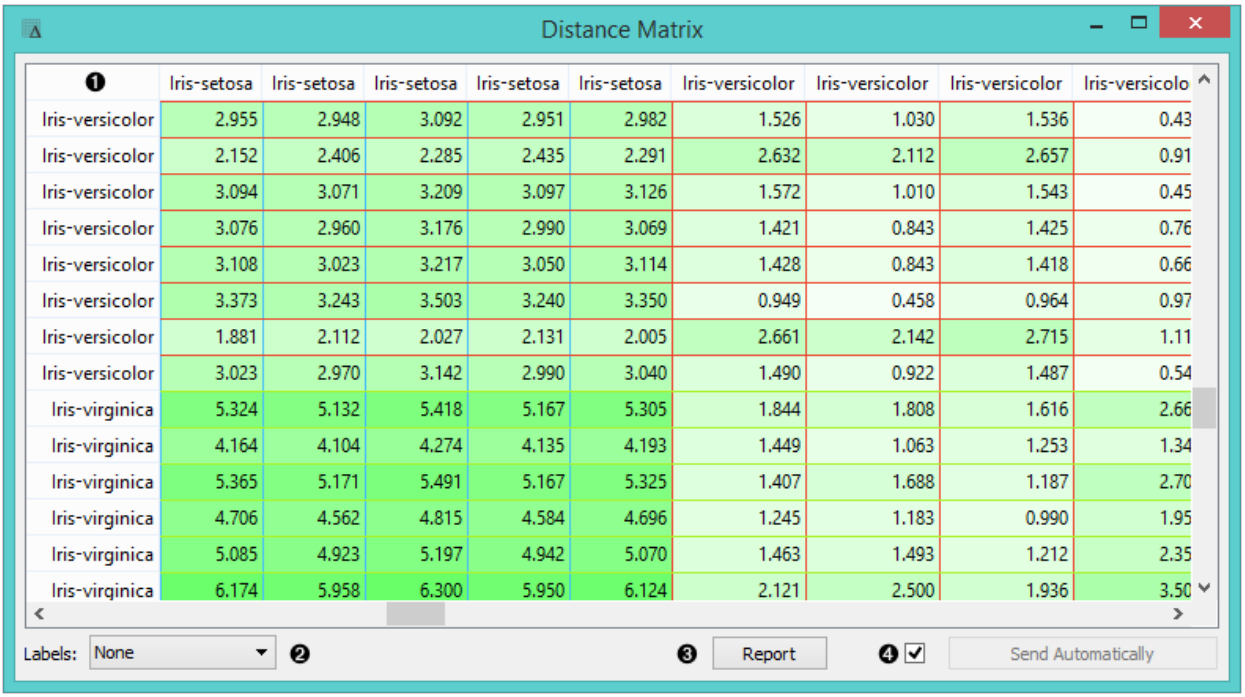
Inputs

- Distances: distance matrix

Outputs

- Distances: distance matrix
- Table: distance measures in a distance matrix

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 dataset 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.



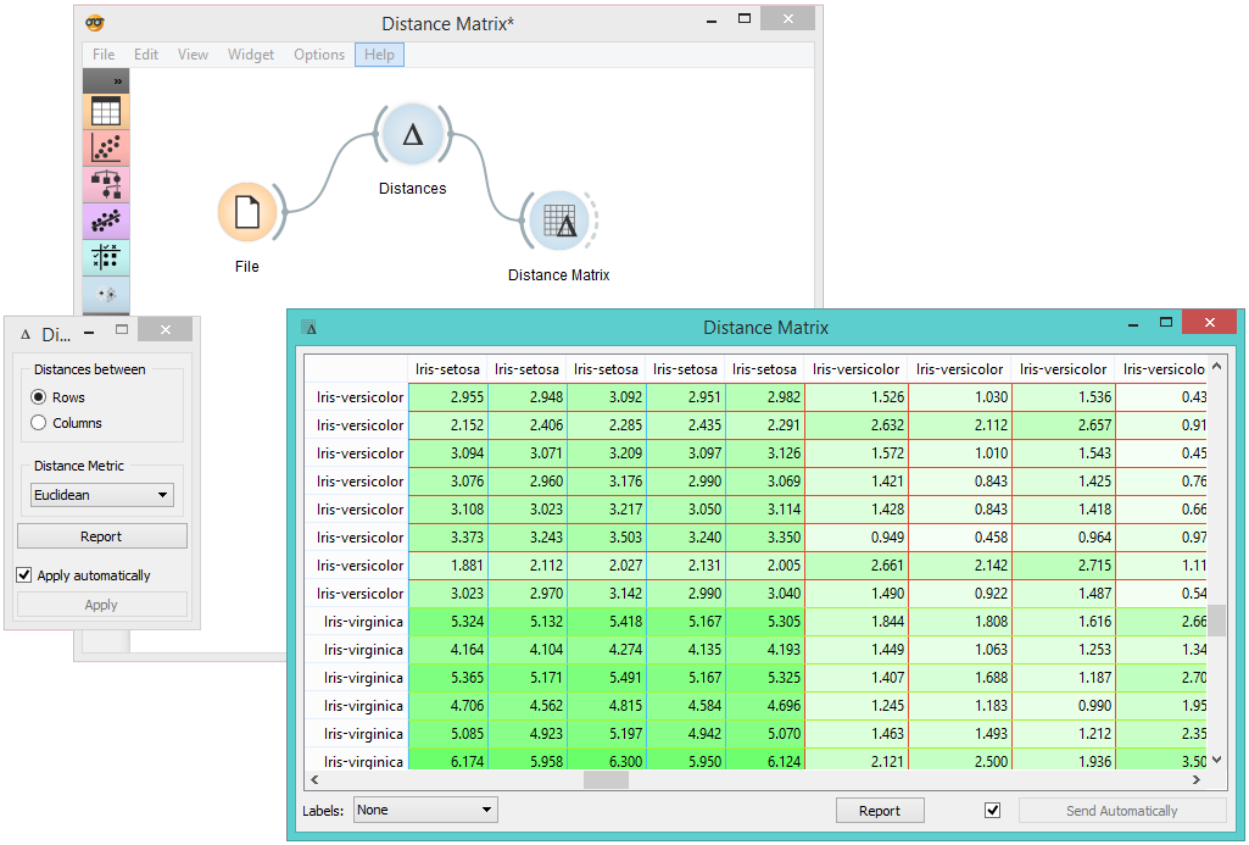
❶	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-versicolor	Iris-versicolor	Iris-versicolor	Iris-versicolo
Iris-versicolor	2.955	2.948	3.092	2.951	2.982	1.526	1.030	1.536	0.43
Iris-versicolor	2.152	2.406	2.285	2.435	2.291	2.632	2.112	2.657	0.91
Iris-versicolor	3.094	3.071	3.209	3.097	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

1. Elements in the dataset 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* dataset and output them in the **Distance Matrix**. It comes as no surprise that Iris Virginica and Iris Setosa are the furthest apart.



2.4.6 Distance Transformation

Transforms distances in a dataset.

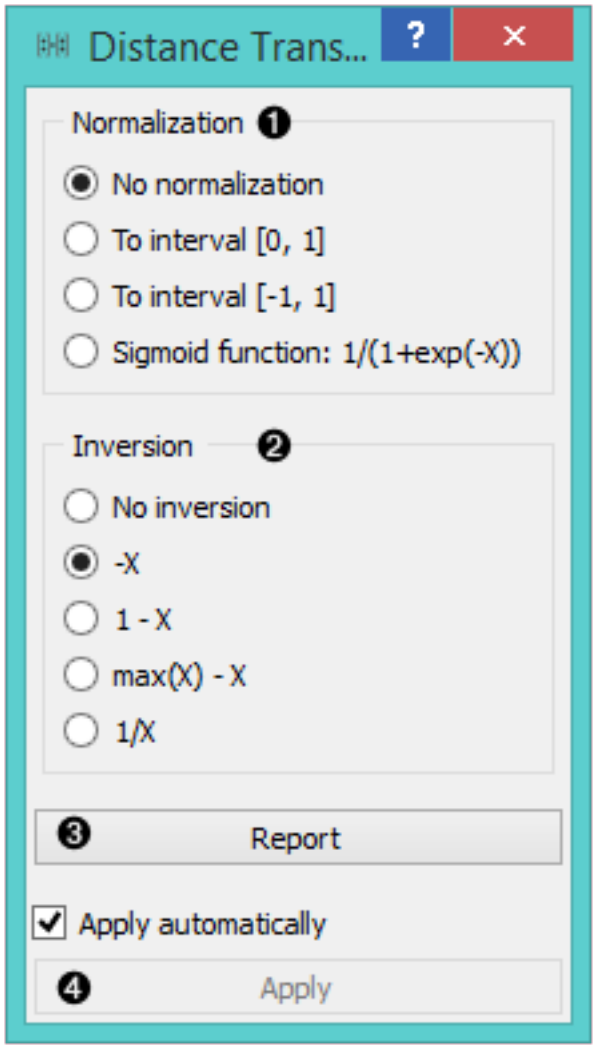
Inputs

- Distances: distance matrix

Outputs

- Distances: transformed distance matrix

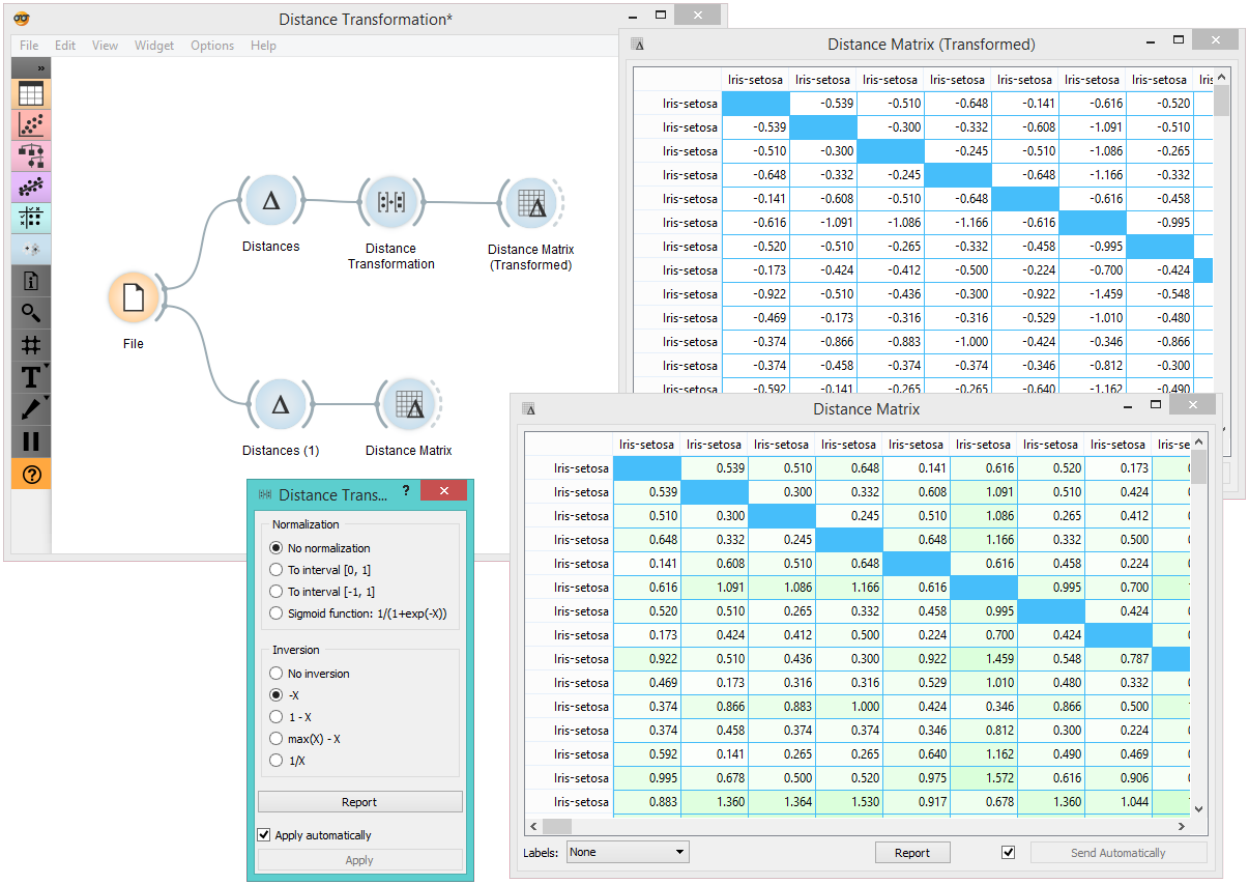
The **Distances Transformation** widget is used for the normalization and inversion of distance matrices. The normalization of data is necessary to bring all the variables into proportion with one another.



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* dataset 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.

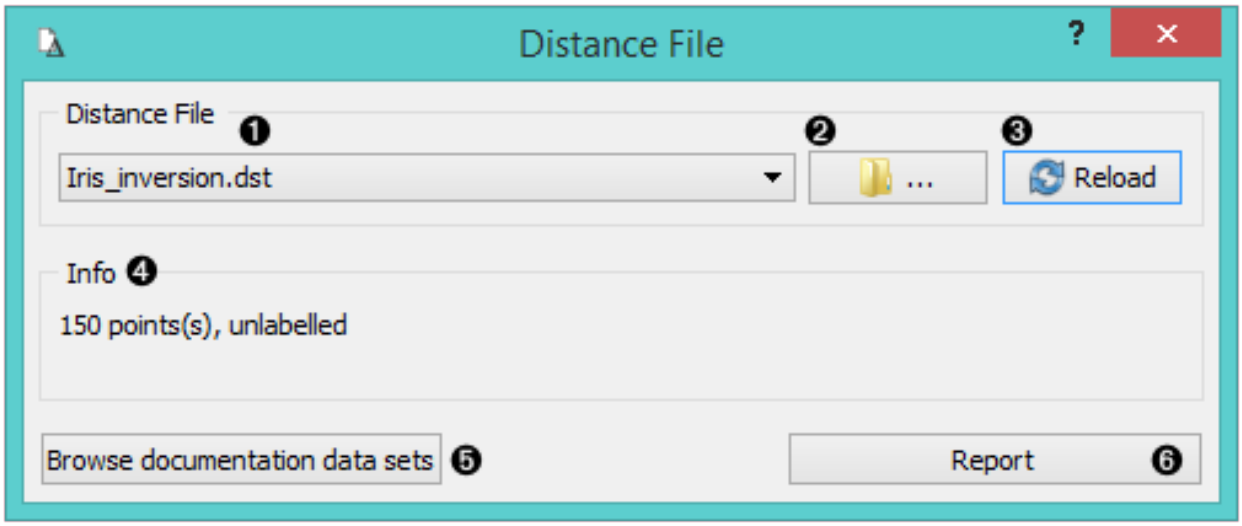


2.4.7 Distance File

Loads an existing distance file.

Outputs

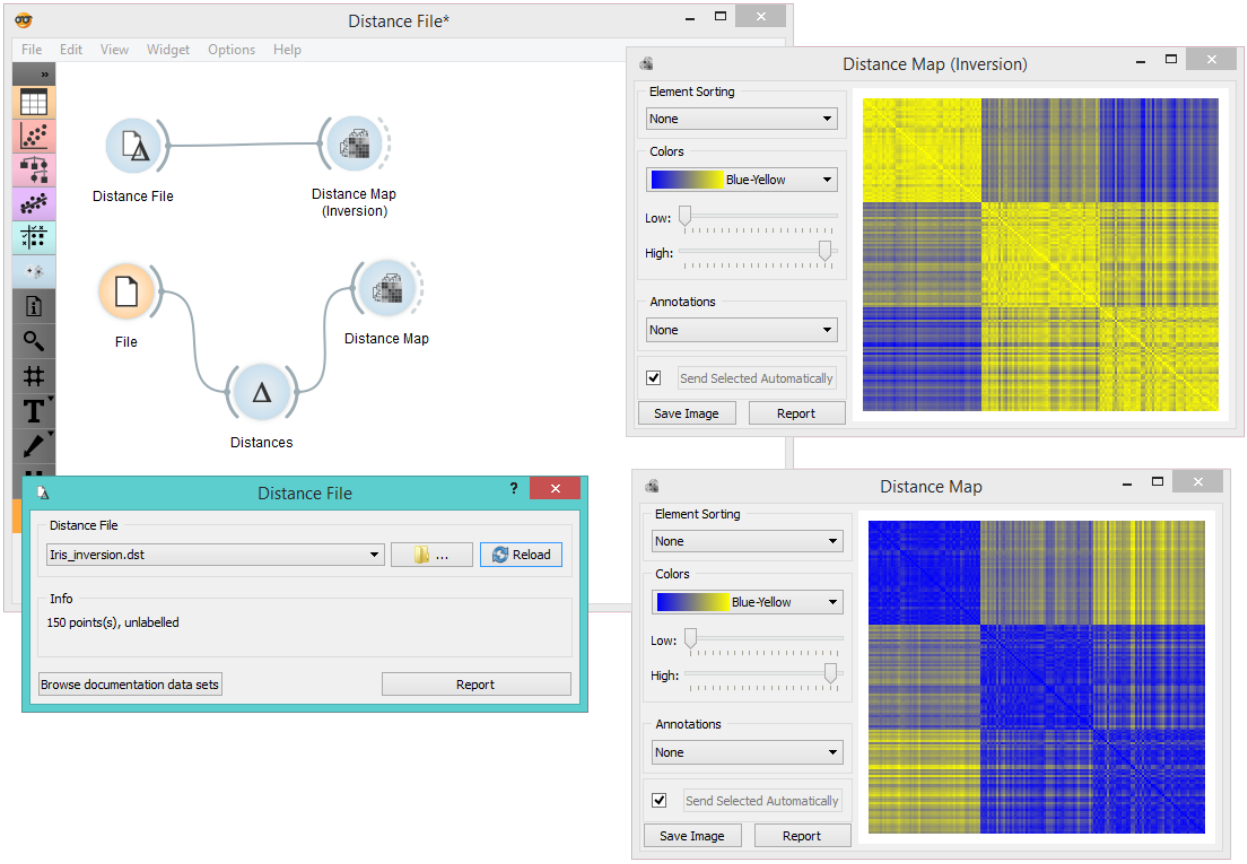
- Distance File: distance matrix



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 datasets.
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* dataset for comparison.

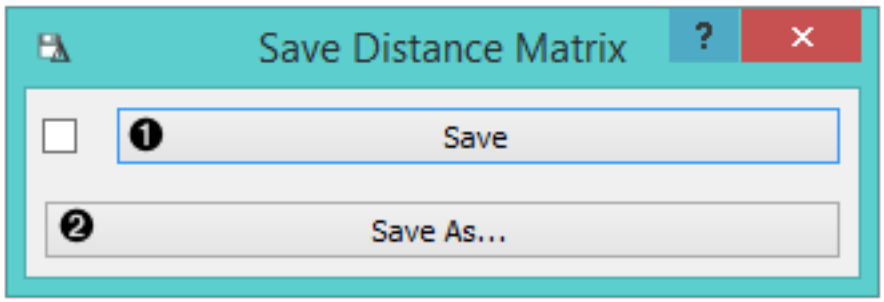


2.4.8 Save Distance Matrix

Saves a distance matrix.

Inputs

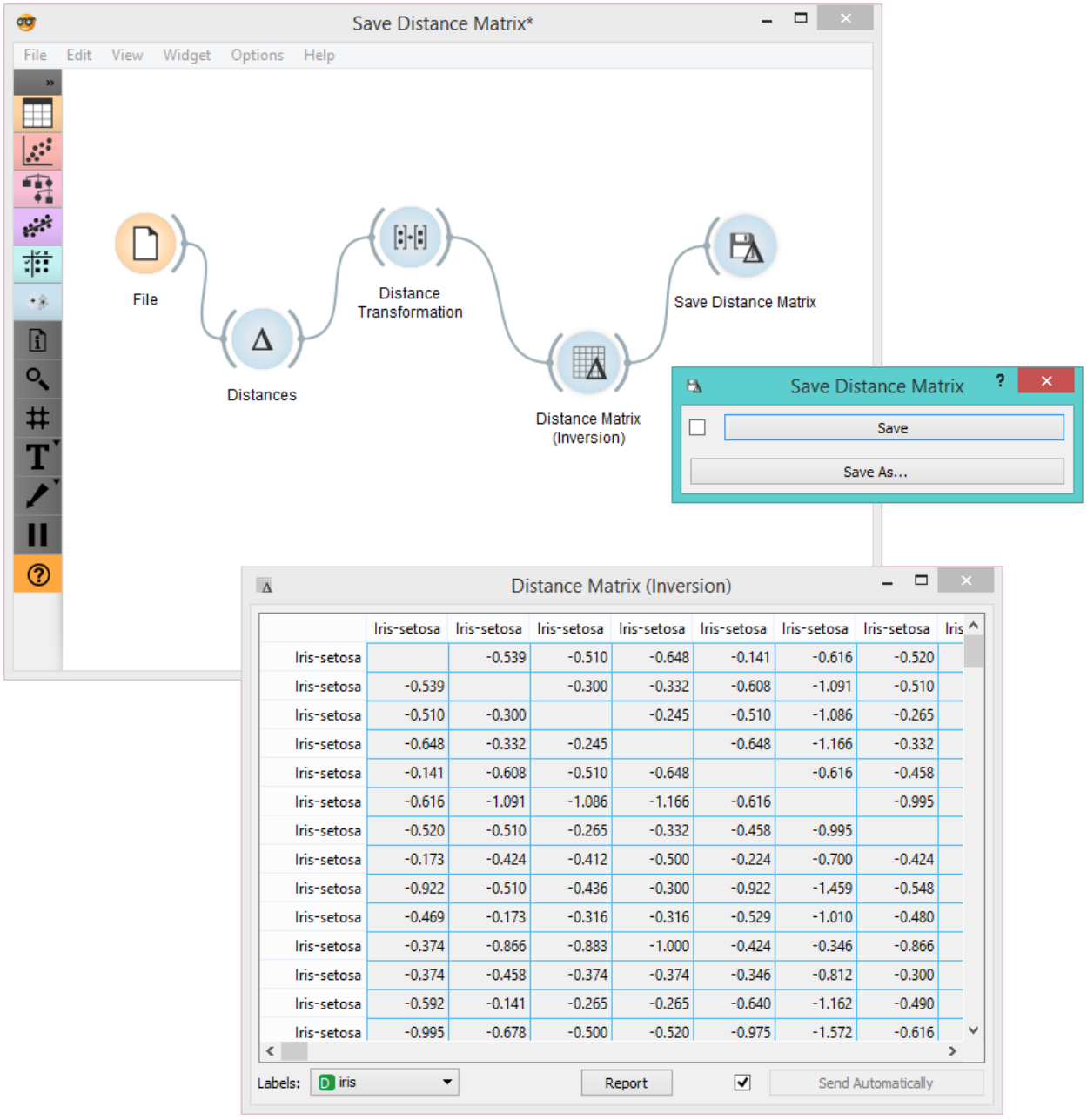
- Distances: distance matrix



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* dataset. 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, see Distance File.



	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris-setosa	Iris
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	

2.4.9 Hierarchical Clustering

Groups items using a hierarchical clustering algorithm.

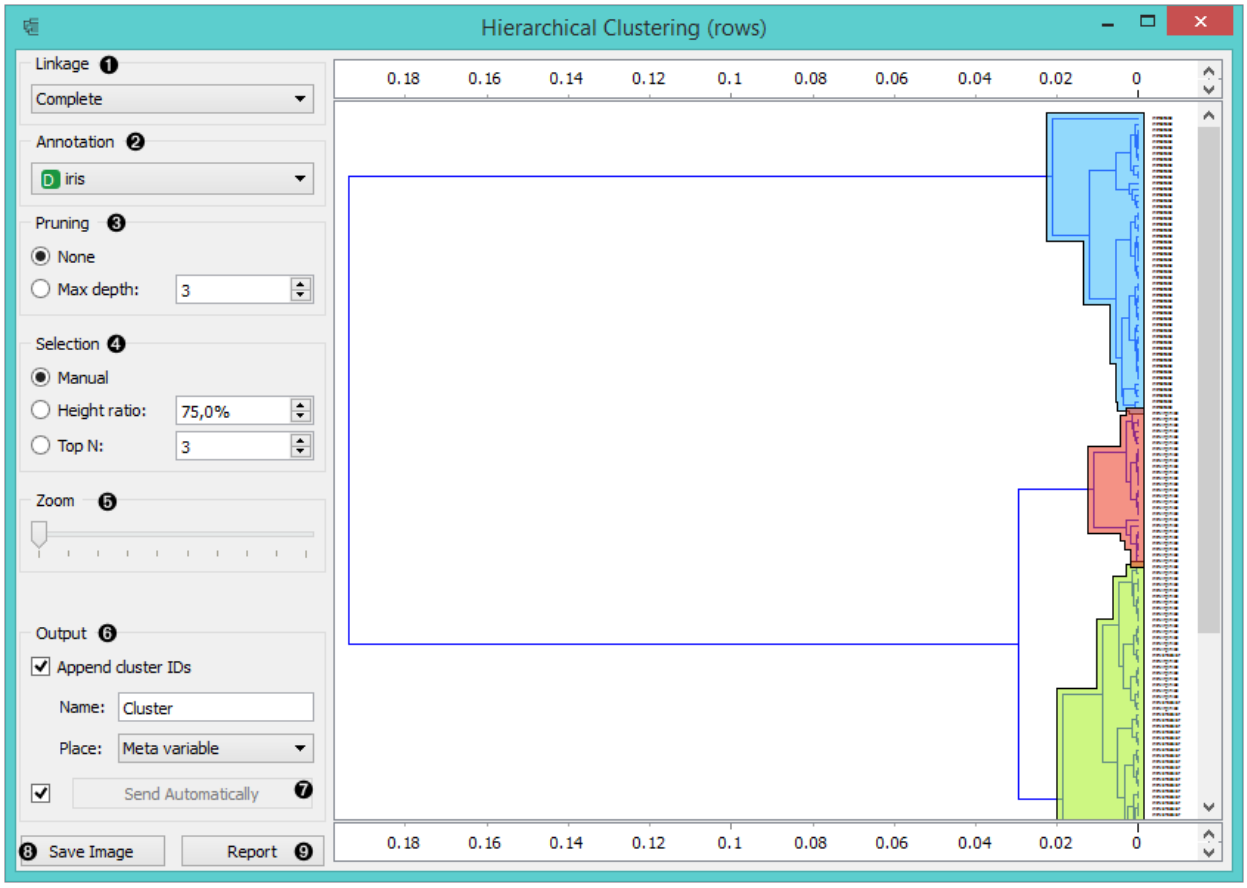
Inputs

- Distances: distance matrix

Outputs

- Selected Data: instances selected from the plot
- Data: data with an additional column showing whether an instance is selected

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



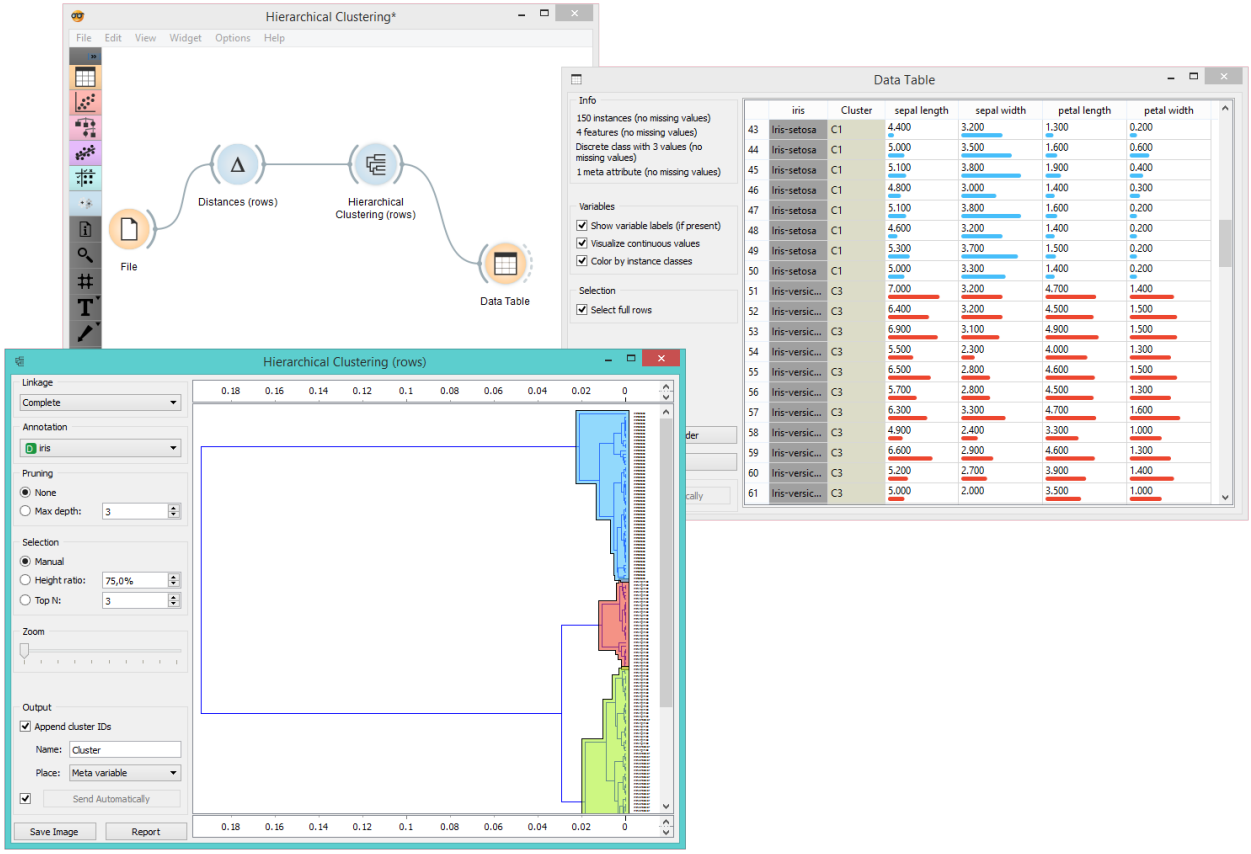
1. The widget supports four ways of measuring distances between clusters:
   - **Single linkage** computes the distance between the closest elements of the two clusters
   - **Average linkage** computes the average distance between elements of the two clusters
   - **Weighted linkage** uses the WPGMA 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.)

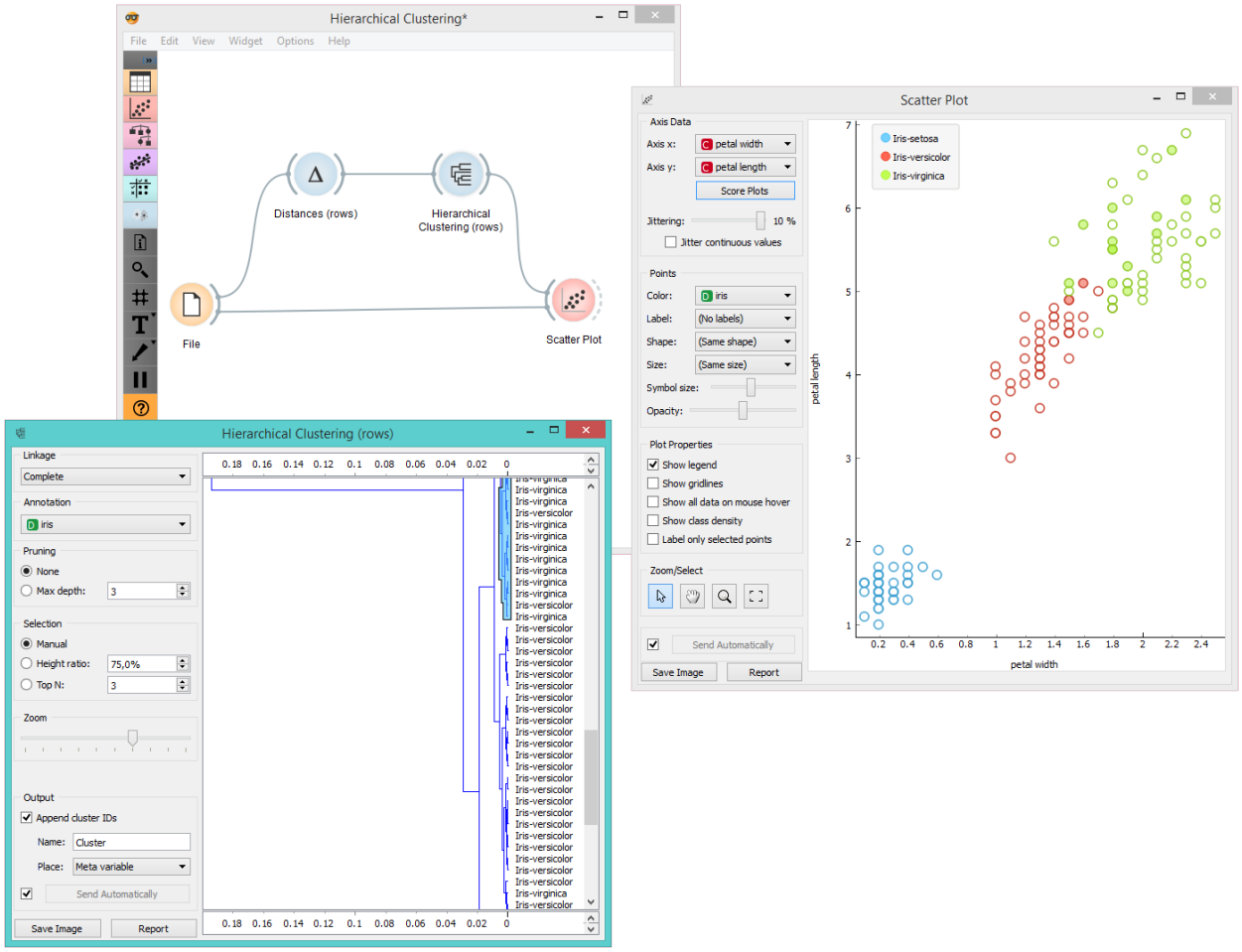
5. Use *Zoom* and scroll to zoom in or out.
6. If the items being clustered are instances, they can be added a cluster index (*Append cluster IDs*). The ID can appear as an ordinary **Attribute**, **Class attribute** or a **Meta attribute**. In the second case, if the data already has a class attribute, 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* dataset 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* dataset 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.



2.4.10 k-Means Clustering

Groups items using the k-Means clustering algorithm.

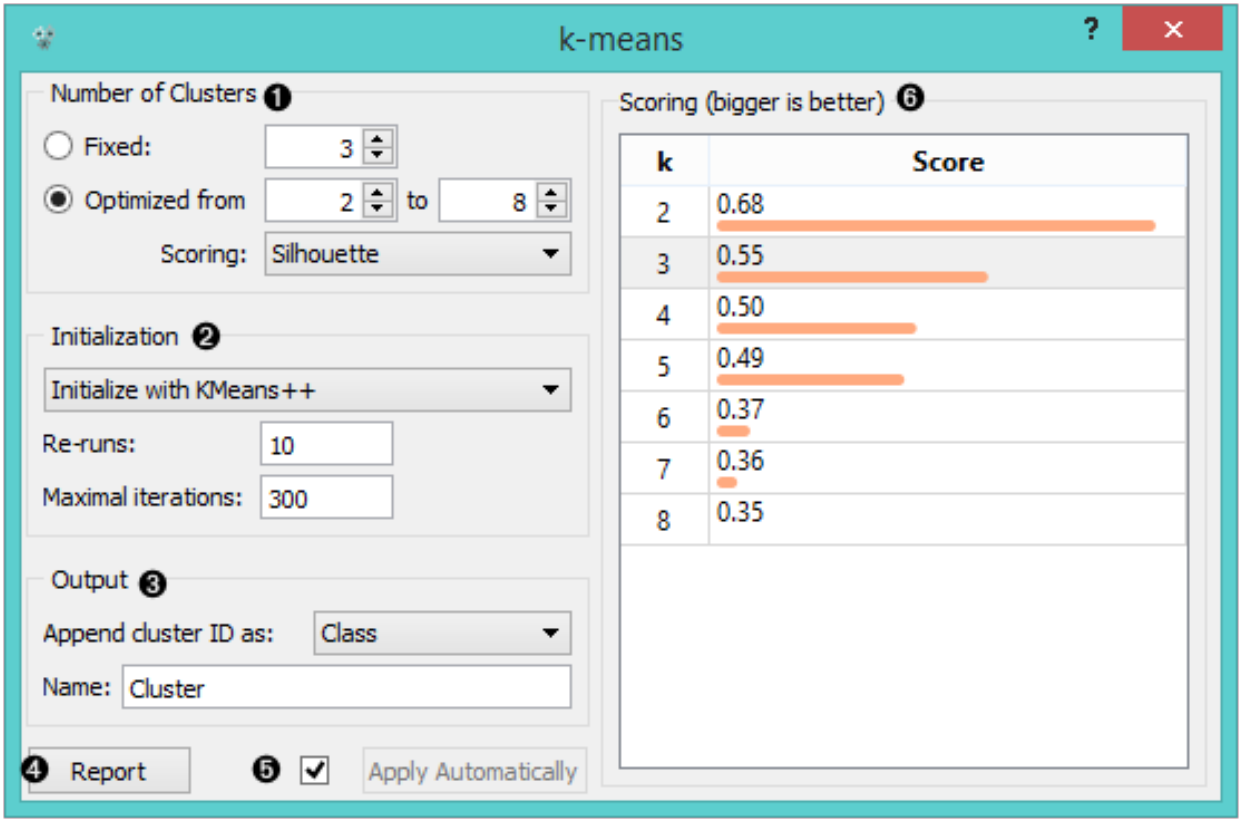
Inputs

- Data: input dataset

Outputs

- Data: dataset with cluster index as a class attribute

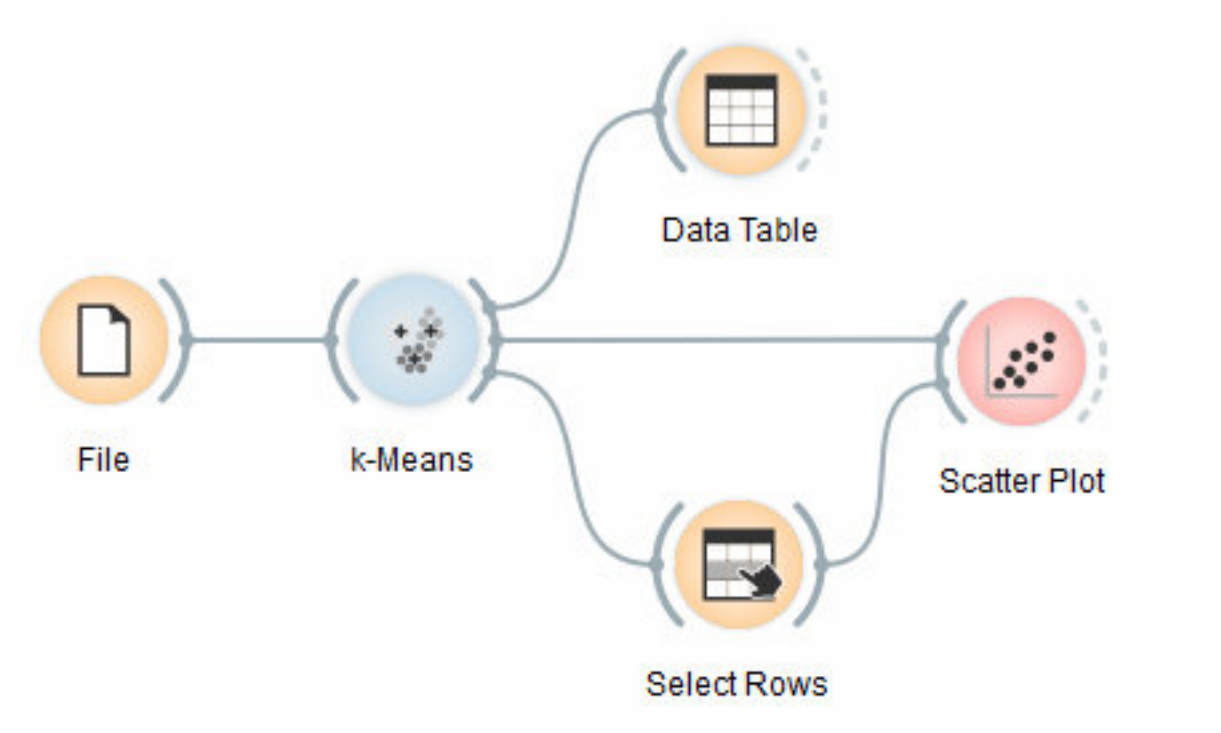
The widget applies the k-Means clustering algorithm to the data and outputs a new dataset in which the cluster index is used as a class attribute. The original class attribute, if it exists, is moved to meta attributes. Scores of clustering results for various k are also shown in the widget.



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 from random initial positions; the result with the lowest within-cluster sum of squares will be used) and **maximal iterations** (the maximum number of iterations within each algorithm run) can be set manually.
3. The widget outputs a new dataset 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.

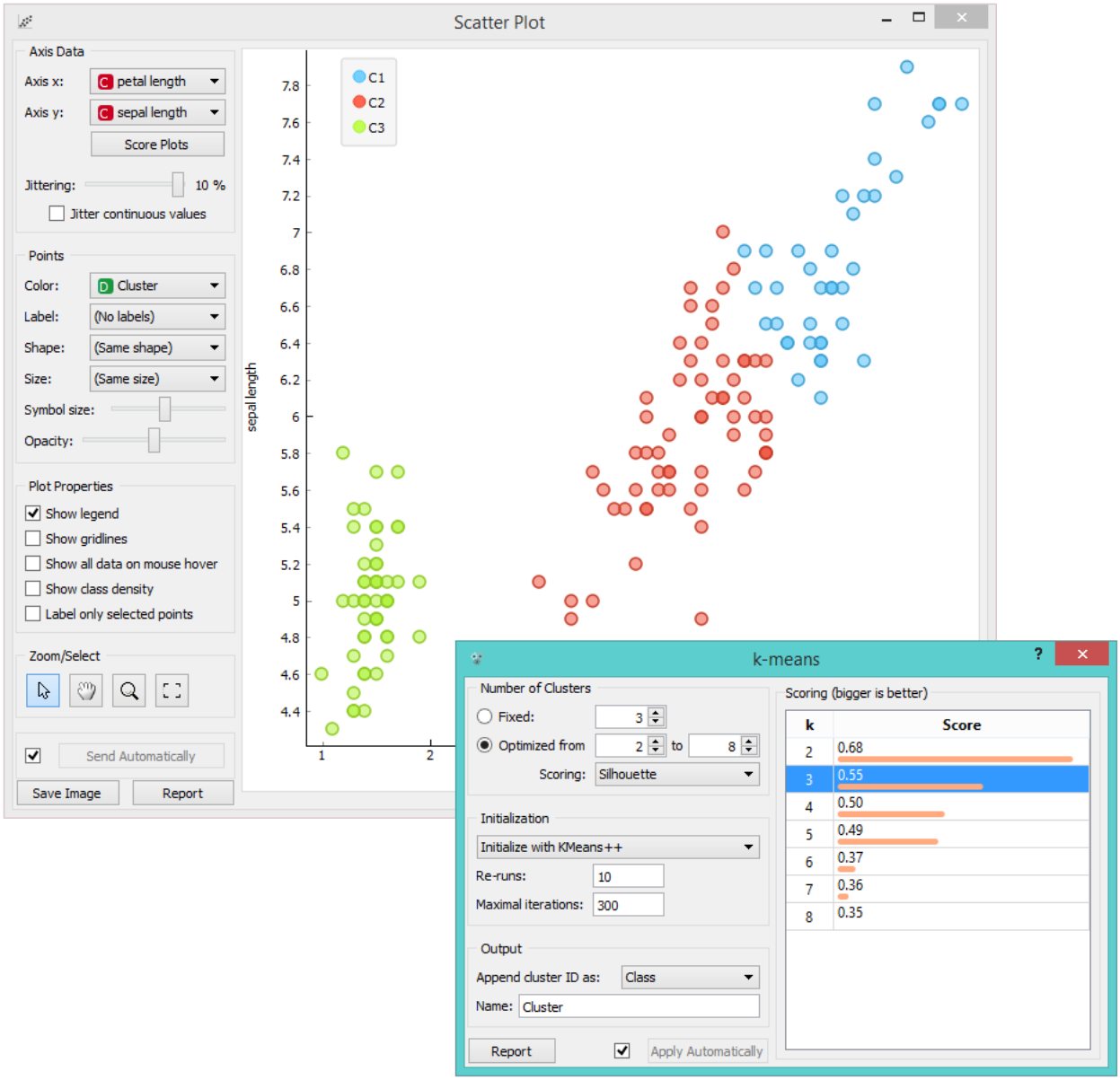
Examples

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

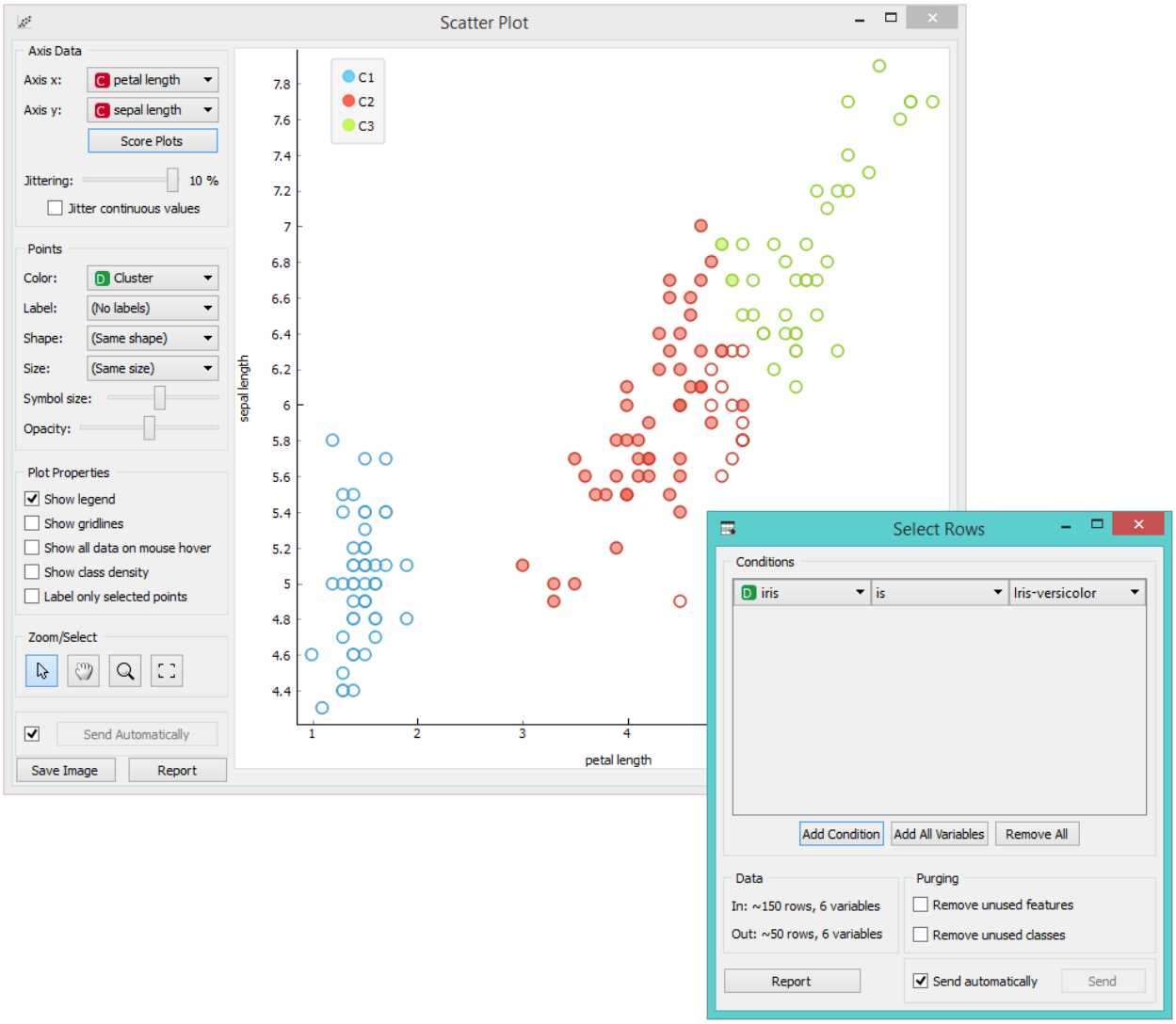


First, we load the *Iris* dataset, divide it into three clusters and show it in the Data Table, where we can observe which instance went into which cluster. The interesting parts are the Scatter Plot and Select Rows.

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.

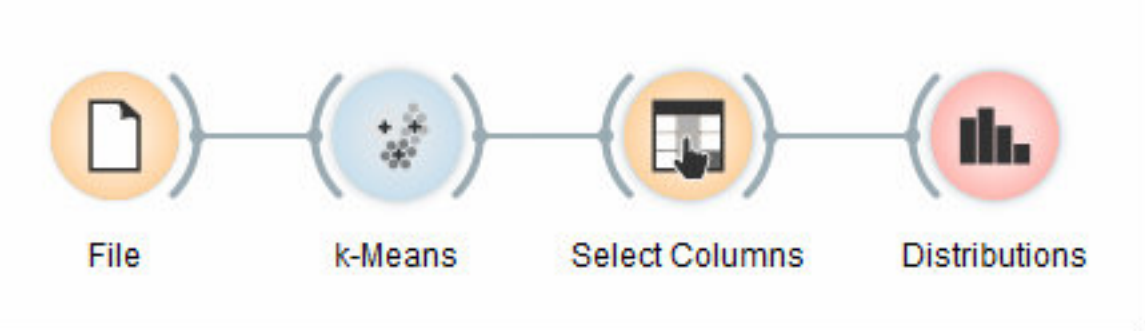


What we are really interested in is how well the clusters induced by the (unsupervised) clustering algorithm match the actual classes in the data. We thus take Select Rows widget, in which we can select individual classes and have the corresponding points marked in the scatter plot. The match is perfect for *setosa*, and pretty good for the other two classes.



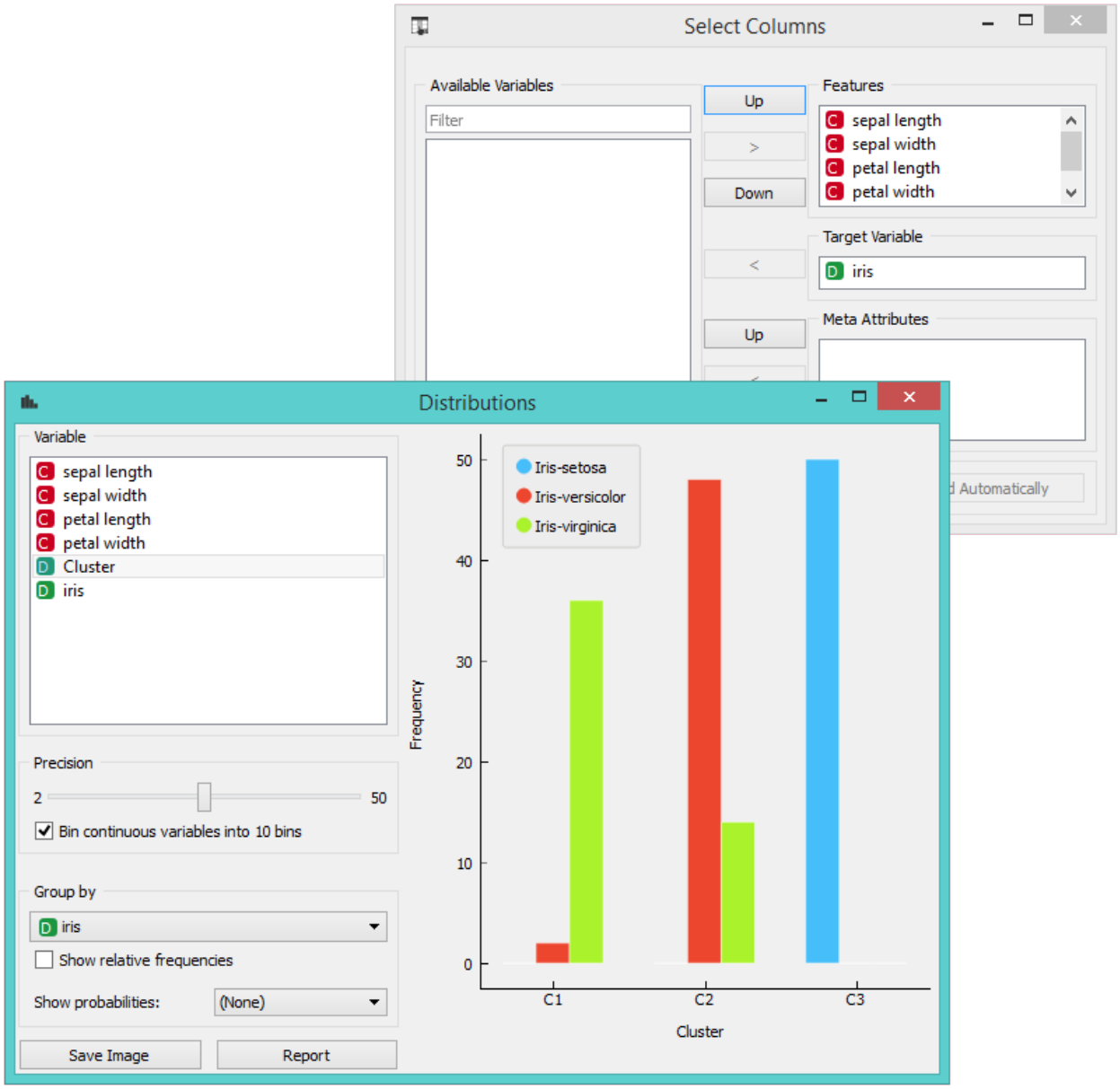
You may have noticed that we left the **Remove unused values/attributes** and **Remove unused classes** in Select Rows unchecked. This is important: if the widget modifies the attributes, it outputs a list of modified instances and the scatter plot cannot compare them to the original data.

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.



2.4.11 Louvain Clustering

Groups items using the Louvain clustering algorithm.

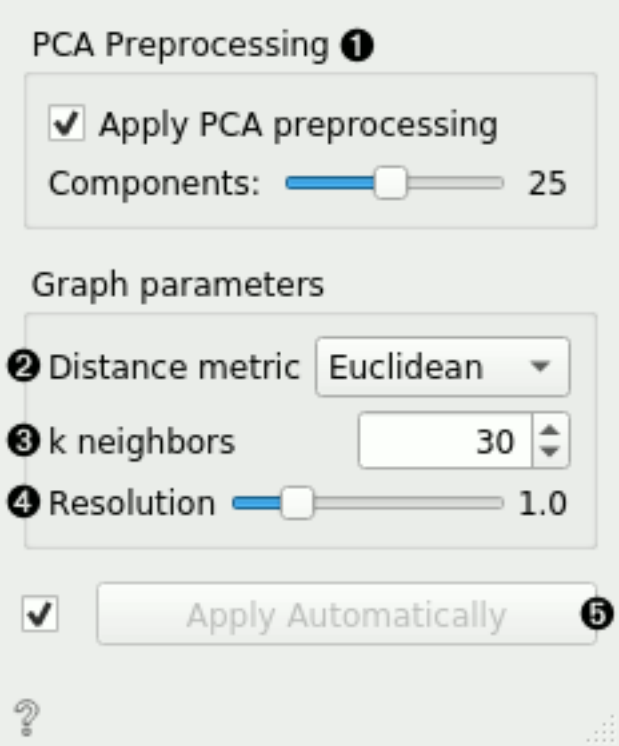
Inputs

- Data: input dataset

Outputs

- Data: dataset with cluster index as a class attribute
- Graph (with the Network addon): the weighted k-nearest neighbor graph

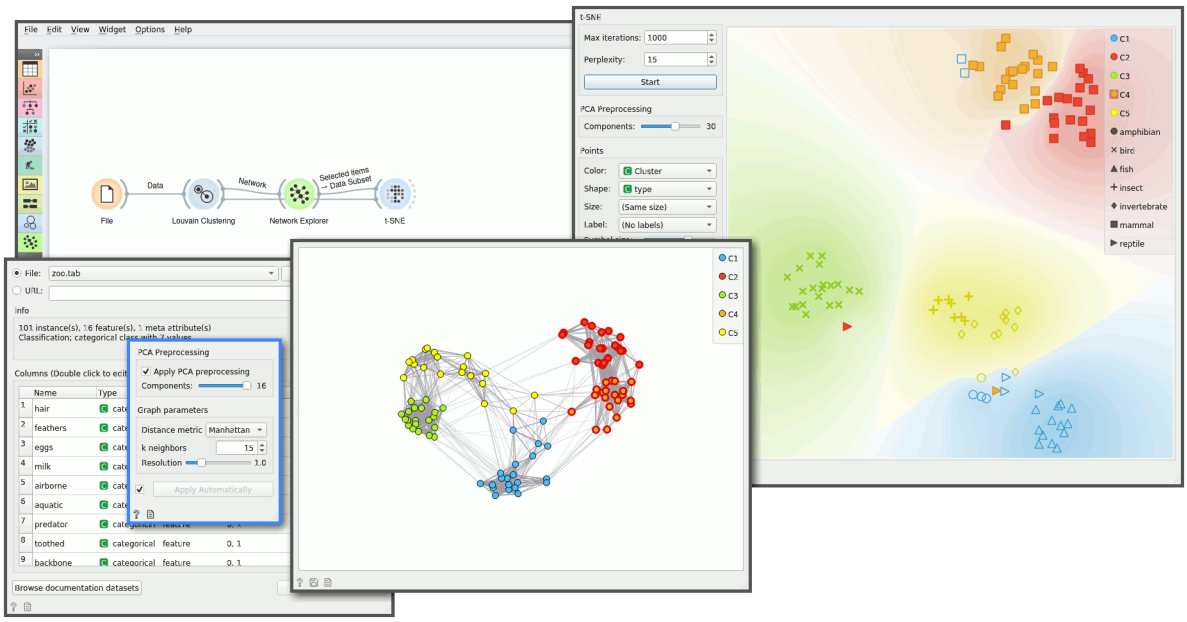
The widget first converts the input data into a k-nearest neighbor graph. To preserve the notions of distance, the Jaccard index for the number of shared neighbors is used to weight the edges. Finally, a modularity optimization community detection algorithm is applied to the graph to retrieve clusters of highly interconnected nodes. The widget outputs a new dataset in which the cluster index is used as a meta attribute.



1. PCA processing is typically applied to the original data to remove noise.
2. The distance metric is used for finding specified number of nearest neighbors.
3. The number of nearest neighbors to use to form the KNN graph.
4. Resolution is a parameter for the Louvain community detection algorithm that affects the size of the recovered clusters. Smaller resolutions recover smaller, and therefore a larger number of clusters, and conversely, larger values recover clusters containing more data points.
5. When *Apply Automatically* is ticked, the widget will automatically communicate all changes. Alternatively, click *Apply*.

Example

Louvain Clustering converts the dataset into a graph, where it finds highly interconnected nodes. We can visualize the graph itself using the **Network Explorer** from the Network addon.



References

Blondel, Vincent D., et al. "Fast unfolding of communities in large networks." Journal of statistical mechanics: theory and experiment 2008.10 (2008): P10008.

Lambiotte, Renaud, J-C. Delvenne, and Mauricio Barahona. "Laplacian dynamics and multiscale modular structure in networks." arXiv preprint, arXiv:0812.1770 (2008).

2.4.12 MDS

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

Inputs

- Data: input dataset
- Distances: distance matrix
- Data Subset: subset of instances

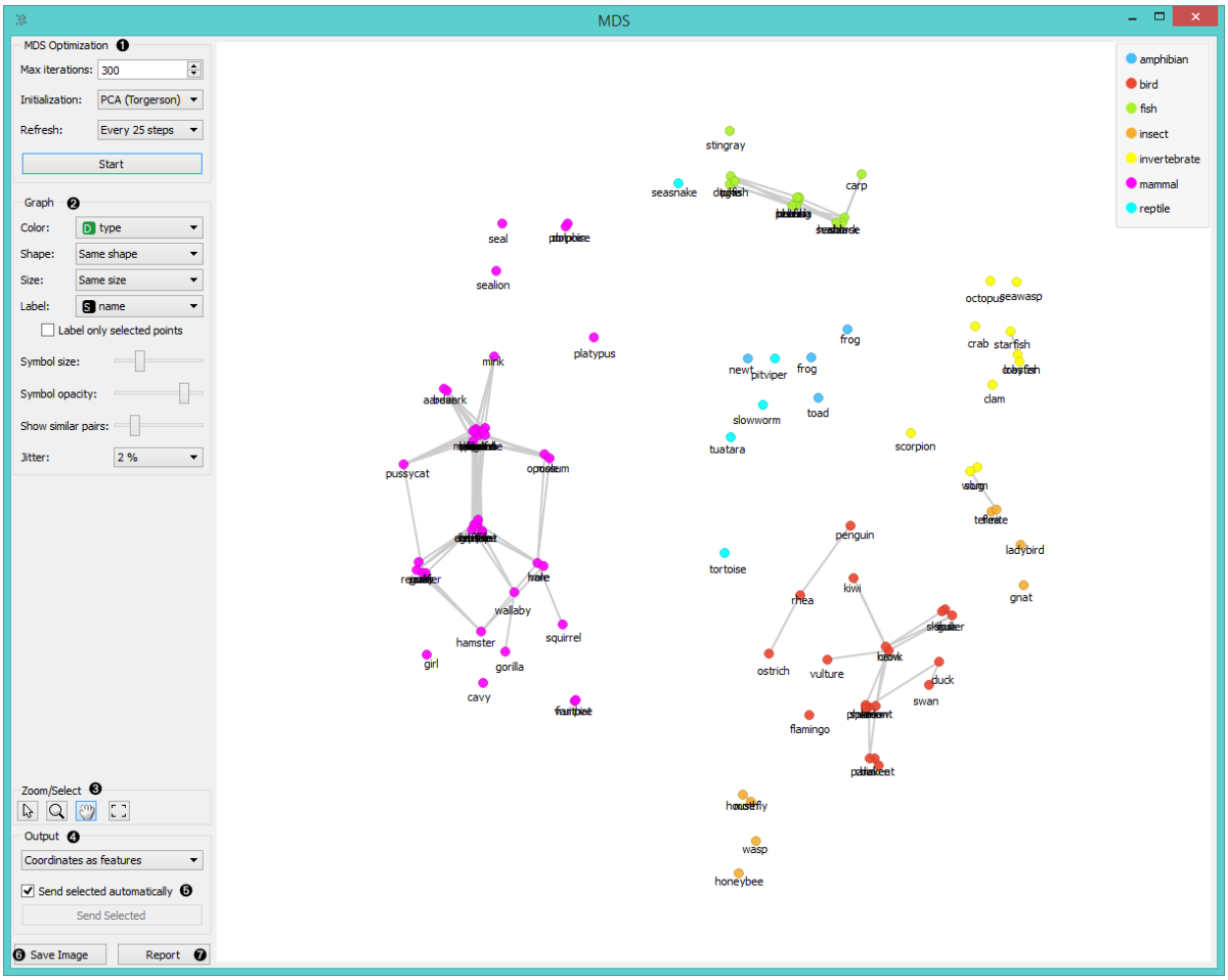
Outputs

- Selected Data: instances selected from the plot
- Data: dataset with MDS coordinates

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 dataset 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.

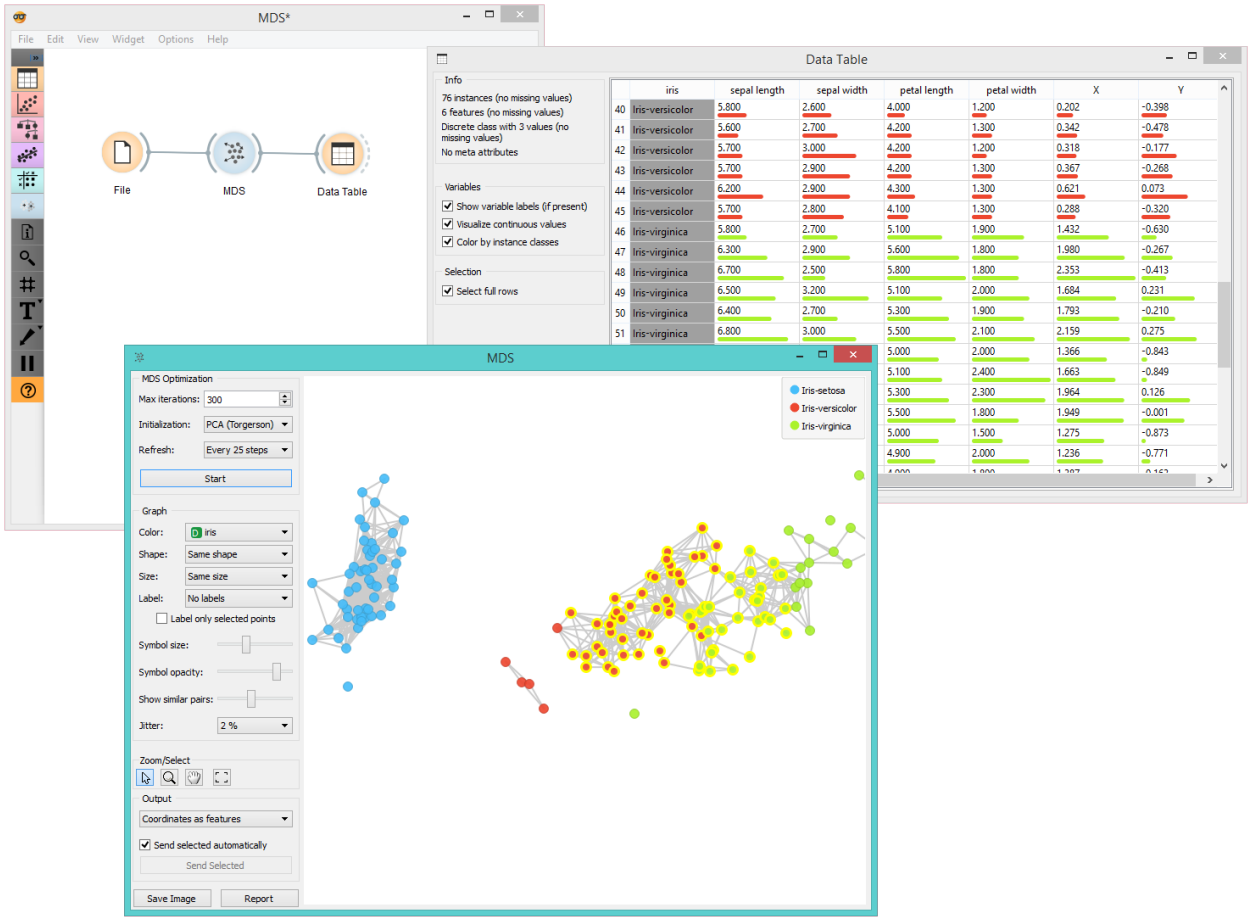
- **Symbol opacity**: Adjust the transparency level of the dots.
- **Show similar pairs**: Adjust the strength of network lines.
- **Jitter**: Set jittering to prevent the dots from overlapping.

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 dataset)
   - **Coordinates only** (MDS coordinates)
   - **Coordinates as features** (input dataset + MDS coordinates as regular attributes)
   - **Coordinates as meta attributes** (input dataset + 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 <../visualize/scatterplot>` widget, so we recommend reading that widget's description as well.

2.4.13 Example

The above graphs were drawn using the following simple schema. We used the *iris.tab* dataset. Using the `Distances <../unsupervised/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 <../data/datatable>` widget.



2.4.14 References

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

2.4.15 t-SNE

Two-dimensional data projection with t-SNE.

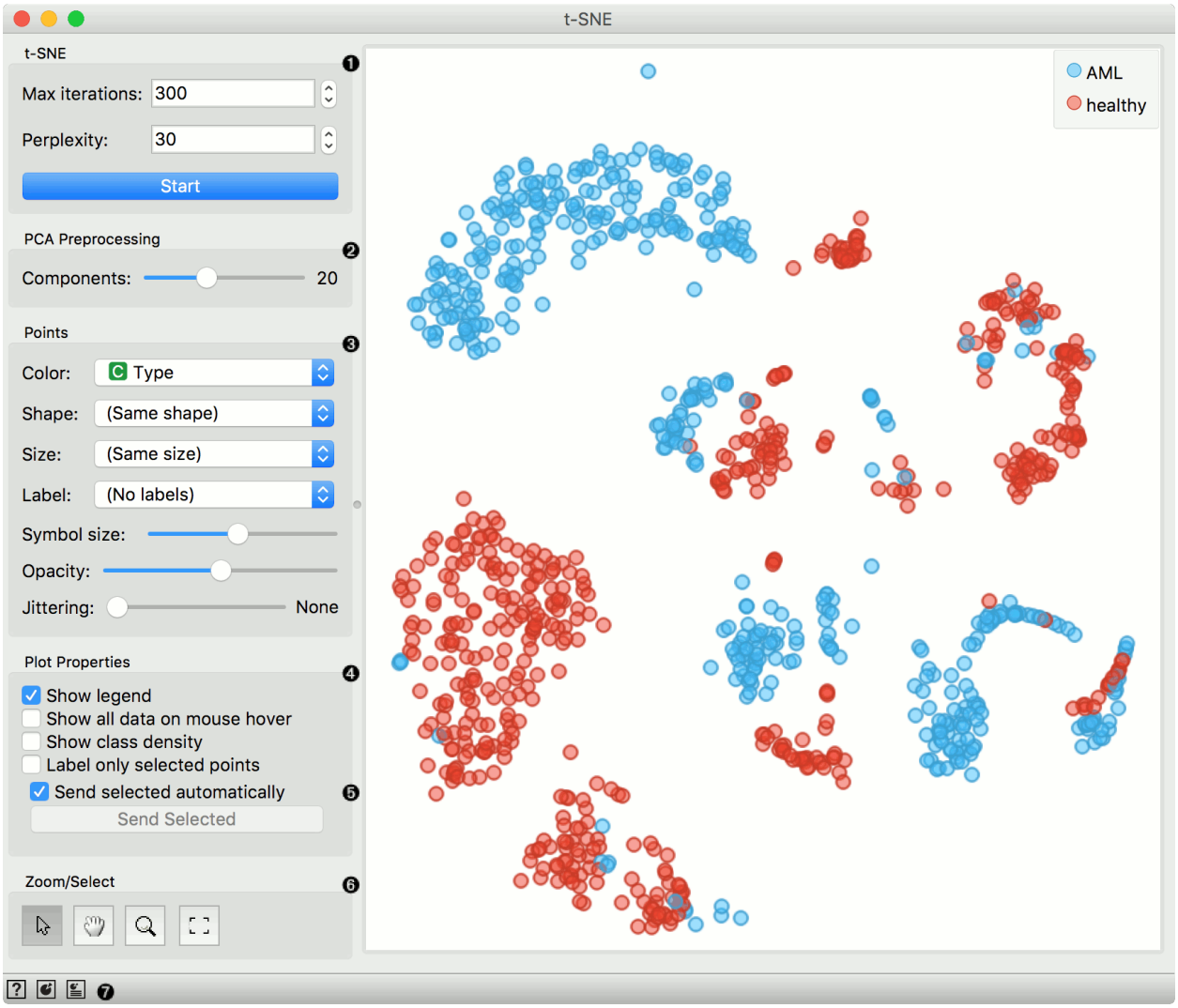
Inputs

- Data: input dataset
- Data Subset: subset of instances

Outputs

- Selected Data: instances selected from the plot
- Data: data with an additional column showing whether a point is selected

The **t-SNE** widget plots the data with a t-distributed stochastic neighbor embedding method. t-SNE is a dimensionality reduction technique, similar to MDS, where points are mapped to 2-D space by their probability distribution.



1. Number of iterations for optimization and the measure of perplexity. Press Start to (re-)run the optimization.
2. Select the number of PCA components used for projection.
3. Set the color of the displayed points (you will get colors for discrete values and grey-scale points for continuous). Set shape, size and label to differentiate between points. Set symbol size and opacity for all data points. Set jittering to randomly disperse data points.
4. Adjust *plot properties*:
   - *Show legend* displays a legend on the right. Click and drag the legend to move it.
   - *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.
   - *Label only selected points* allows you to select individual data instances and label them.
5. If *Send selected automatically* is ticked, changes are communicated automatically. Alternatively, press *Send Selected*.
6. *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.
7. Access help, save image or produce a report.

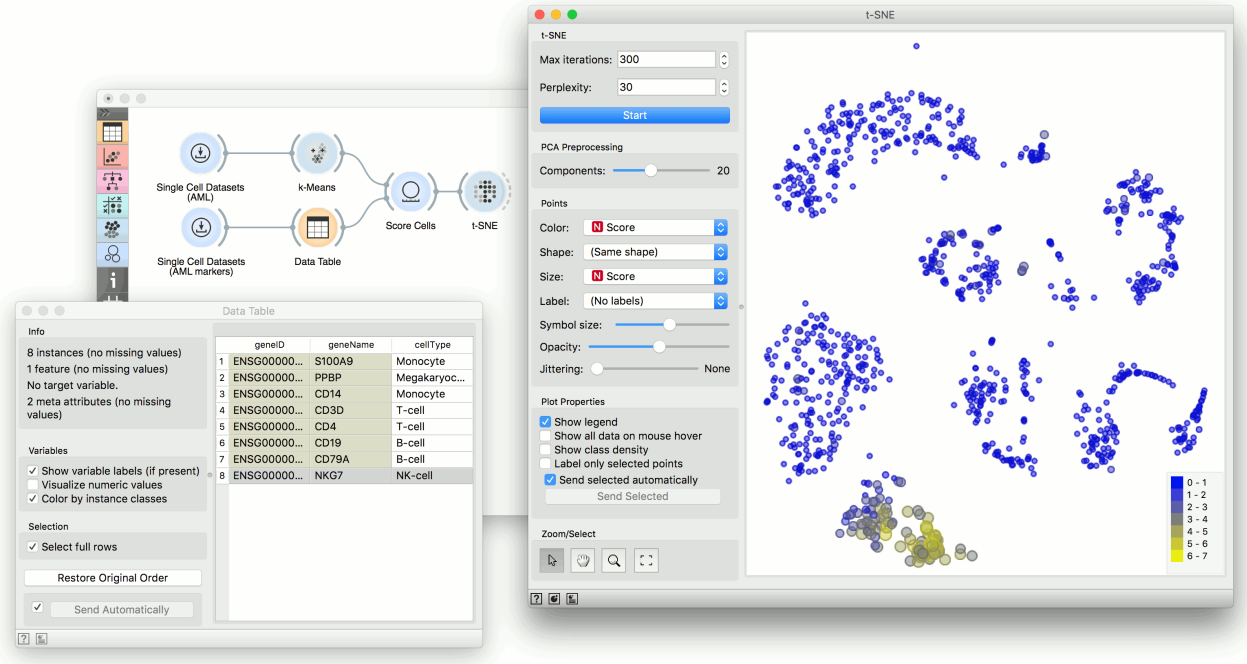
Example

We will use **Single Cell Datasets** widget to load *Bone marrow mononuclear cells with AML (sample)* data. Then we will pass it through **k-Means** and select 2 clusters from Silhouette Scores. Ok, it looks like there might be two distinct clusters here.

But can we find subpopulations in these cells? Let us load *Bone marrow mononuclear cells with AML (markers)* with **Single Cell Datasets**. Now, pass the marker genes to **Data Table** and select, for example, natural killer cells from the list (NKG7).

Pass the markers and k-Means results to **Score Cells** widget and select *geneName* to match markers with genes. Finally, add **t-SNE** to visualize the results.

In **t-SNE**, use *Scores* attribute to color the points and set their size. We see that killer cells are nicely clustered together and that t-SNE indeed found subpopulations.



2.4.16 Manifold Learning

Nonlinear dimensionality reduction.

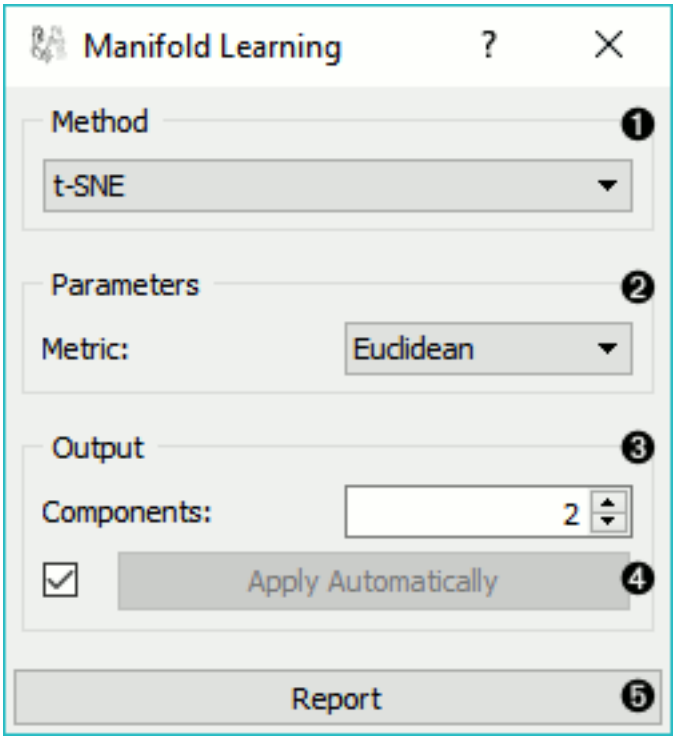
Inputs

- Data: input dataset

Outputs

- Transformed Data: dataset with reduced coordinates

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.

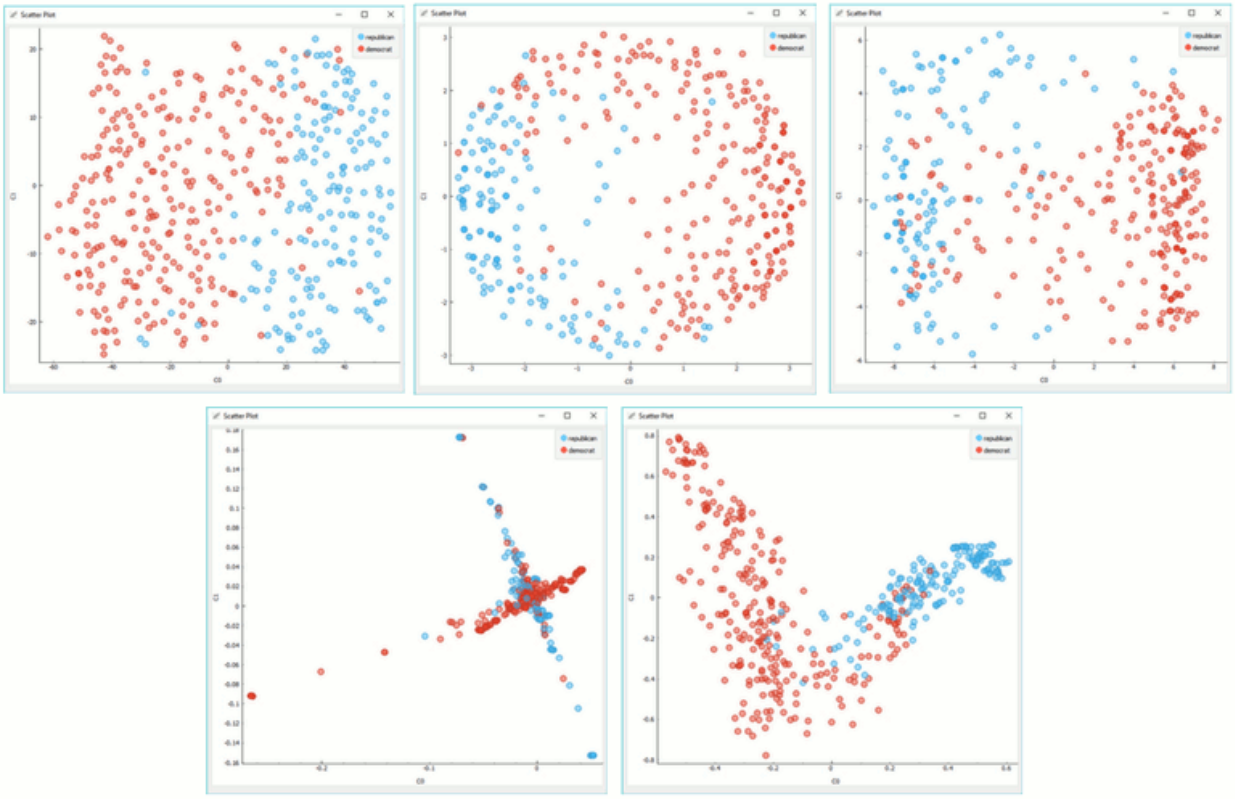


1. Method for manifold learning:
   - t-SNE
   - MDS, see also MDS widget
   - Isomap
   - Locally Linear Embedding
   - Spectral Embedding
2. Set parameters for the method:
   - t-SNE (distance measures):
     - *Euclidean* distance
     - *Manhattan*
     - *Chebyshev*
     - *Jaccard*
     - *Mahalanobis*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

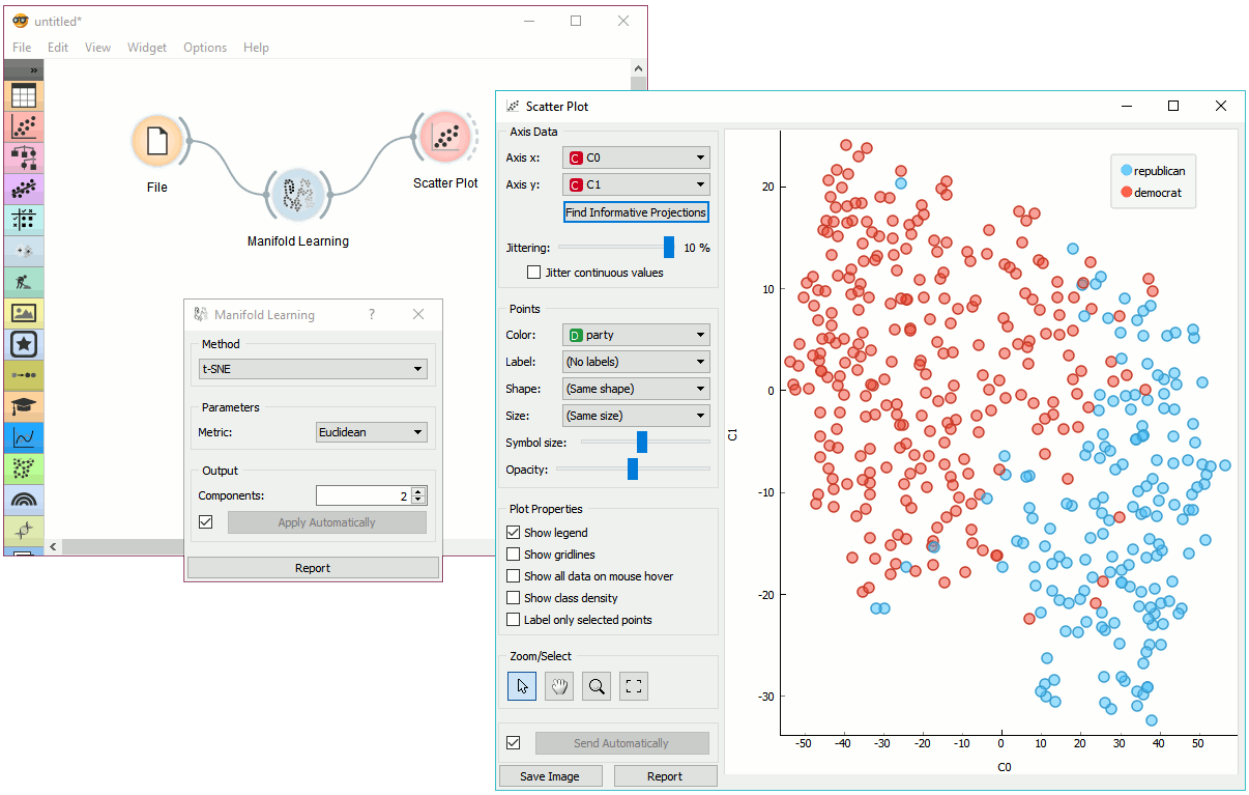
Manifold Learning widget produces different embeddings for high-dimensional data.



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 datasets 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.



2.5 Evaluation

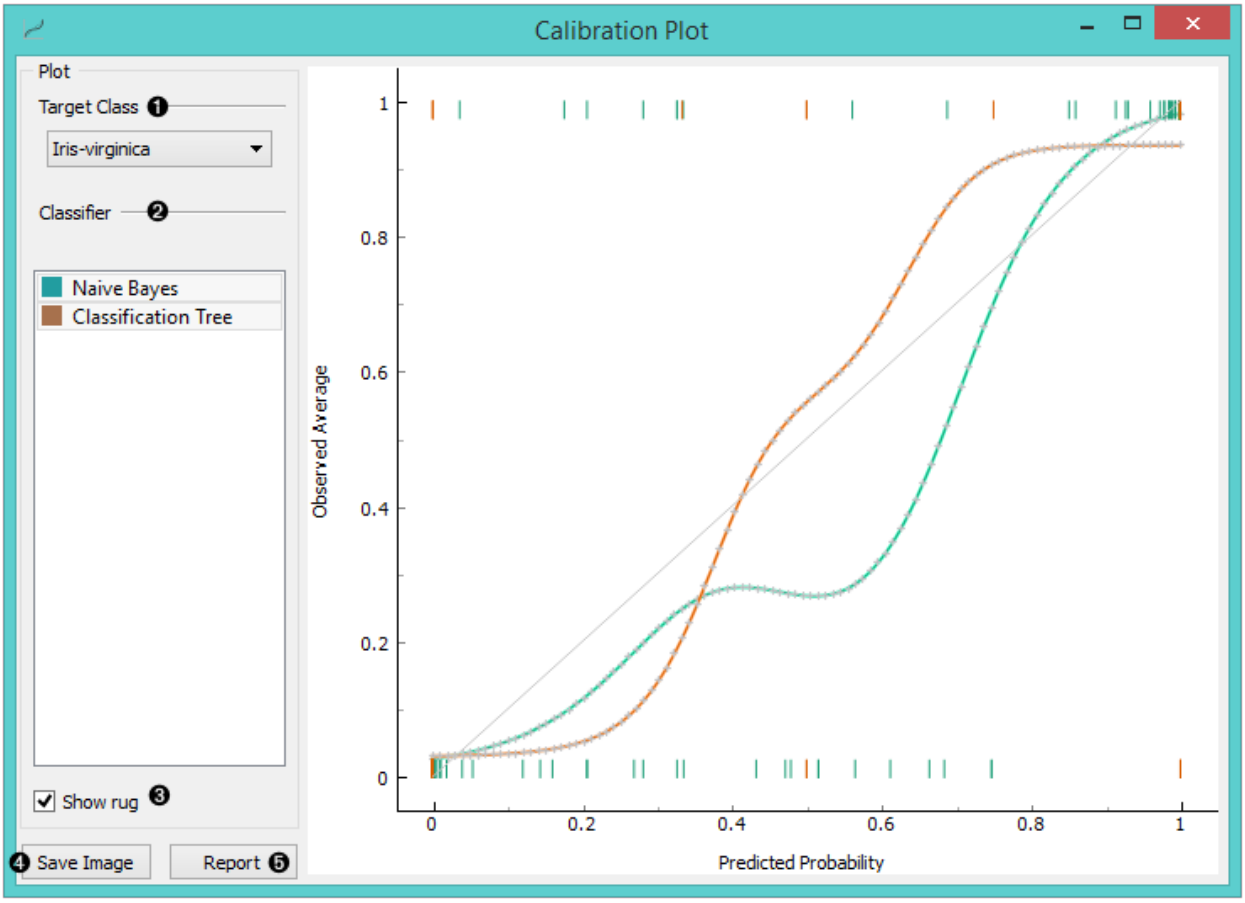
2.5.1 Calibration Plot

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

Inputs

- Evaluation Results: results of testing classification algorithms

The Calibration Plotplots class probabilities against those predicted by the classifier(s).

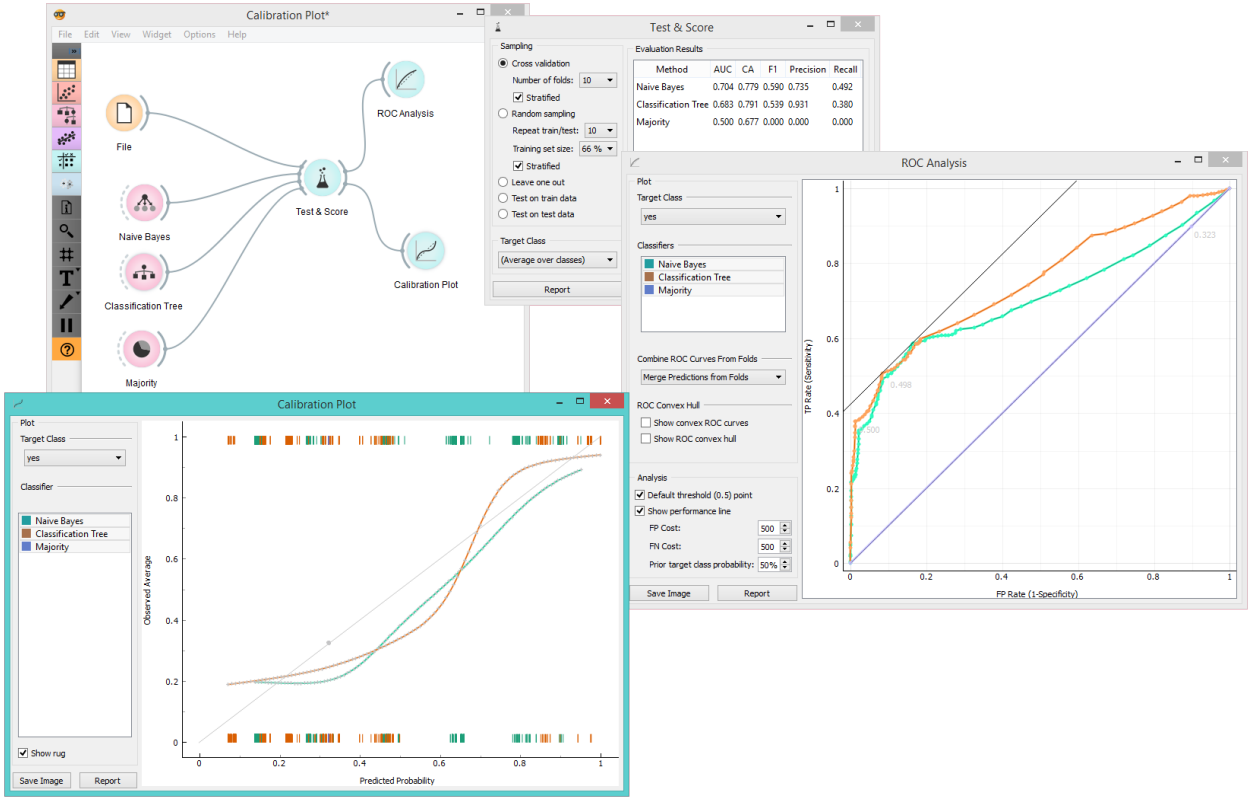


1. Select the desired target class from the drop down menu.
2. Choose which classifiers to plot. The diagonal represents optimal behavior; the closer the classifier's curve gets, the more accurate its prediction probabilities are. Thus we would use this widget to see whether a classifier is overly optimistic (gives predominantly positive results) or pessimistic (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, Tree and Constant) and input them into Test & Score. We used the *Titanic* dataset. 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.



2.5.2 Predictions

Shows proportions between the predicted and actual class.

Inputs

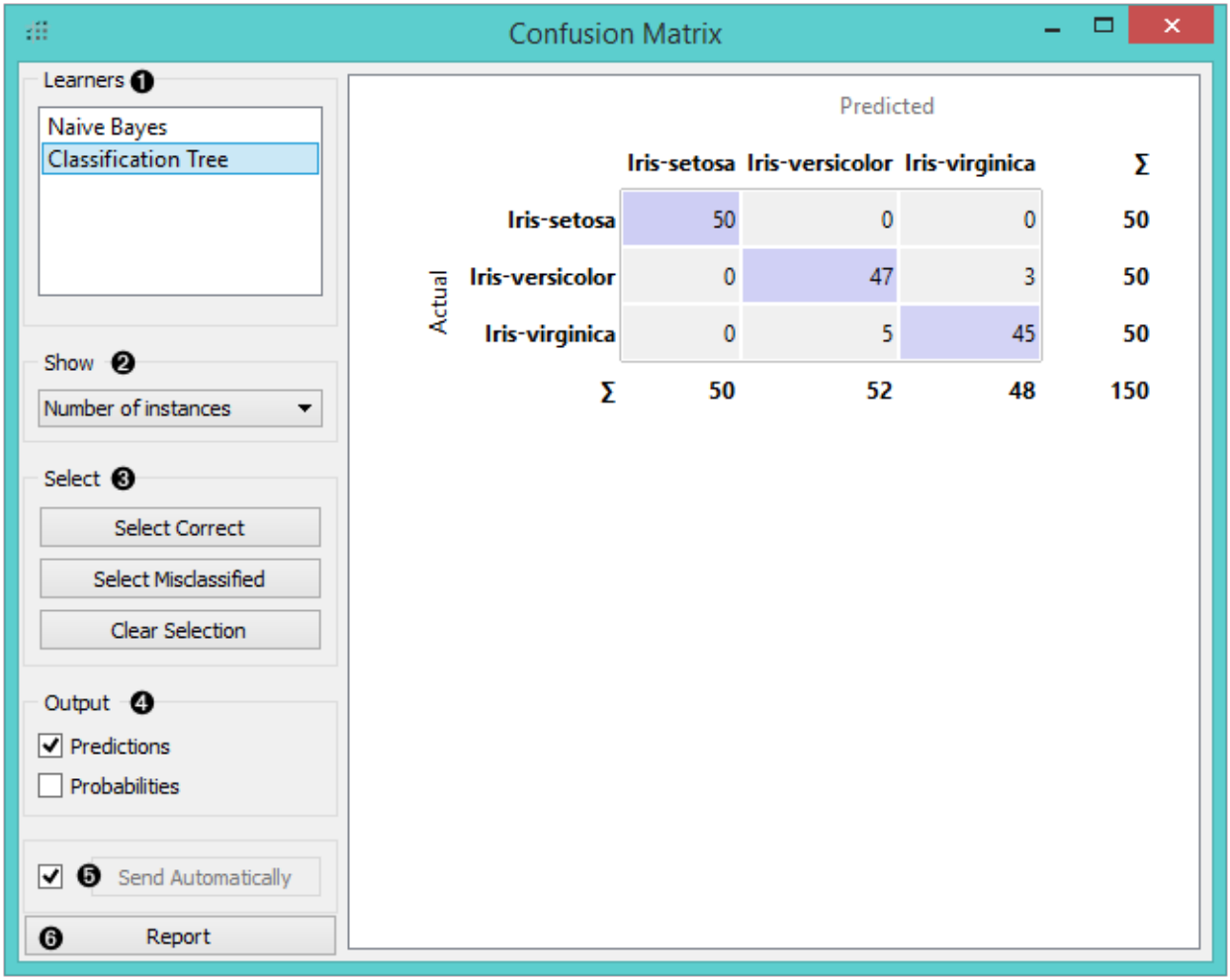
- Evaluation results: results of testing classification algorithms

Outputs

- Selected Data: data subset selected from confusion matrix
- Data: data with the additional information on whether a data instance was selected

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 & Score; 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 Tree and Naive Bayesian models trained and tested on the *iris* data. The right-hand side of the widget contains the matrix for the naive Bayesian model (since this model 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 versicol-

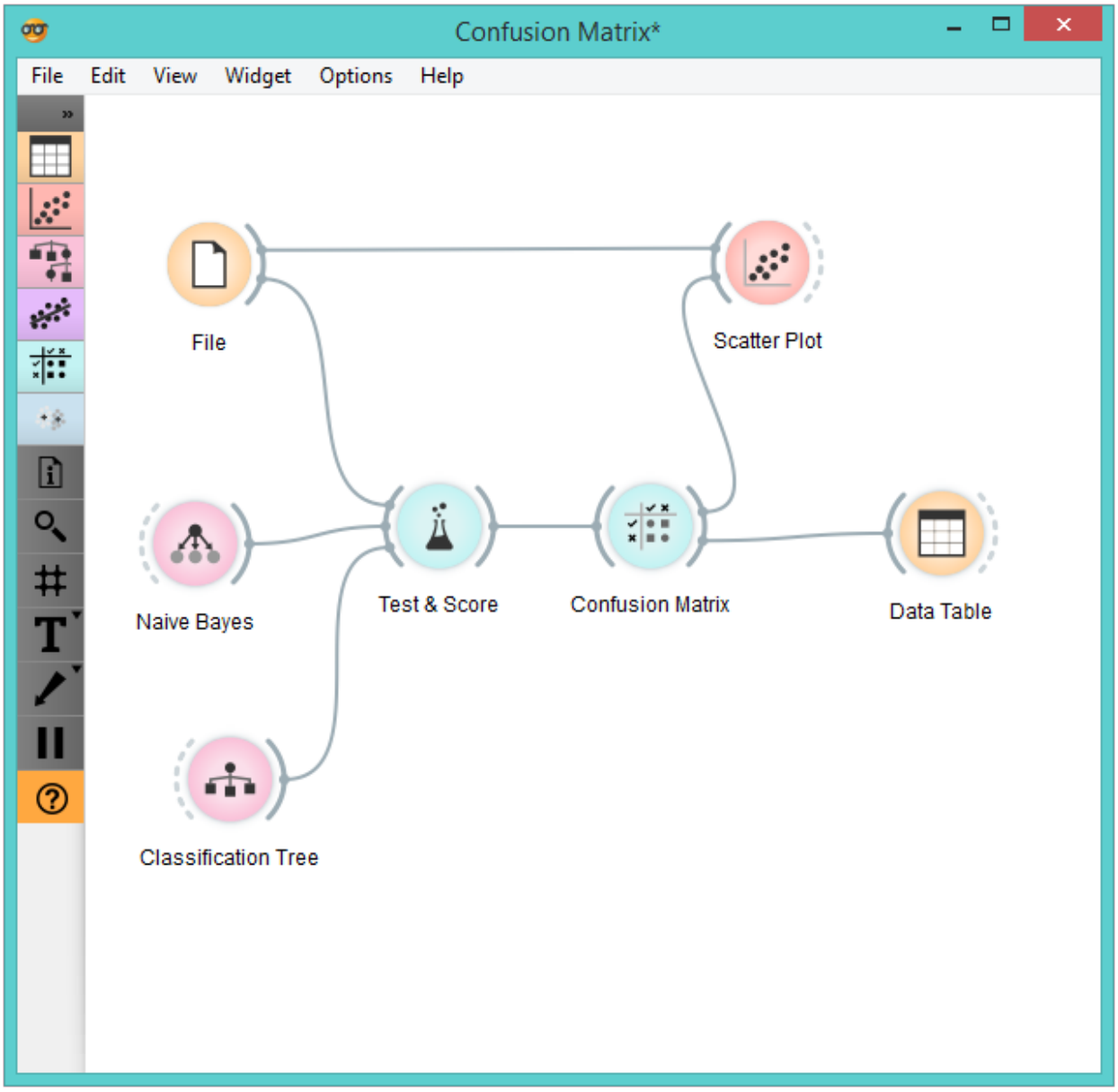
		Predicted			
		Iris-setosa	Iris-versicolor	Iris-virginica	Σ
Actual	Iris-setosa	100.0 %	0.0 %	0.0 %	50
	Iris-versicolor	0.0 %	88.7 %	6.4 %	50
	Iris-virginica	0.0 %	11.3 %	93.6 %	50
	Σ	50	53	47	150

ors 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).
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

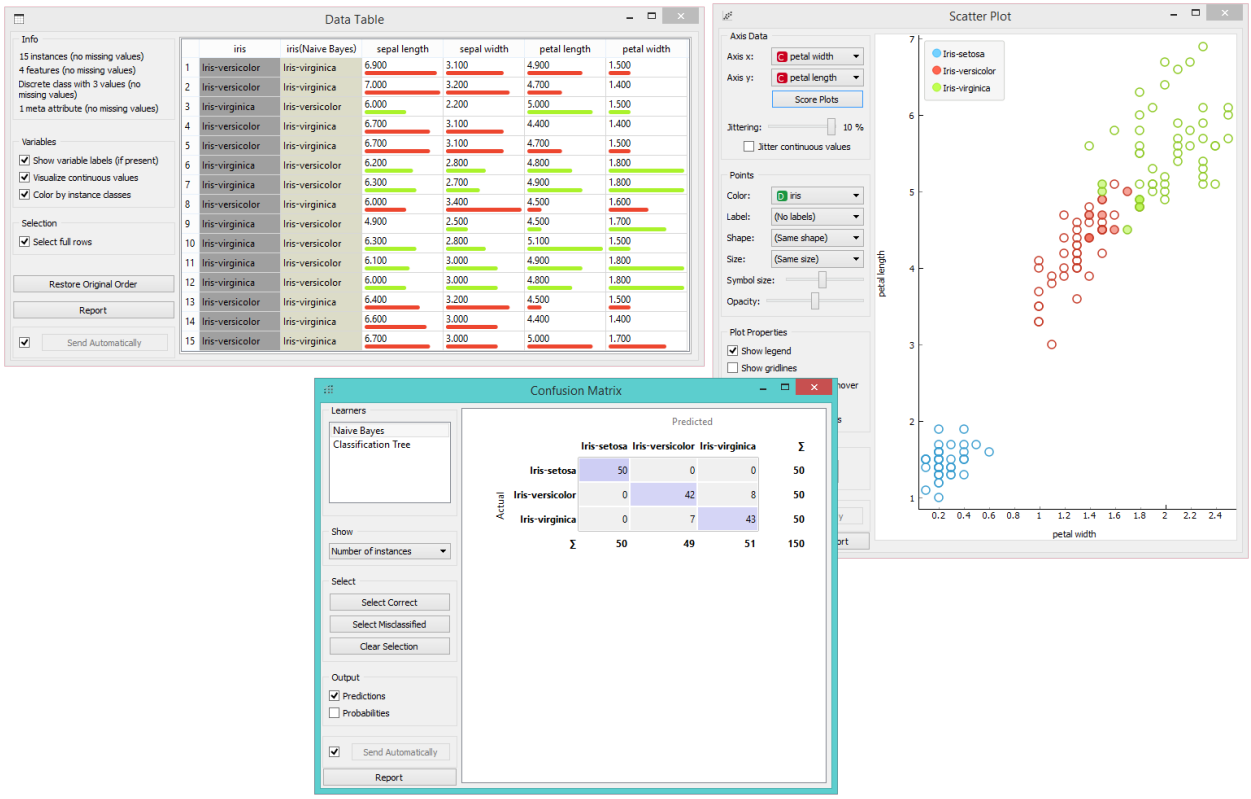
The following workflow demonstrates what this widget can be used for.



Test & Score gets the data from File and two learning algorithms from Naive Bayes and 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 Scatter Plot 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.



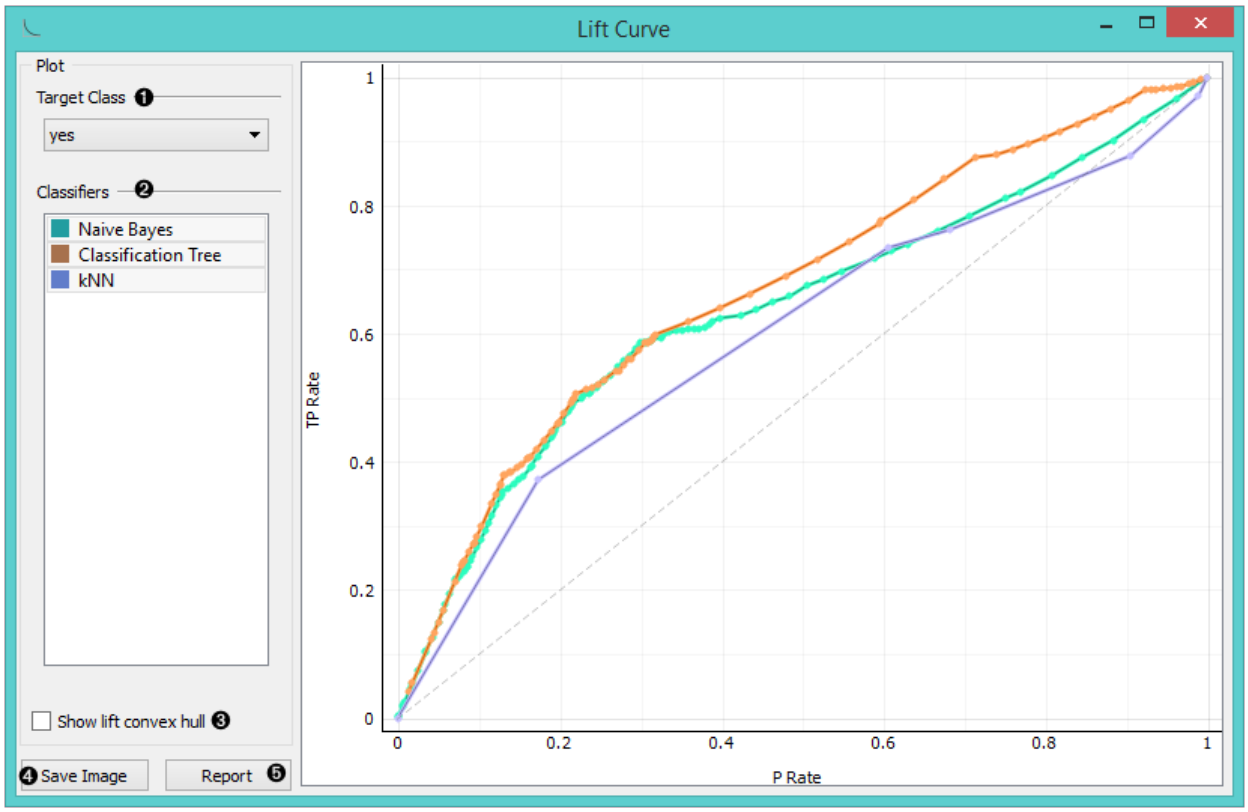
2.5.3 Lift Curve

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

Inputs

- Evaluation Results: results of testing classification algorithms

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.



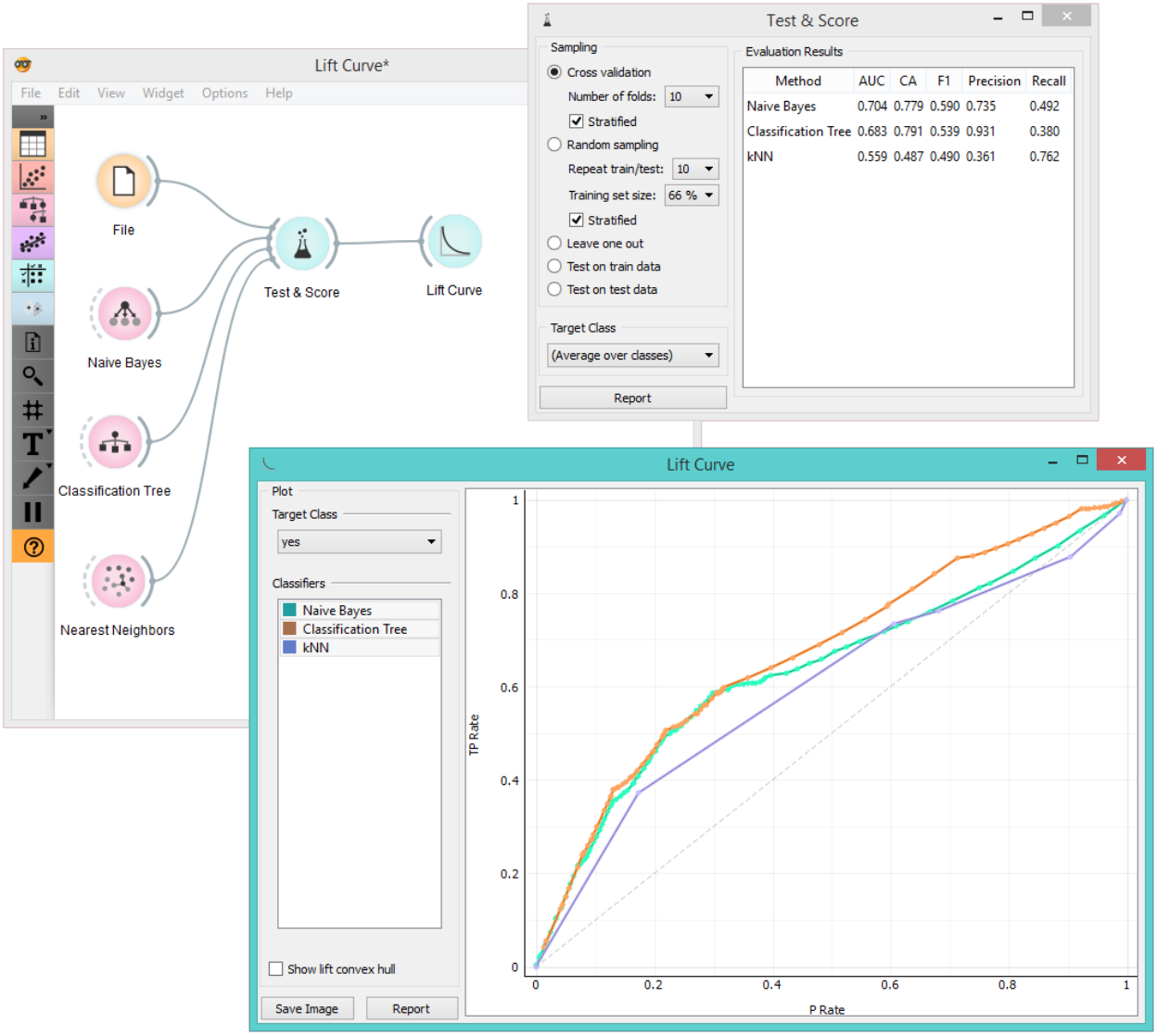
1. Choose the desired *Target class*. The default class is chosen alphabetically.
2. If test results contain more than one classifier, the user can choose which curves she or he wants to see plotted. Click on a classifier to select or deselect the curve.
3. *Show lift convex hull* plots a convex hull 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 behavior 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* dataset. 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 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.

2.5.4 Predictions

Shows models' predictions on the data.

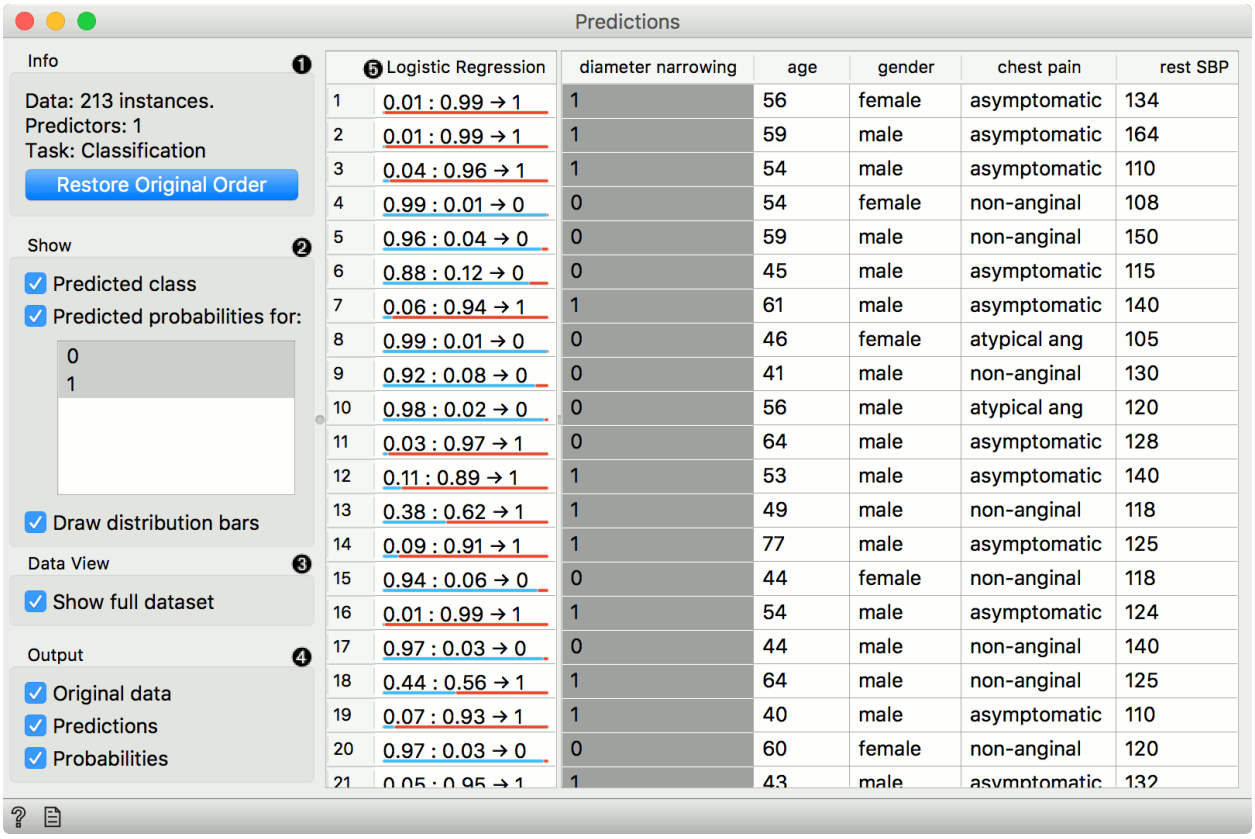
Inputs

- Data: input dataset
- Predictors: predictors to be used on the data

Outputs

- Predictions: data with added predictions
- Evaluation Results: results of testing classification algorithms

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



1. Information on the input, namely the number of instances to predict, the number of predictors and the task (classification or regression). If you have sorted the data table by attribute and you wish to see the original view, press *Restore Original Order*.
2. You can select the options for classification. If *Predicted class* is ticked, the view provides information on predicted class. If *Predicted probabilities for* is ticked, the view provides information on probabilities predicted by the classifier(s). You can also select the predicted class displayed in the view. The option *Draw distribution bars* provides a visualization of probabilities.
3. By ticking the *Show full dataset*, you can view the entire data table (otherwise only class variable will be shown).
4. Select the desired output.
5. Predictions.

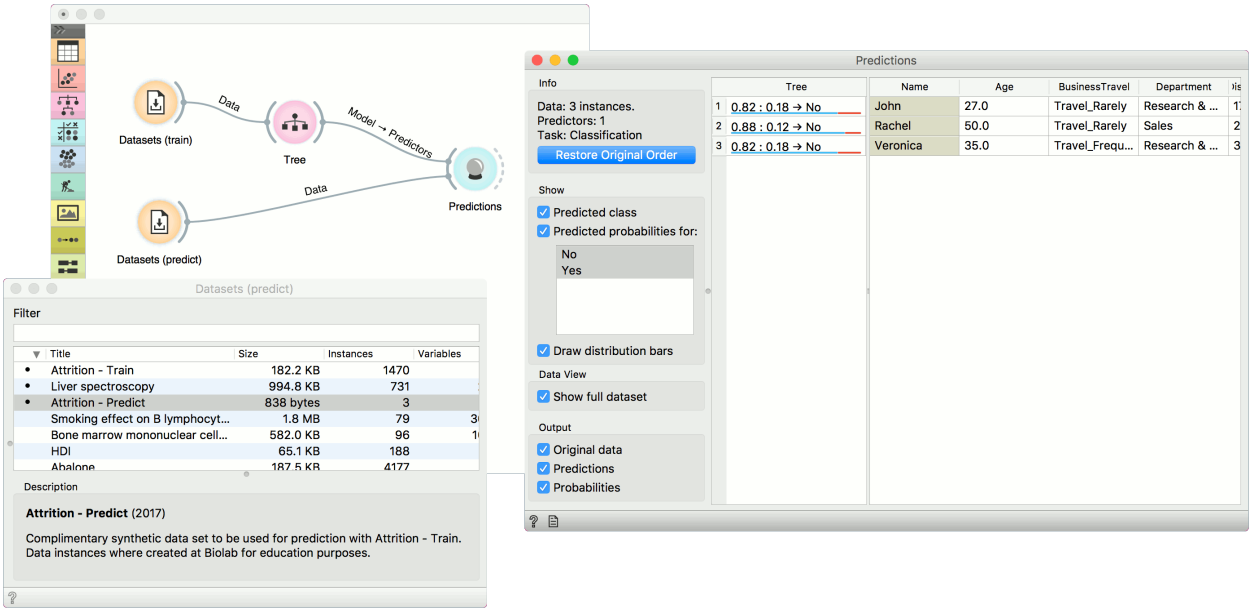
The widget show the probabilities and final decisions of predictive models. The output of the widget is another dataset, where predictions are appended as new meta attributes. You can select which features you wish to output (original data, predictions, probabilities). The result can be observed in a Data Table. If the predicted data includes true class values, the result of prediction can also be observed in a Confusion Matrix.

Examples

In the first example, we will use *Attrition - Train* data from the Datasets widget. This is a data on attrition of employees. In other words, we wish to know whether a certain employee will resign from the job or not. We will construct a predictive model with the Tree widget and observe probabilities in **Predictions**.

For predictions we need both the training data, which we have loaded in the first **Datasets** widget and the data to predict, which we will load in another Datasets widget. We will use *Attrition - Predict* data this time. Connect the second data set to **Predictions**. Now we can see predictions for the three data instances from the second data set.

The Tree model predicts none of the employees will leave the company. You can try other model and see if predictions change. Or test the predictive scores first in the Test & Score widget.



In the second example, we will see how to properly use Preprocess with **Predictions** or Test & Score.

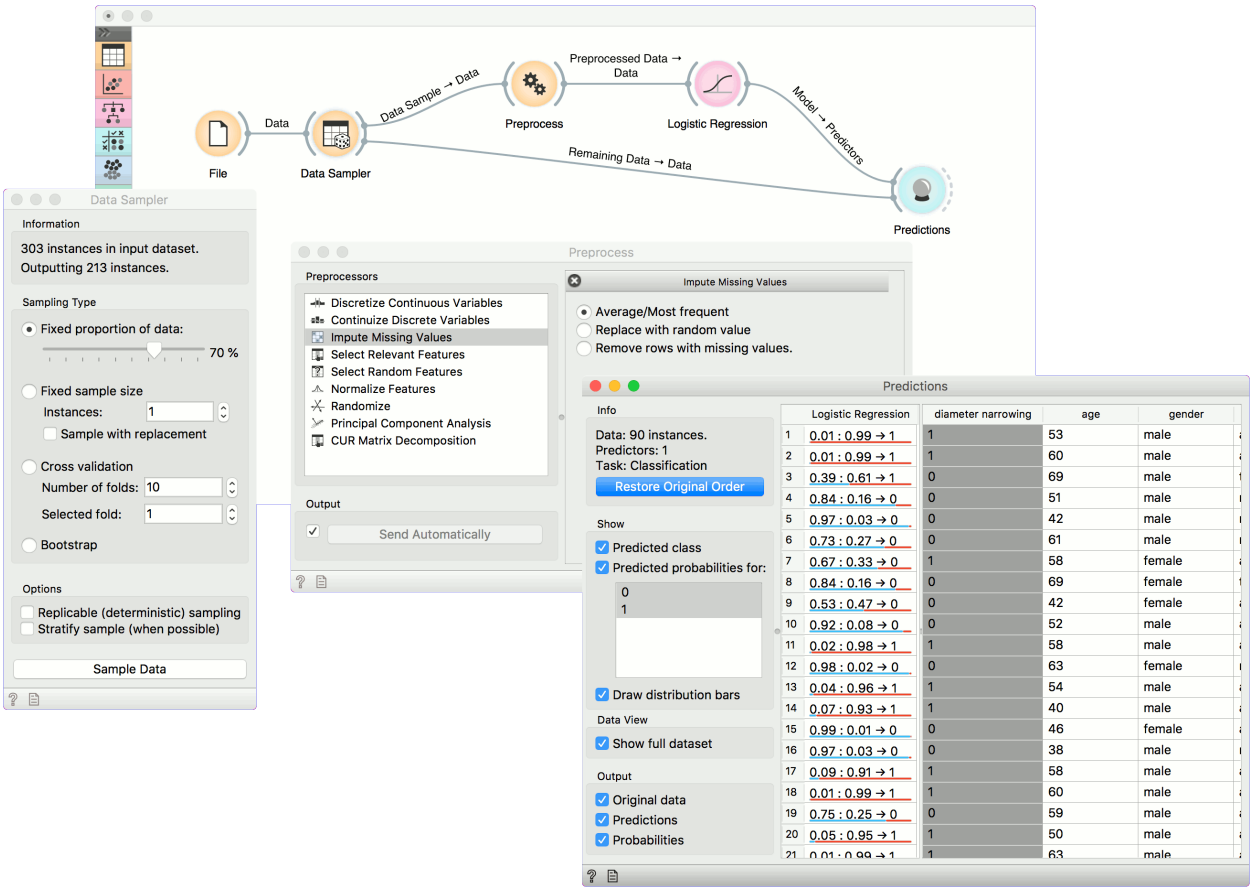
This time we are using the *heart disease.tab* data from the File widget. You can access the data through the dropdown menu. This is a dataset with 303 patients that came to the doctor suffering from a chest pain. After the tests were done, some patients were found to have diameter narrowing and others did not (this is our class variable).

The heart disease data have some missing values and we wish to account for that. First, we will split the data set into train and test data with Data Sampler.

Then we will send the *Data Sample* into Preprocess. We will use *Impute Missing Values*, but you can try any combination of preprocessors on your data. We will send preprocessed data to Logistic Regression and the constructed model to **Predictions**.

Finally, **Predictions** also needs the data to predict on. We will use the output of Data Sampler for prediction, but this time not the *Data Sample*, but the *Remaining Data*, this is the data that wasn't used for training the model.

Notice how we send the remaining data directly to **Predictions** without applying any preprocessing. This is because Orange handles preprocessing on new data internally to prevent any errors in the model construction. The exact same preprocessor that was used on the training data will be used for predictions. The same process applies to Test & Score.



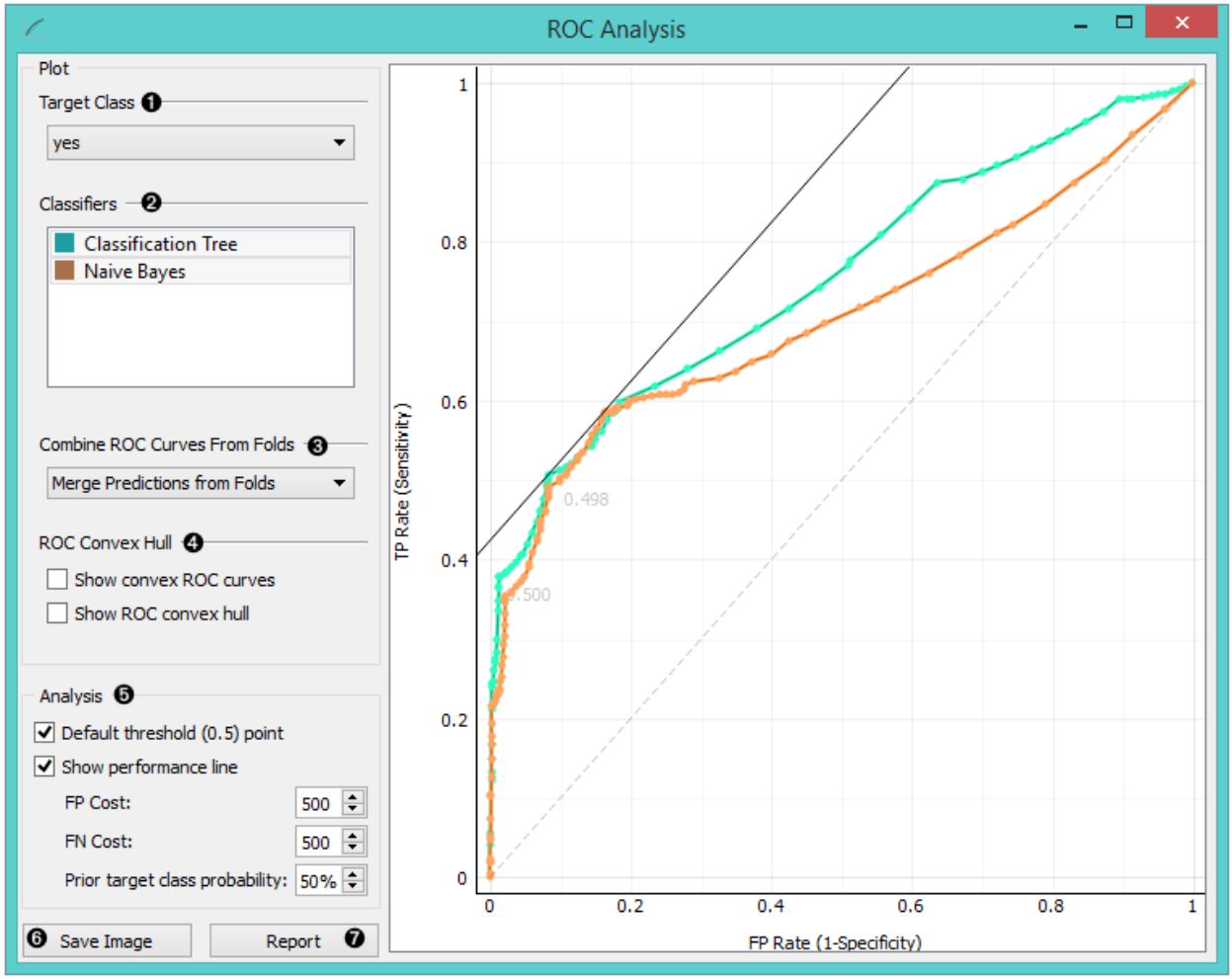
2.5.5 ROC Analysis

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

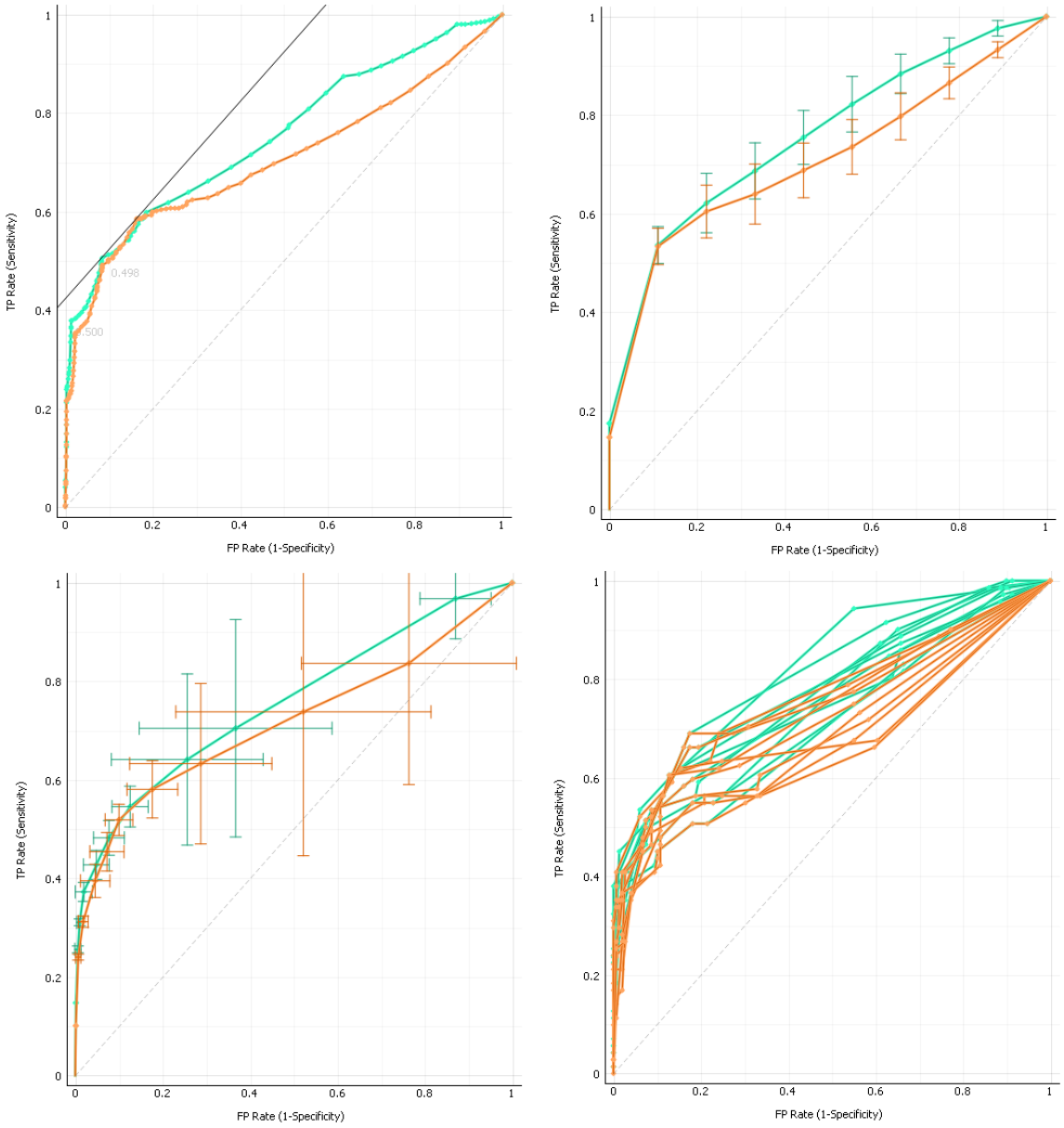
Inputs

- Evaluation Results: results of testing classification algorithms

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.



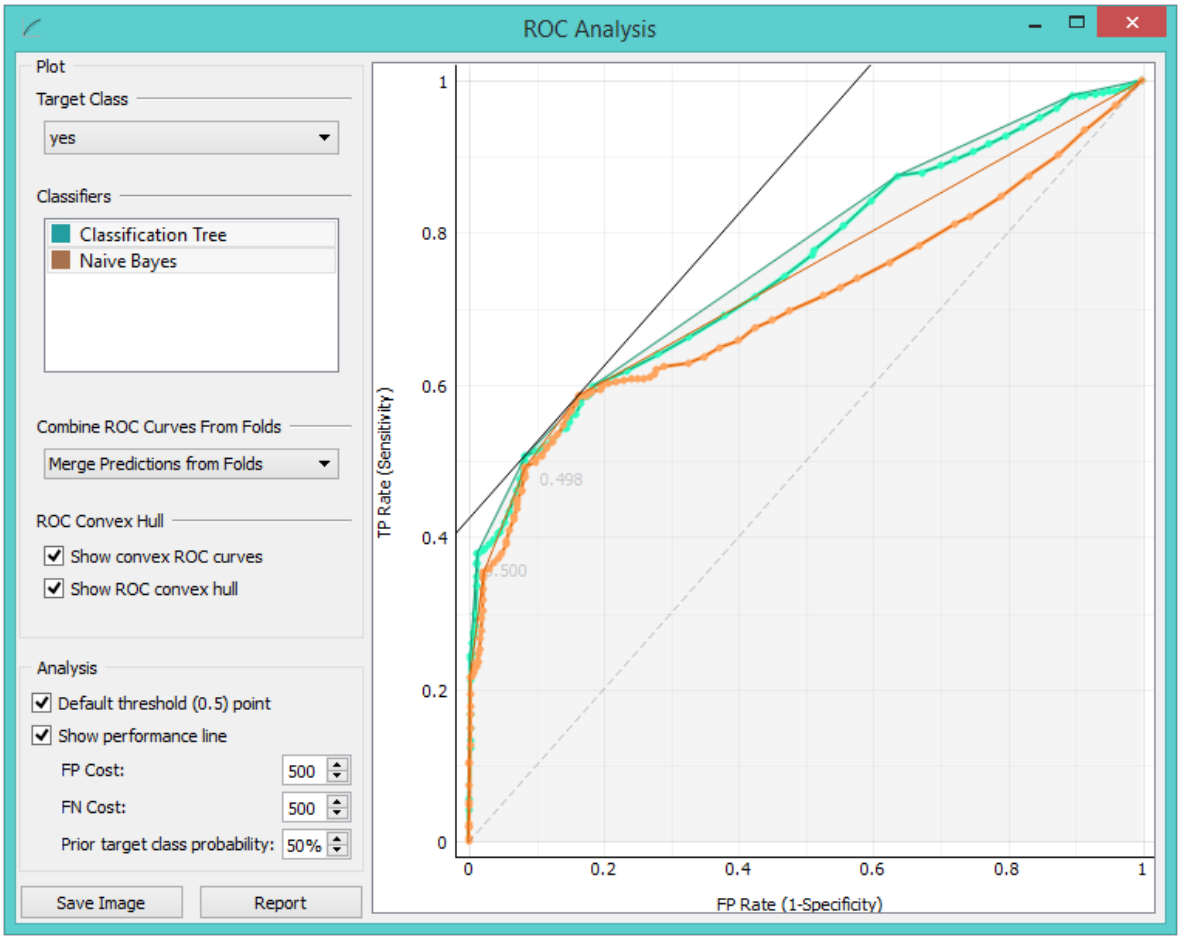
1. Choose the desired *Target Class*. The default class is chosen alphabetically.
2. If test results contain more than one classifier, the user can choose which curves she or he wants to see plotted. Click on a classifier to select or deselect it.
3. When the data comes from multiple iterations of training 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

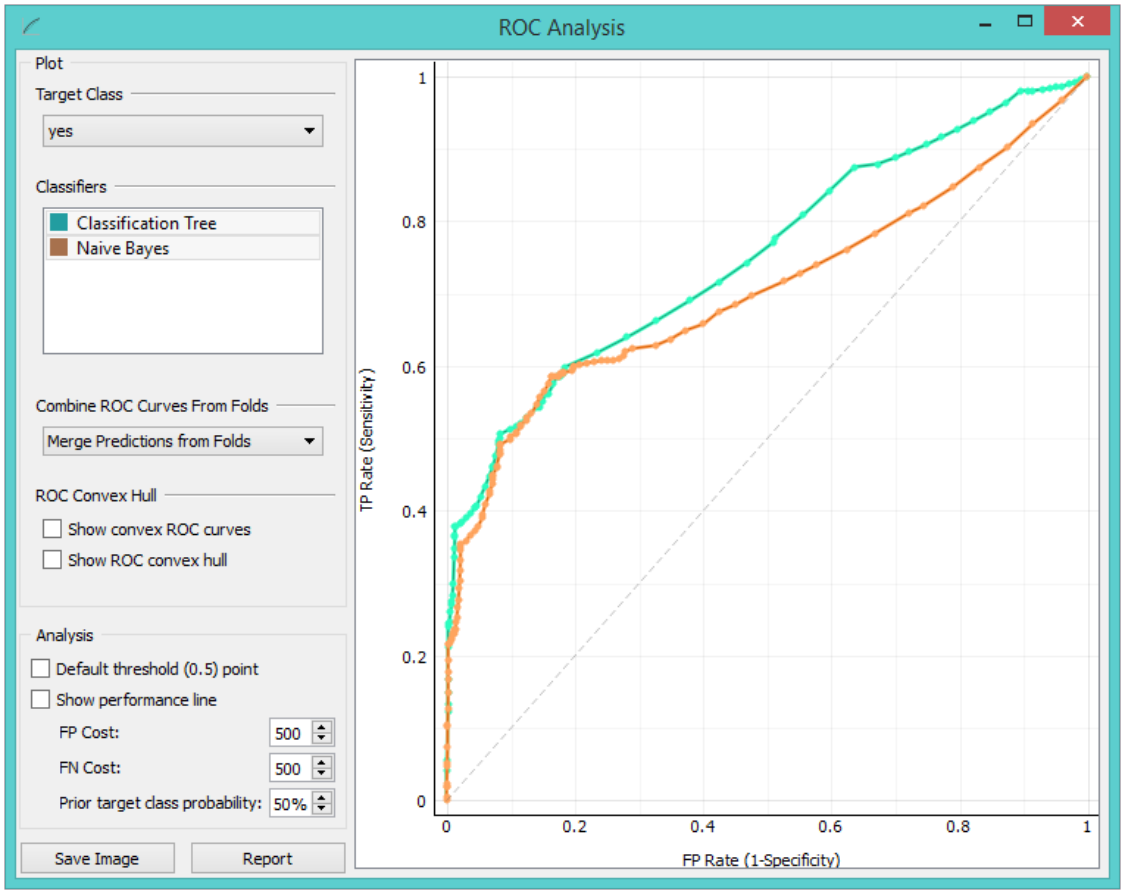
4. Option *Show convex ROC curves* refers to convex curves over each individual classifier (the thin lines positioned over curves). *Show ROC convex hull* plots a convex hull combining all classifiers (the gray area below the curves). Plotting both types of convex curves makes sense since selecting a 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.



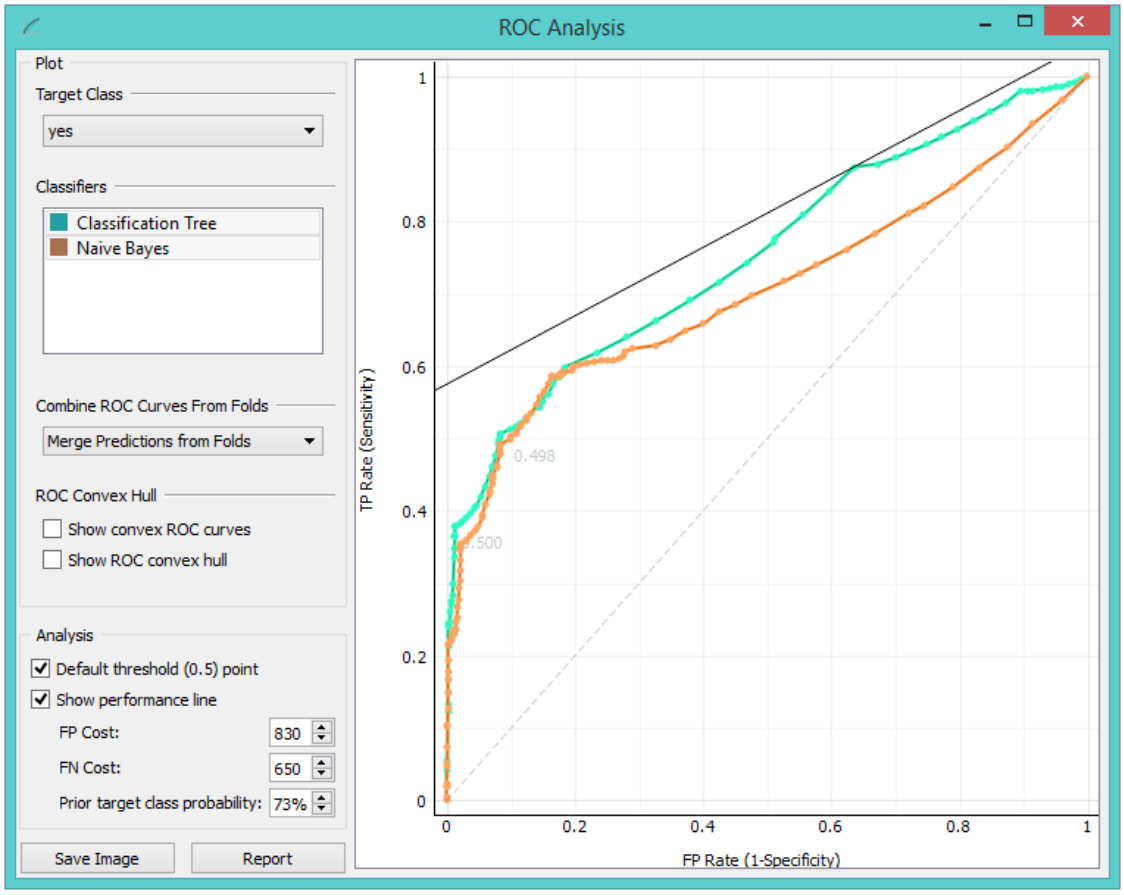
The diagonal dotted line represents the behavior of a random classifier. The full diagonal line represents iso-performance. A black "*A*" symbol at the bottom of the graph proportionally readjusts the graph.

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 Tree and Naive Bayes, in **Test&Score** and then compare their performance in **ROC Analysis**, Life Curve and Calibration Plot.

2.5.6 Test and Score

Tests learning algorithms on data.

Inputs

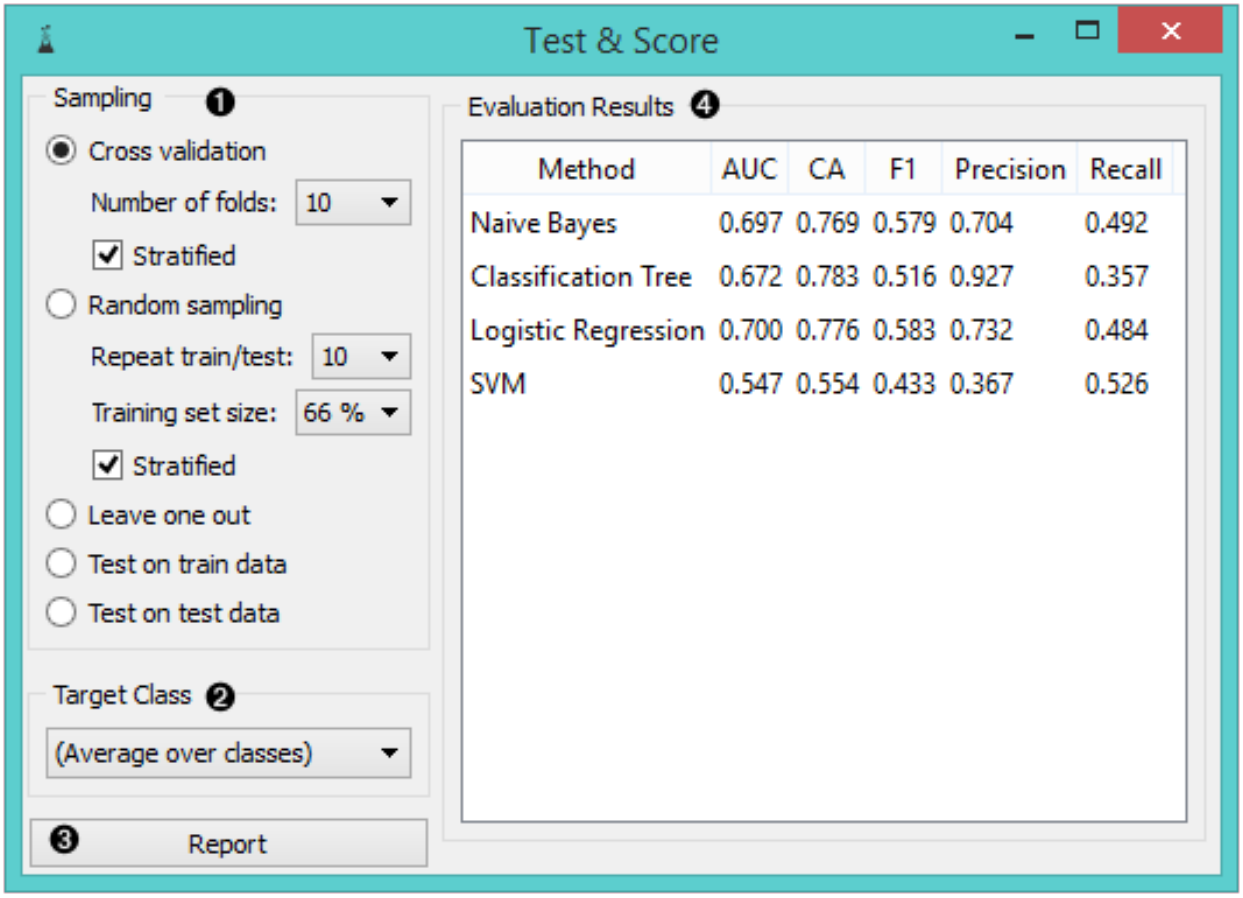
- Data: input dataset
- Test Data: separate data for testing
- Learner: learning algorithm(s)

Outputs

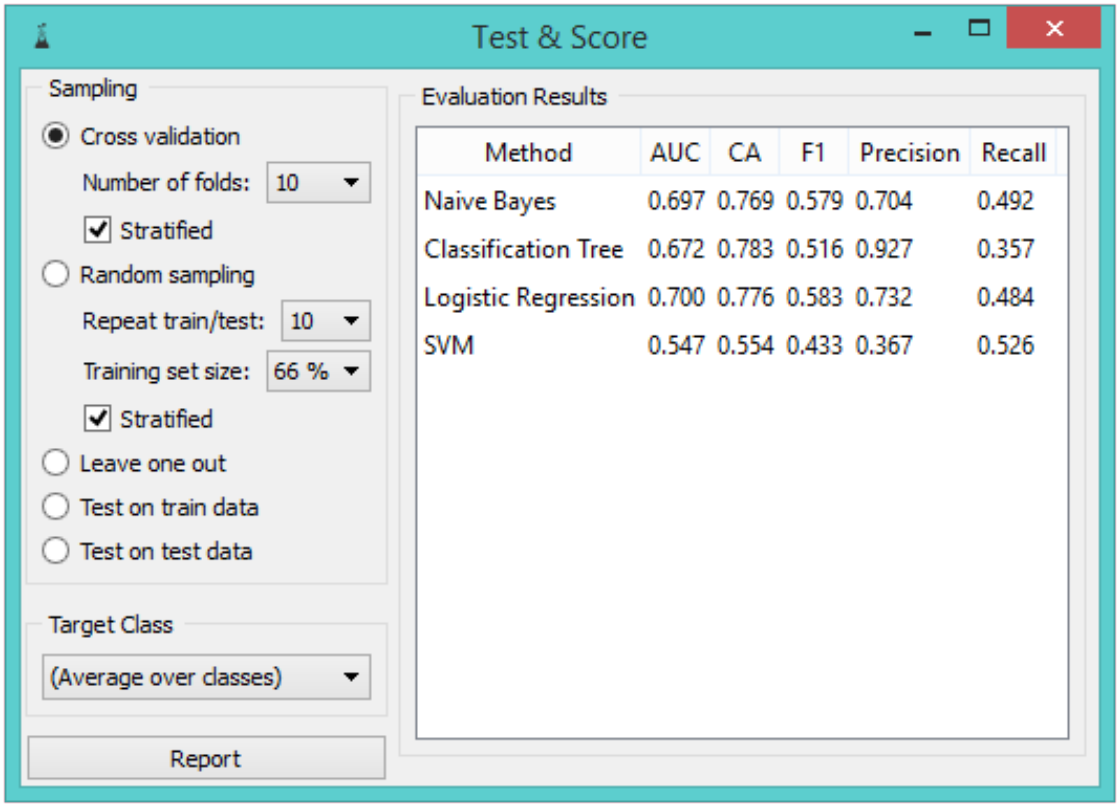
- Evaluation Results: results of testing classification algorithms

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 the curve. 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.

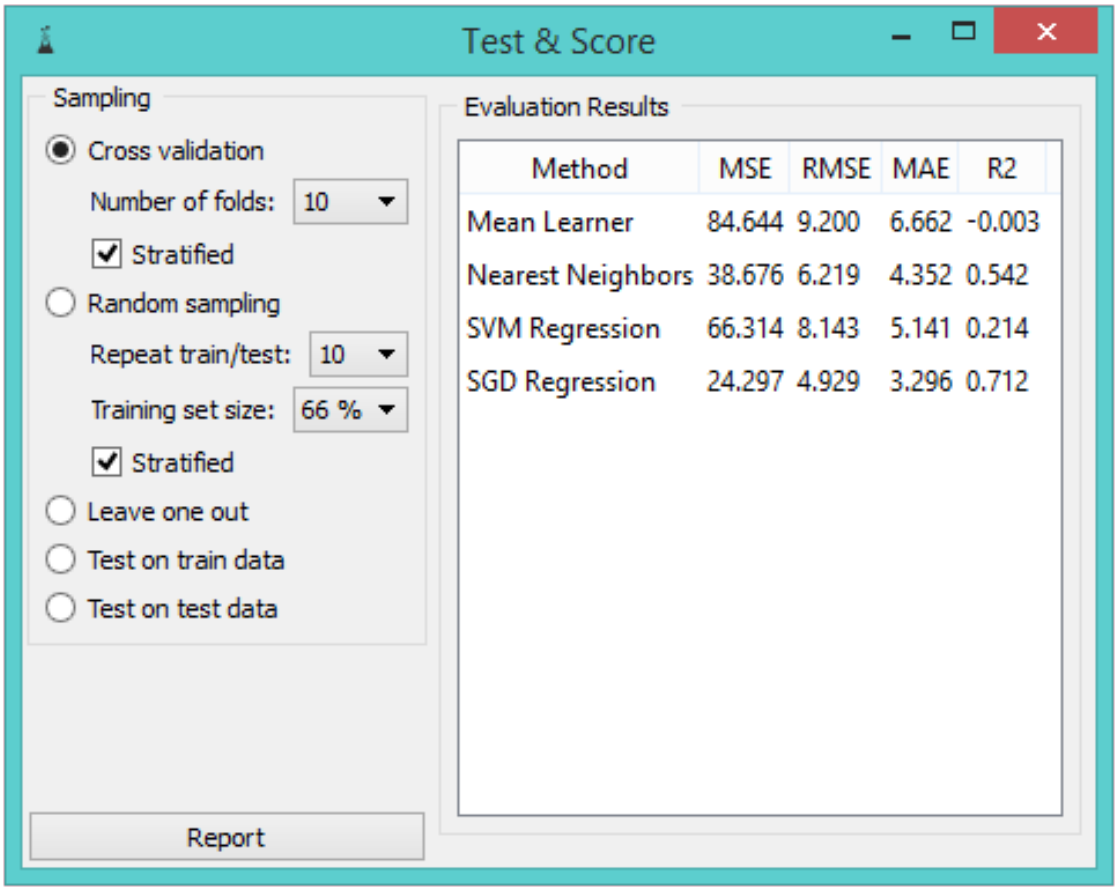
The *Learner* signal has an uncommon property: it can be connected to more than one widget to test multiple learners with the same procedures.



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 dataset 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 dataset 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. For classification, *Target class* can be selected at the bottom of the widget. When *Target class* is (Average over classes), methods return scores that are weighted averages over all classes. For example, in case of the classifier with 3 classes, scores are computed for class 1 as a target class, class 2 as a target class, and class 3 as a target class. Those scores are averaged with weights based on the class size to retrieve the final score.
3. Produce a report.
4. The widget will compute a number of performance statistics:



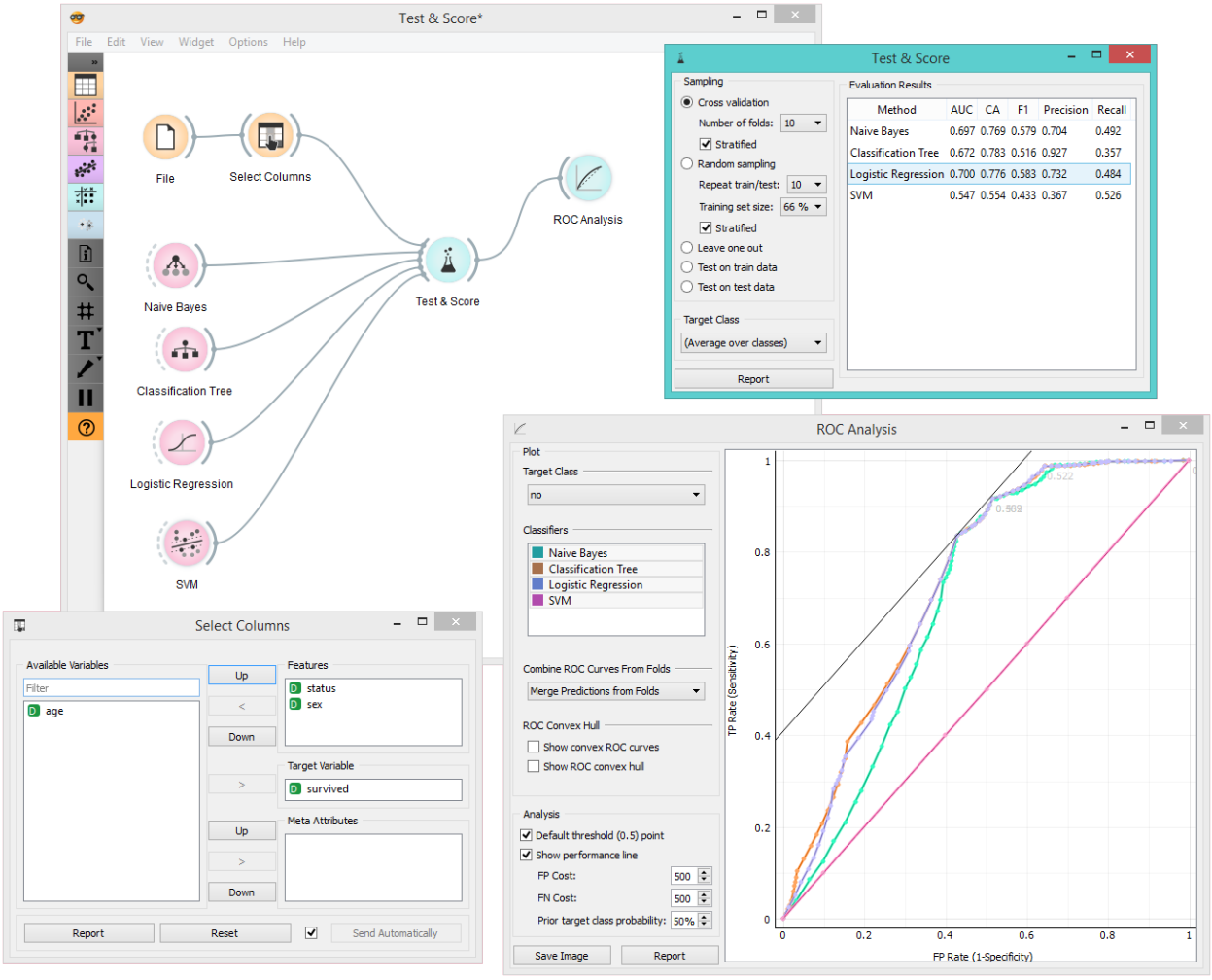
- Classification
  - Area under ROC is the area under the receiver-operating curve.
  - Classification accuracy is the proportion of correctly classified examples.
  - F-1 is a weighted harmonic mean of precision and recall (see below).
  - Precision is the proportion of true positives among instances 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
  - 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.
  - R2 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 dataset 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* dataset, 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.