

BIOLAB AND COLLABORATORS

# DATA SCIENCE FOR ARCHAEOLOGY

BIOLAB

Copyright © 2021 Biolab and Collaborators

PUBLISHED BY BIOLAB

TUFTE-LATEX.GOOGLECODE.COM

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at <http://www.apache.org/licenses/LICENSE-2.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

*First printing, March 2021*

# *Contents*

<i>Workflows in Orange</i>	7
<i>Basic data exploration</i>	11
<i>Saving your work</i>	14
<i>Loading data sets</i>	15
<i>Assignment: Data Preprocessing</i>	17
<i>Assignment: Geolocations</i>	19
<i>Hierarchical Clustering</i>	21
<i>Animal Kingdom</i>	23
<i>k-Means Clustering</i>	24
<i>Silhouettes</i>	27
<i>Principal Component Analysis</i>	30
<i>Assignment: Hierarchical Clustering</i>	35
<i>Classification</i>	38

<i>Classification Trees</i>	39
<i>Classification Accuracy</i>	42
<i>Random Forests</i>	43
<i>Assignment: Clustering vs. Classification</i>	44
<i>Naive Bayes</i>	45
<i>Logistic Regression</i>	46
<i>Support Vector Machines</i>	48
<i>k-Nearest Neighbors</i>	49
<i>Assignment: Decision Boundaries</i>	50
<i>Assignment: Model Explanation</i>	51
<i>How to Cheat</i>	52
<i>Cross-Validation</i>	55
<i>Assignment: Overfitting</i>	56
<i>Linear Regression</i>	57
<i>Regularization</i>	60
<i>Image Embedding</i>	62

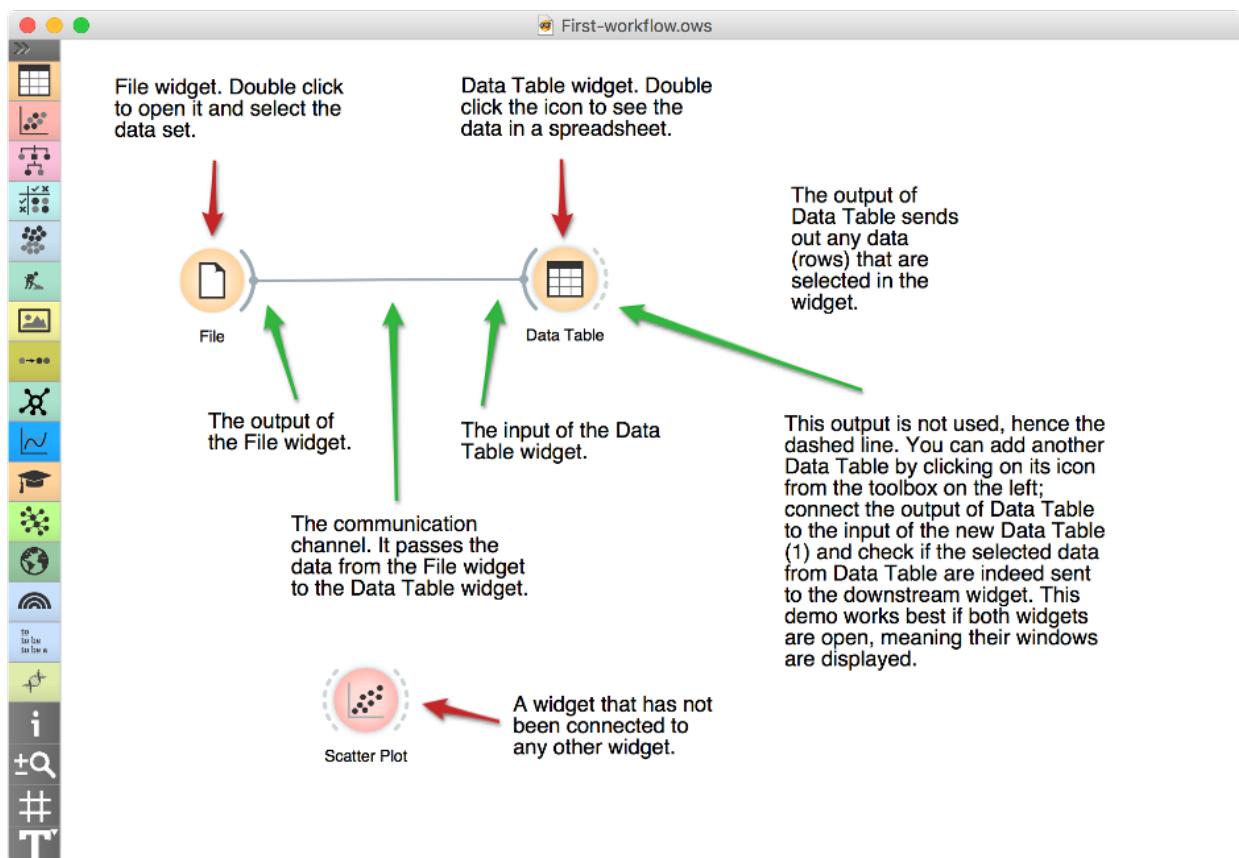
*Assignment: Image Analytics* 65

*Bibliography* 67



# Workflows in Orange

ORANGE WORKFLOWS consist of components that read, process and visualize data. We call them "widgets". Widgets are placed on a drawing board (the "canvas"). Widgets communicate by sending information along a communication channel. Output from one widget is used as input to another.

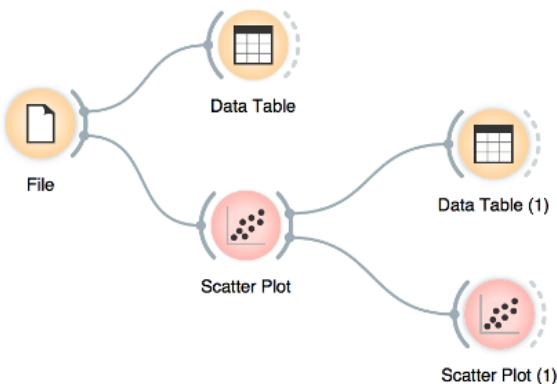


We construct workflows by dragging widgets onto the canvas and connecting them by drawing a line from the transmitting widget to the receiving widget. The widget's outputs are on the right and the inputs on the left. In the workflow above, the *File* widget sends data to the *Data Table* widget.

A simple workflow with two connected widgets and one widget without connections. The outputs of a widget appear on the right, while the inputs appear on the left.

Start by constructing a workflow that consists of a File widget, two *Scatter Plot* widgets and two Data Table widgets:

Workflow with a File widget that reads the data from a disk and sends it to the Scatter Plot and Data Table widget. The Data Table renders the data in a spreadsheet, while the Scatter Plot visualizes it. Selected data points from the plot are sent to two other widgets: Data Table (1) and Scatter Plot (1).



The File widget reads data from your local disk. Open the File widget by double clicking its icon. Orange comes with several pre-loaded data sets. From these ("Browse documentation data sets..."), choose *brown-selected.tab*, a yeast gene expression data set.

Orange workflows often start with a File widget. The brown-selected data set comprises 186 rows (genes) and 81 columns. Out of the 81 columns, 79 contain gene expressions of baker's yeast under various conditions, one column (marked as a "meta attribute") provides gene names, and one column contains the "class" value or gene function.

**File**

File: brown-selected.tab

URL:

**Info**

**Brown dataset**  
Gene expression of baker's yeast.

186 instance(s)  
79 feature(s) (1.5% missing values)  
Classification; categorical class with 3 values (no missing values)  
1 meta attribute(s)

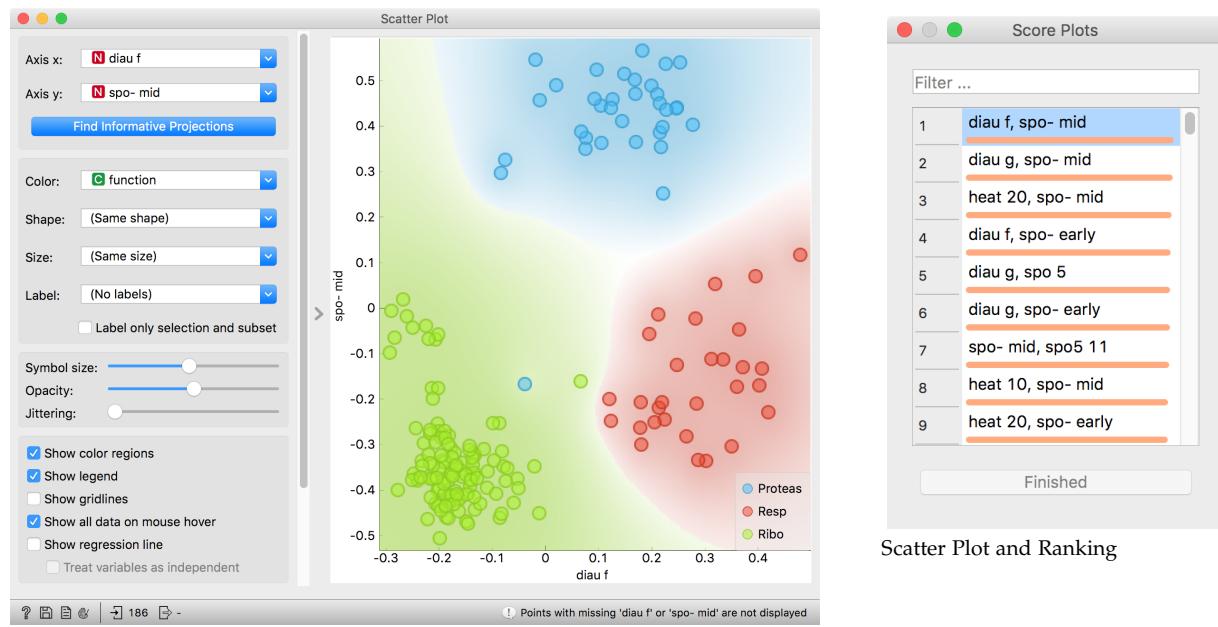
**Columns (Double click to edit)**

	Name	Type	Role	Values
/6	diau d	N numeric	feature	
77	diau e	N numeric	feature	
78	diau f	N numeric	feature	
79	diau g	N numeric	feature	
80	function	C categorical	target	Proteas, Resp, Ribo
81	gene	S text	meta	

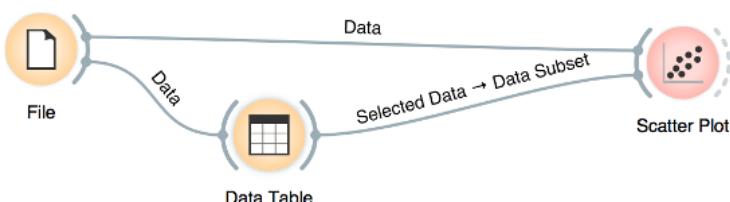
?  186

After you load the data, open the other widgets. In the Scatter Plot widget, select a few data points and watch as they appear in the Data Table (1). Use a combination of two Scatter Plot widgets, where the second scatter plot shows a detail from a smaller region selected in the first scatter plot.

The following is more of a side note, but it won't hurt. The scatter plot for a pair of random features does not provide much information on gene function. Does this change with a different choice of feature pairs in the visualization? *Rank projections* (the button on the top left of the Scatter Plot widget) can help you find a good feature pair. How do you think this works? Could the suggested pairs of features be useful to a biologist?

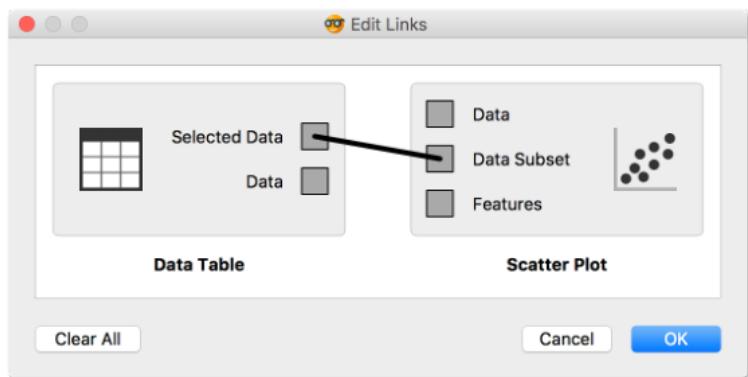


We can connect the output of the Data Table widget to the Scatter Plot widget to highlight the chosen data instances (rows) in the scatter plot.



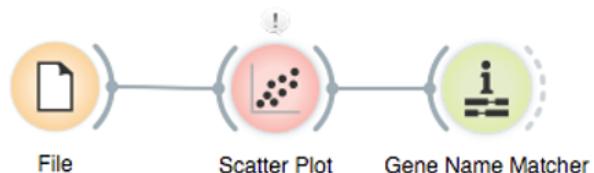
In this workflow, we have switched on the option "Show channel names between widgets" in File/Preferences.

How does Orange distinguish between the primary data source and the data selection? It uses the first connected signal as the entire data set and the second one as its subset. To make changes or to check what is happening under the hood, double click on the line connecting the two widgets.



The rows in the data set we are exploring in this lesson are gene profiles. We could perhaps use widgets from the Bioinformatics add-on to get more information on the genes we selected in any of the Orange widgets.

Orange comes with a basic set of widgets for data input, preprocessing, visualization and modeling. For other tasks, like text mining, network analysis, and bioinformatics, there are addons. Check them out by selecting Add-ons... from the Options menu.

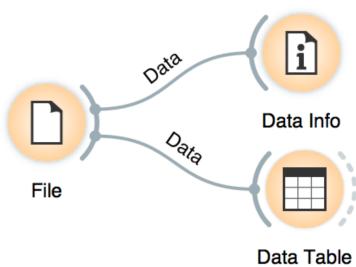


# *Basic data exploration*

LET US CONSIDER ANOTHER PROBLEM, this time from clinical medicine. We will dig for something interesting in the data and explore it with visualization widgets. You will get to know Orangebetter, and also learn about several interesting visualizations.

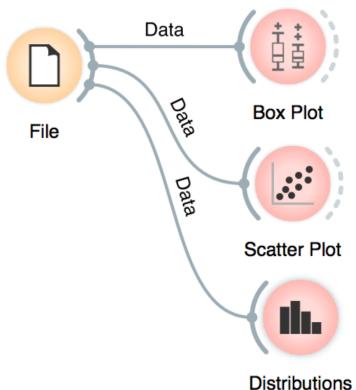
We will start with an empty canvas; to clean it from our previous lesson, use either File/New or select all the widgets and remove them (use the backspace/delete key).

Now again, add the File widget and open another documentation data set: heart\_disease. How does the data look like?



A simple workflow to inspect the loaded dataset.

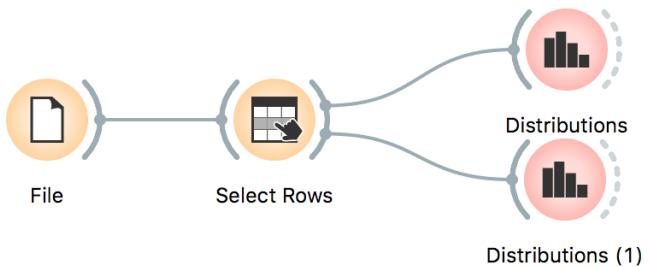
Let us check whether common visualizations tell us anything interesting. (Hint: look for gender differences. These are always interesting and occasionally even real.)



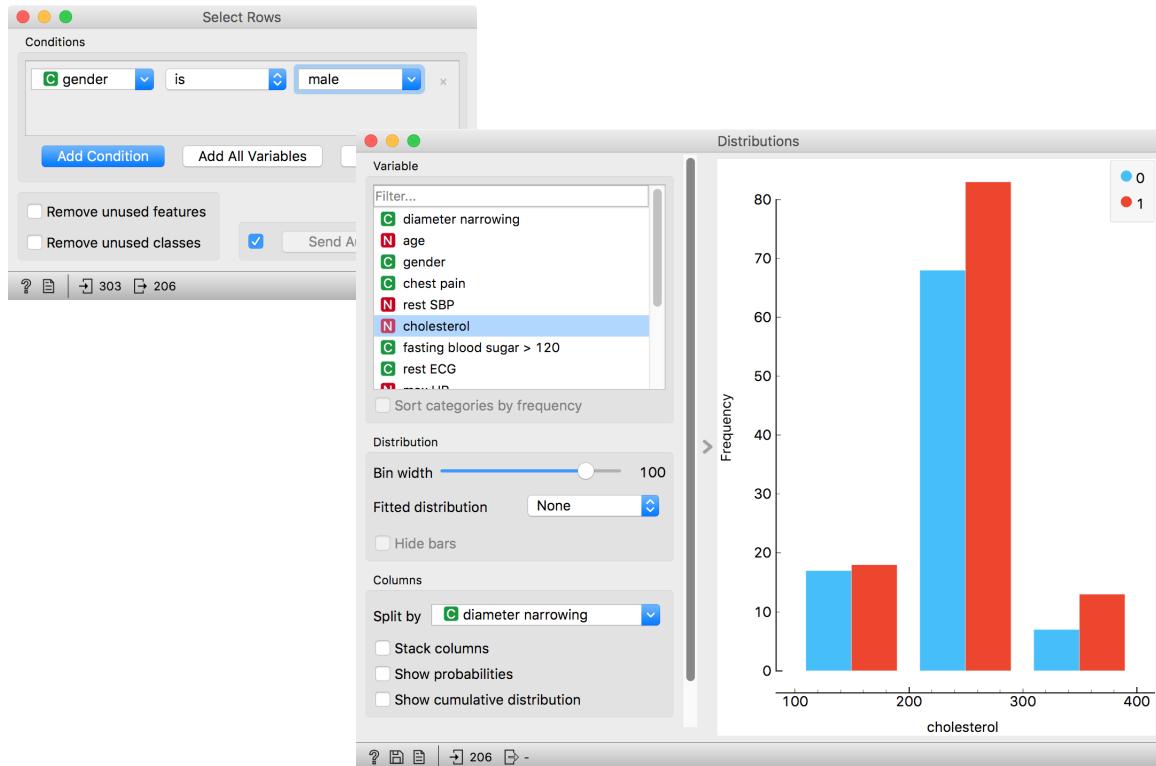
Quick check with common statistics and visualization widgets.

Data can also be split by the value of features, in this case the gender.

The two Distributions widgets get different data: the upper gets the selected rows and the lower gets the rest. Double-click the connection between the widgets to access setup dialog, as you've learned in the previous lesson.



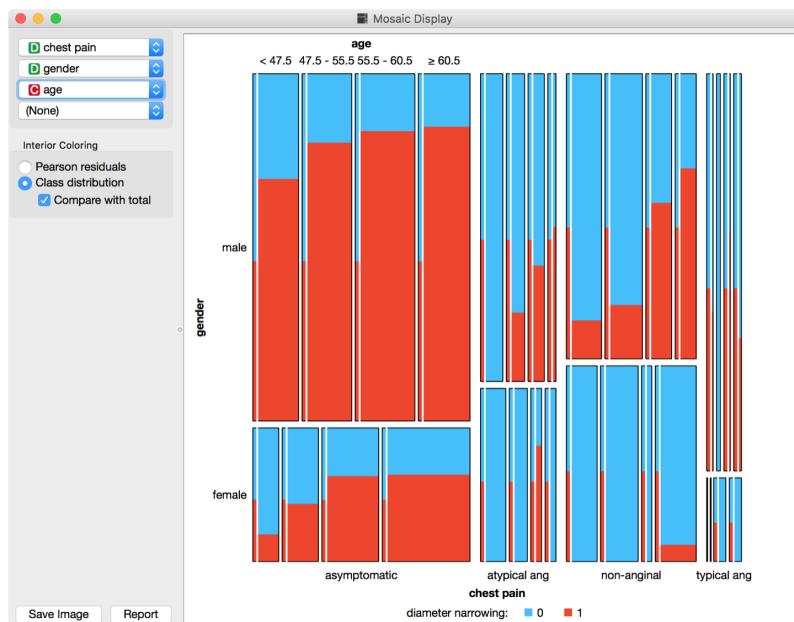
In the Select Rows widget, we select the female patients. You can also add other conditions. Selection of data instances provides a powerful combination with visualization of data distribution. Try having at least two widgets open at the same time and explore the data.



There are two less known — but great — visualizations for observing interactions between features.

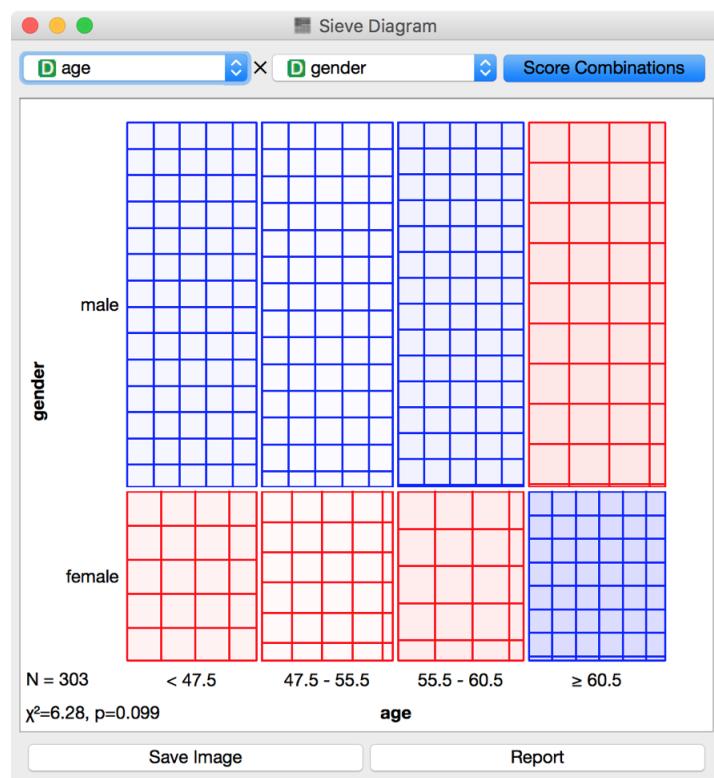
Mosaic display shows a rectangle split into columns with widths reflecting the prevalence of different types of chest pain. Each column is then further split vertically according to gender distributions within the column. The resulting rectangles are split again horizontally according to age group sizes. Within the resulting bars, the red and blue areas represent the outcome distribution for each group and the tiny strip to the left of each shows the overall distribution.

What can you read from this diagram?



You can play with the widget by trying different combinations of 1-4 features.

Another visualization, Sieve diagram also splits a rectangle horizontally and vertically, but with independent cuts, so the areas correspond to the expected number of data instances if the observed variables were independent. For instance,  $1/4$  of patients are older than 60, and  $1/3$  of patients are female, so the area of the bottom right rectangle is  $1/12$  of the total area. With roughly 300 patients, we would expect  $1/12 \times 300 = 25$  older women in our data. As a matter of fact, there are 34. Sieve diagram shows the difference between the expected and the observed frequencies by the grid density and the color of the field.

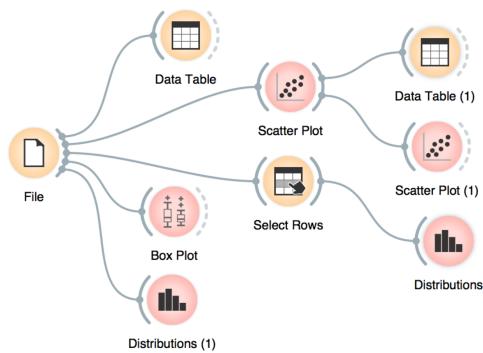


See the Score Combinations button? Guess what it does? And how it scores the combinations? (Hint: there are some Greek letters at the bottom of the widget.)

# *Saving your work*

AT THE END OF A LESSON, your workflow may look like this:

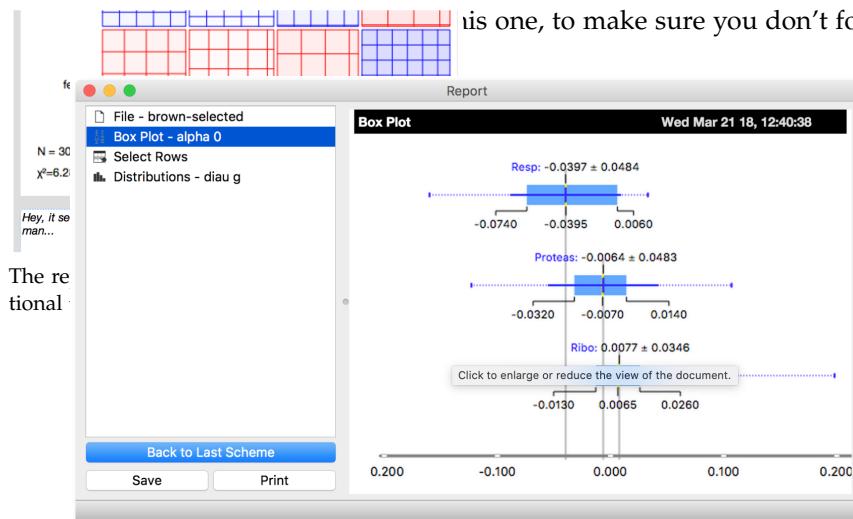
A fairly complex workflow that you would want to share or reuse at a later time.



You can save this workflow, otherwise called a "*Schema*" using the File/Save menu and share it with your colleagues. Just don't forget to put the data files in the same directory as the file with the workflow.

Widgets also have a Report button in their bottom status bar, which you can use to keep a log of your analysis. When you find something interesting, just click it and the graph will be added to your log. You can also add reports from the widgets on the path to this one, to make sure you don't forget anything relevant.

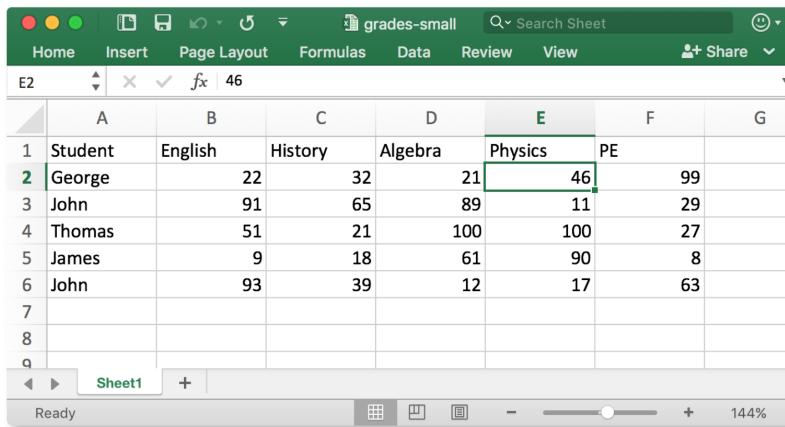
Clicking on a section of the report window allows you to add a comment.



You can save the report as HTML or PDF, or a report file that includes all workflow related report items that you can later open in Orange. In this way, you and your colleagues can reproduce your analysis results.

# Loading data sets

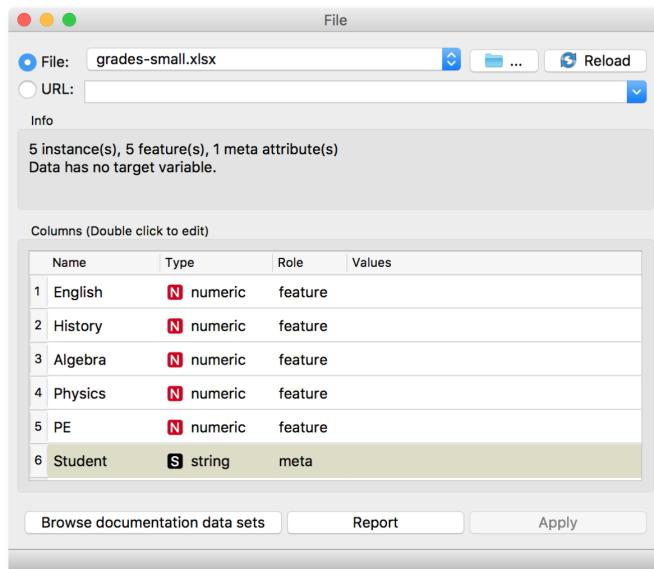
THE DATA SETS WE HAVE WORKED WITH in the previous lesson come with the Orange installation. Orange can read data from many file formats which include tab and comma separated and Excel files. To see how this works, let's prepare a data set (with school subjects and grades) in Excel and save it on a local disk.



A screenshot of Microsoft Excel showing a data table titled "grades-small". The table has columns labeled A through G. Column A is "Student", B is "English", C is "History", D is "Algebra", E is "Physics", F is "PE", and G is empty. Row 1 contains the column headers. Rows 2 through 6 contain data for five students: George, John, Thomas, James, and John. The "Physics" column (E) shows values 46, 11, 100, 61, and 17 respectively. The "PE" column (F) shows values 99, 29, 27, 8, and 63 respectively. The "Student" column (A) shows values George, John, Thomas, James, and John.

	A	B	C	D	E	F	G
1	Student	English	History	Algebra	Physics	PE	
2	George		22	32	21	46	99
3	John		91	65	89	11	29
4	Thomas		51	21	100	100	27
5	James		9	18	61	90	8
6	John		93	39	12	17	63
7							
8							
9							

In Orange, we can use, for example, the File widget to load this data set.



Make a spreadsheet in Excel with the numbers shown on the left. Of course, you can use any other editor, but remember to save your file in the *comma separated values (\*.csv)* format.

Looks good! Orange has correctly guessed that student names are character strings and that this column in the data set is special,

meant to provide additional information and not to be used for any kind of modeling (more about this in the upcoming lectures). All other columns are numeric features.

It is always good to check if all the data was read correctly. Now, you can connect the *File* widget with the *Data Table* widget,

Make the simple workflow shown on the right.



and double click on the Data Table to see the data in a spreadsheet format. Nice, everything is here.

The *Data Table* widget shows the loaded data set, you can select rows, which will appear on the output of the widget. It is also possible to do simple data visualizations. Explore the functionalities!

Instead of using Excel, we could also use Google Sheets, a free online spreadsheet alternative. Then, instead of finding the file on the local disk, we would enter its URL address to the *File* widget URL entry box.

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).

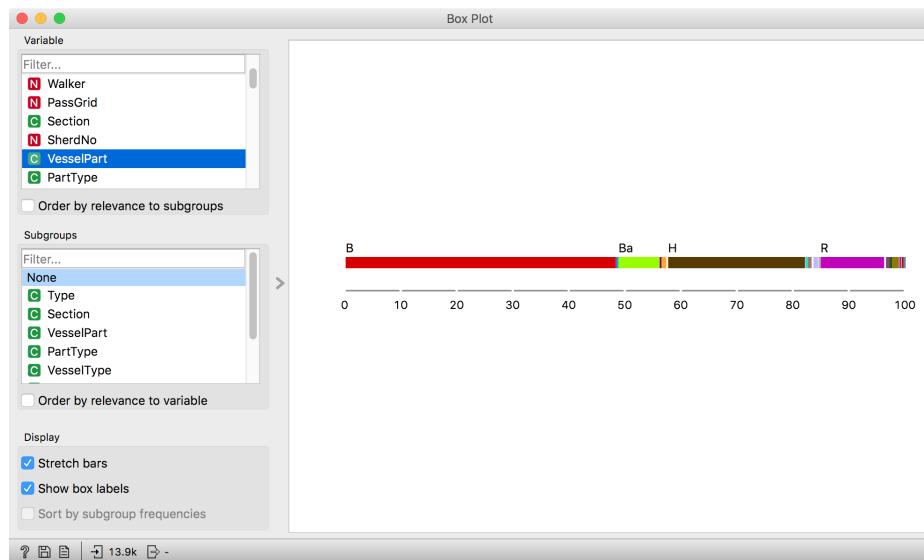
There is more to input data formatting and loading. If you would really like to dive in for more, check out the documentation page on [Loading your Data](#), or a [video tutorial](#) on this subject.

# Assignment: Data Preprocessing

DATA PREPROCESSING IS CRUCIAL IN ARCHEOLOGY. The data often contains many diverse labels, some of which are certain and others speculative. Machine learning doesn't handle categorical data with many different labels well, hence we need to preprocess the data.

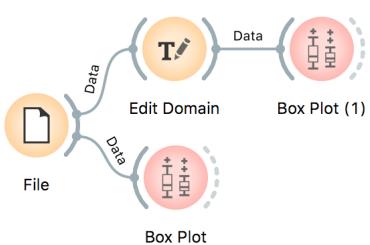
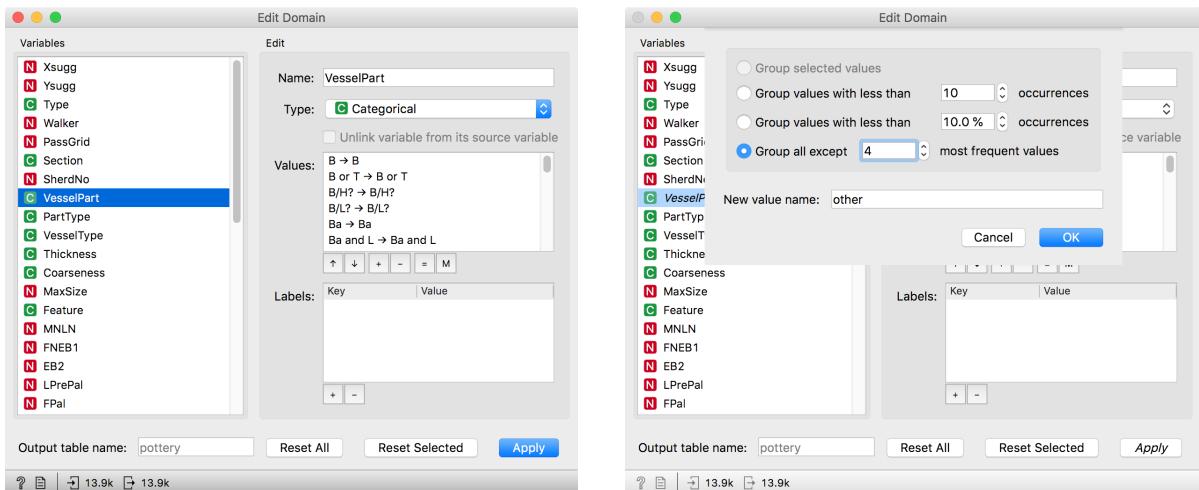
We will use the pottery data from the Antikythera data base. The data contains 13,865 data instances and 36 features describing the pottery collected across the full extent of the Greek island of Antikythera.

Let us look at the VesselPart feature, which describes the part of the ceramic vessel involved and it has 34 values.

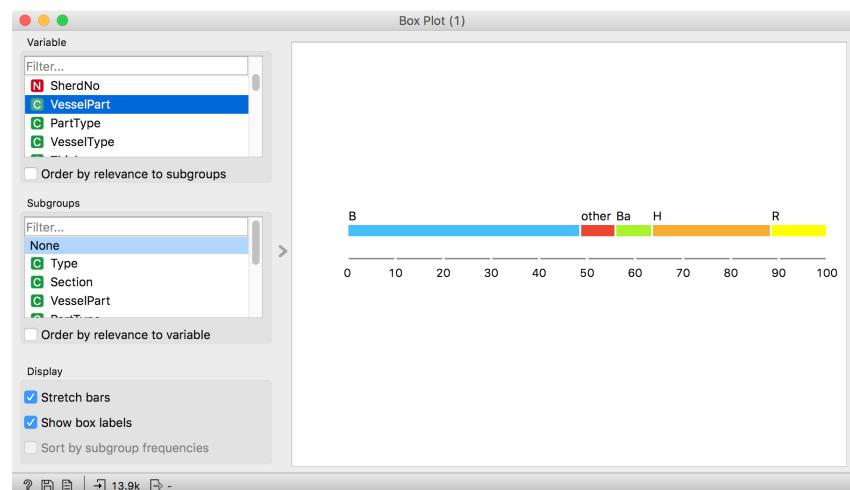


But looking at the *Box Plot*, we notice there are 4 fairly frequent values, while others are rare. We would like to keep B (body), Ba (base), H (handle), and R (rim), and consider all the others as one group.

We will use *Edit Domain* to merge infrequent values together. Select *VesselPart* and press M. This opens a dialogue for merging values. You can use any technique you wish. Here, since we know there are 4 frequent values, we will use the final option, *Group all except 4 most frequent values*. We can also give the name to the new value, but let's leave it a *other* for now.



You can see the less frequent values have been remapped. Do not forget to press *Apply* once you have finished editing. Looking at the Box Plot, the values are now much easier to interpret.



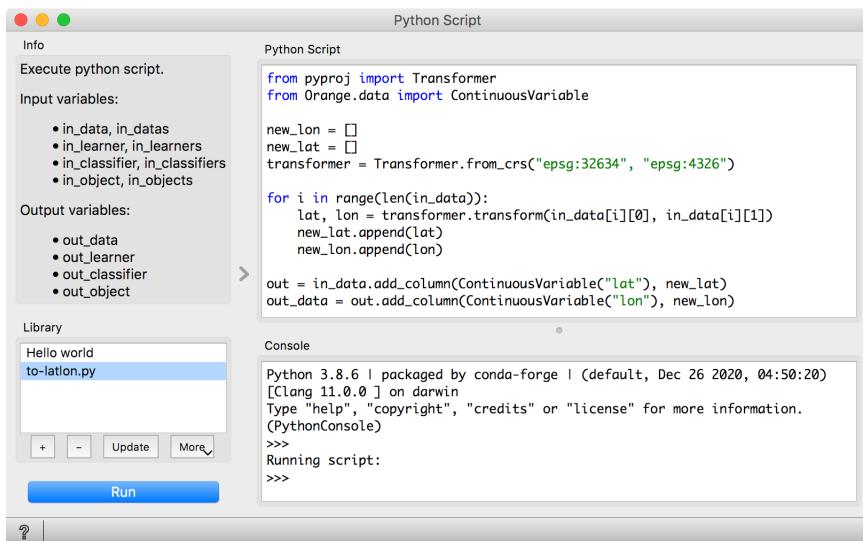
### Assignment

Preprocess the pottery data using the technique from above and answer the following questions:

- Put metadata to meta attributes using Select Columns. Which features did you put there? Why?
- Which features had to be preprocessed by merging infrequent values?
- Is there another way we could merge values?
- Use Box Plot after Edit Domain and

# Assignment: Geolocations

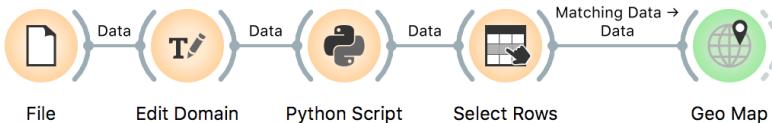
The Antikythera data set contains information on the geolocation of artefacts. The location is encoded as  $X_{\text{sugg}}$ ,  $Y_{\text{sugg}}$  coordinates in *epsg:32634* system. We need to transform the data to the more standard *EPSG: 4326* projection, which enables us to see the data on a map. We will do this with the help of a *Python Script*.



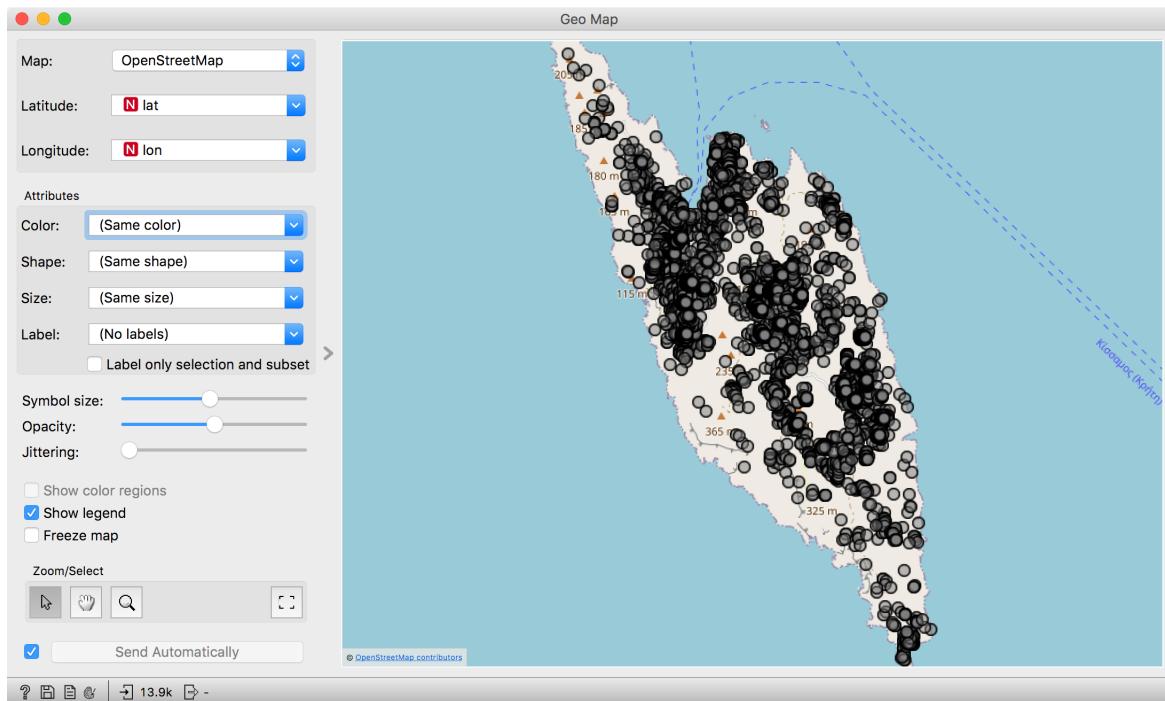
Now, we can plot the data on the map. We will use the Geo add-on for this. If you have not installed the add-on yet, go to Options → Add-ons and install it. Restart Orange for the add-on to appear.

Connect *Geo Map* to *Python Script*. The widget should automatically find latitude and longitude attributes, but if it doesn't, you can set them manually. The plot shows where the pottery shards were collected.

For Python Script to work, one needs to have the 'pyproj' package installed in the Orange environment. To bypass this, [here](#) are the preprocessed data you can load with File and connect directly to Geo Map.



### Assignment



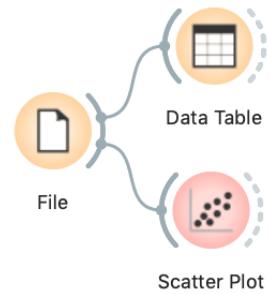
Use Geo Map widget and color the points by an appropriate attribute to answer the following questions:

- Does the location of the shard correspond to the way it was collected?
- Use *Select Rows* to keep only data instances with over 75% certainty that they belong to a certain period. Go through different periods (attributes from MNLN to Other) and observe if items from one period were found in a specific location of the island. Some periods will produce no results (no items with such high certainty), but that's ok. Skip them.
- Use *Select Rows* to explore categorical variables. For example, set the filter to *VesselPart* is *B* and observe where in the island the shards belonging to the body of the vessel were found. Explore variables of your interest and draw conclusions.

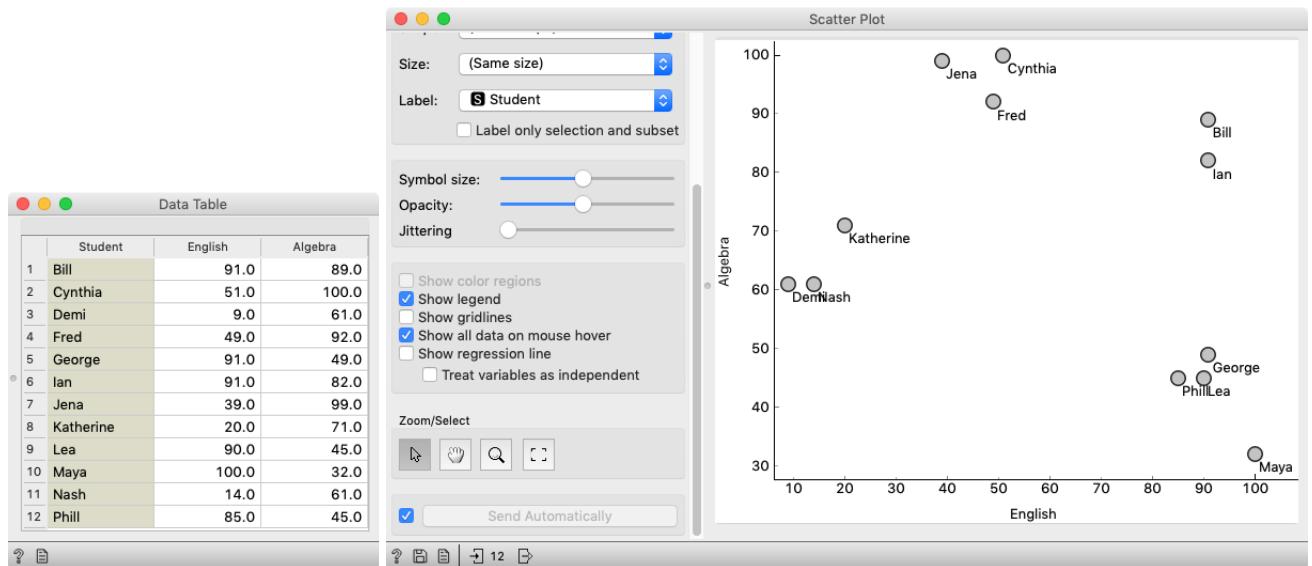
# Hierarchical Clustering

WE ARE INTERESTED IN FINDING CLUSTERS IN OUR DATA. That is, we would like to identify groups of data instances that are close together, similar to each other. Consider a simple, two-featured data set (see the side note) and plot it in the *Scatter Plot*. How many clusters do we have? What defines a cluster? Which data instances should belong to the same cluster? How does the clustering algorithm actually work?

First, we need to define what we mean by "similar". We will assume that all our data instances are described (profiled) with continuous features. One simple measure of similarity is the Euclidean distance. So, we would like to group data instances with small Euclidean distances.

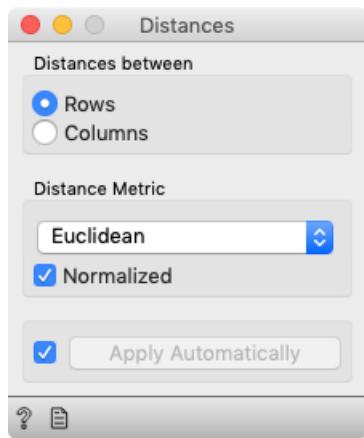


We will introduce clustering with a simple data set on students and their grades in English and Algebra. Load the data set from <http://file.biolab.si/text/grades.tab>.

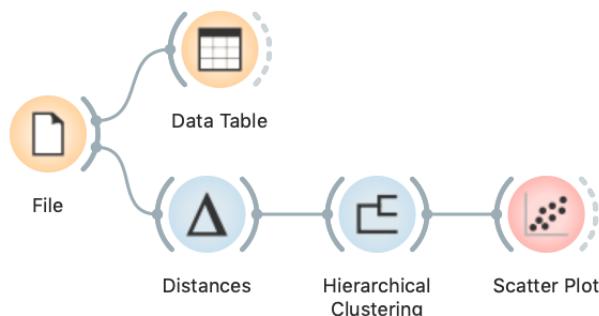


Next, we need to define a clustering algorithm. Say that we start with each data instance being its own cluster, and then, at each step, we join the clusters that are closest together. We estimate the distance between the clusters with, say, the average distance between all their pairs of data points. This algorithm is called hierarchical clustering.

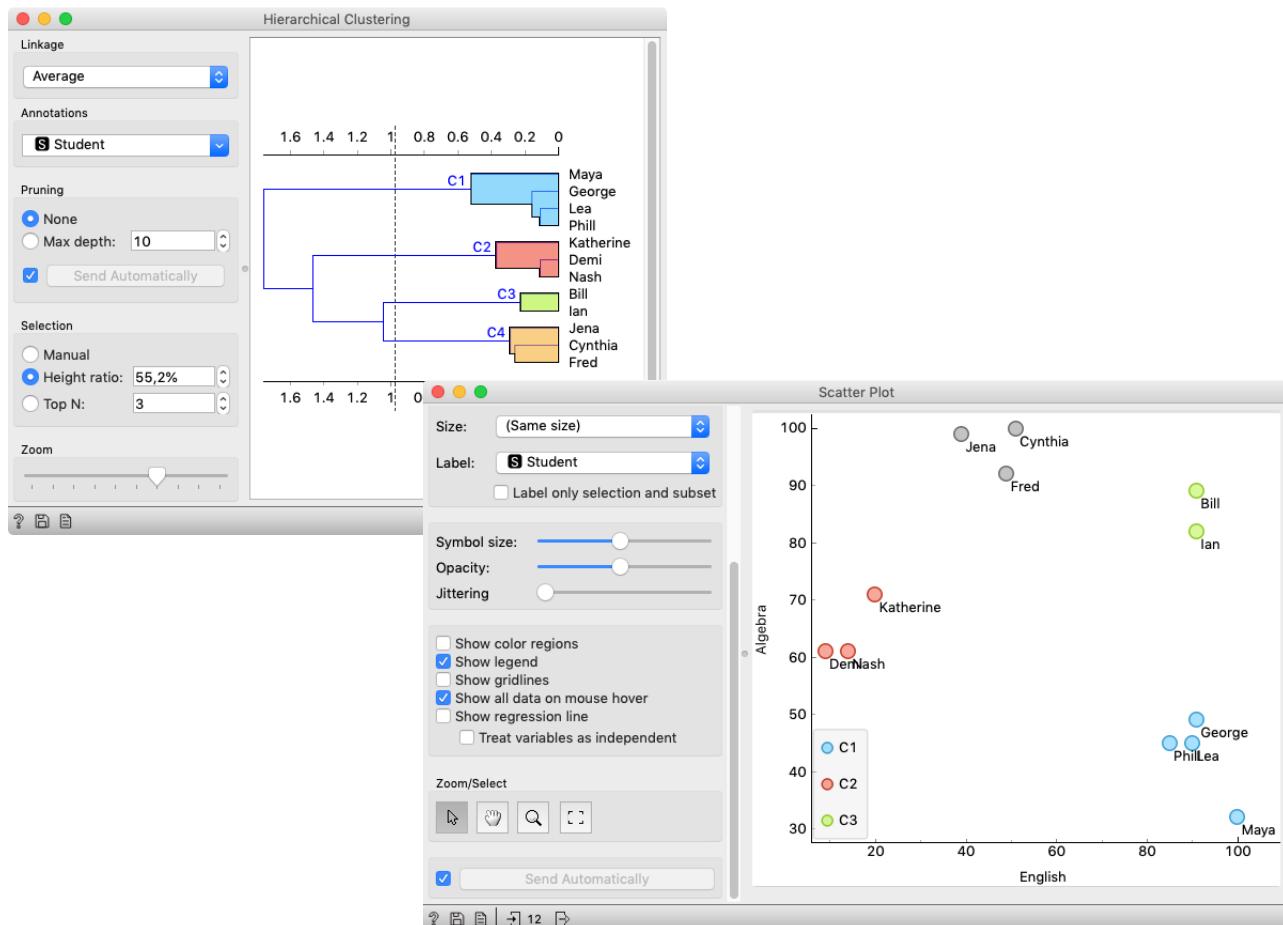
There are different ways to measure the similarity between clusters. The estimate we have described is called average linkage. We could also estimate the distance through the two closest points in each clusters (single linkage), or through the two points that are furthest away (complete linkage).



One possible way to observe the results of clustering on our small data set with grades is with the following workflow:



Couldn't be simpler. Load the data, measure the distances, use them in hierarchical clustering, and visualize the results in a scatter plot. The *Hierarchical Clustering* widget allows us to cut the hierarchy at a certain distance score and output the corresponding clusters:



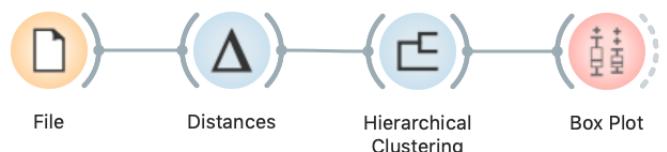
# Animal Kingdom

Your lecturers spent a substantial part of their youth admiring a particular Croatian chocolate called Animal Kingdom. Each chocolate bar came with a card—a drawing of some (random) animal, and the associated album made us eat a lot of chocolate.

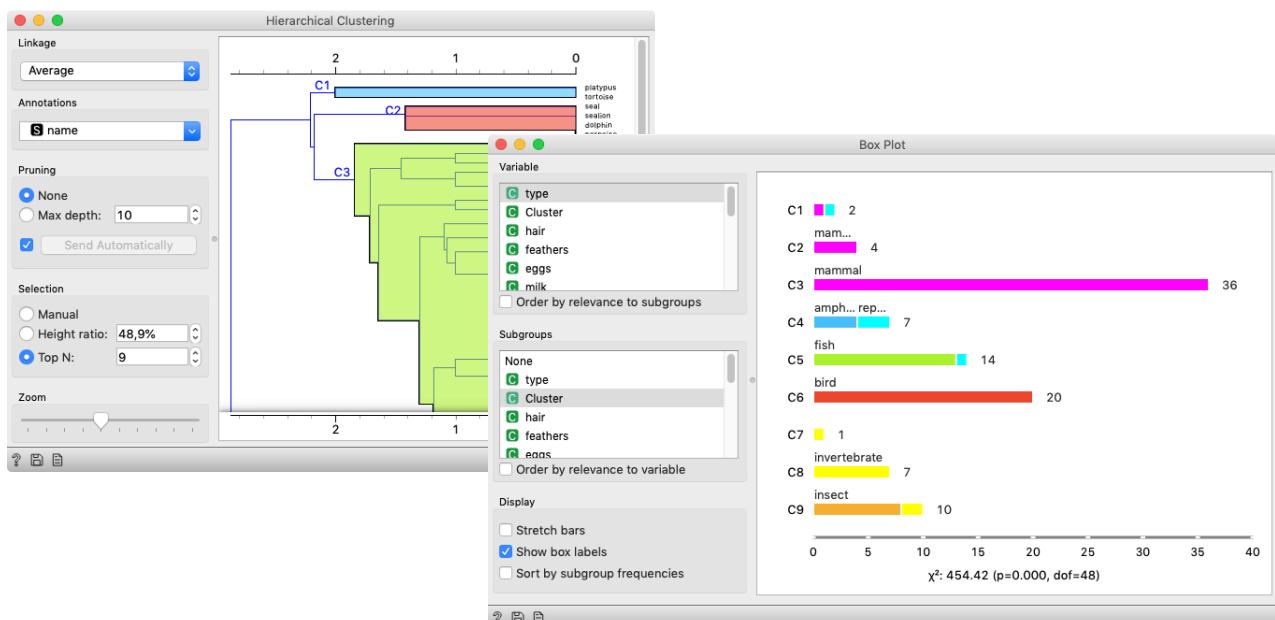
Funny stuff was we never understood the order in which the cards were laid out in the album. We later learned about taxonomy, but being more inclined to engineering we never mastered learning it in our biology classes. Luckily, there's data mining and the idea that taxonomy simply stems from measuring the distance between species.

Here we use the `zoo` data (from the documentation data sets) with attributes that report on various features of animals (has hair, has feathers, lays eggs). We measure the distance and compute the clustering. Animals in this data set are annotated with type (mammal, insect, bird, and so on). It would be cool to know if the clustering re-discovered these groups of animals.

To split the data into clusters, let us manually set a threshold by dragging the vertical line left or right in the visualization. Can you say what is the appropriate number of groups?



Hierarchical clustering works fast for smaller data sets. But for bigger ones it fails. Simply, it cannot be used. Why?



What is wrong with those mammals? Why can't they be in one single cluster? Two reasons. First, they represent 40% of the data instances. Second, they include some weirdos. Who are they?

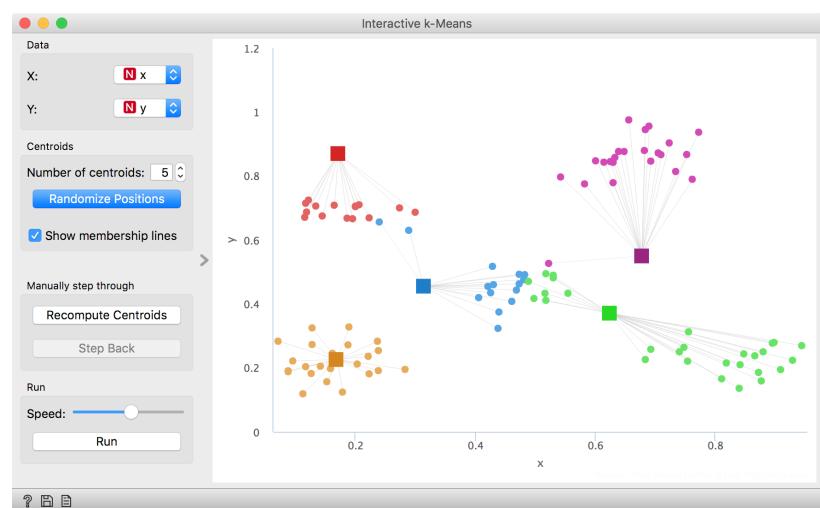
# *k*-Means Clustering

HIERARCHICAL CLUSTERING IS NOT SUITABLE FOR LARGER DATA SETS due to the prohibitive size of the distance matrix: with 30 thousand objects, the distance matrix already has almost one billion elements. An alternative approach that avoids using the distance matrix is k-means clustering.

K-means clustering randomly selects  $k$  centers (with  $k$  specified in advance). Then it alternates between two steps. In one step, it assigns each point to its closest center, thus forming  $k$  clusters. In the other, it recomputes the centers of the clusters. Repeating these two steps typically converges quite fast; even for big data sets with millions of data points it usually takes just a couple of ten or hundred iterations.

Orange's Educational add-on provides a widget *Interactive k-Means*, which illustrates the algorithm.

Use the *Paint Data* widget to paint some data - maybe five groups of points. Feed it to Interactive k-means and set the number of centroids to 5. You may get something like this.



Keep pressing Recompute Centroids and Reassign Membership until the plot stops changing. With this simple, two-dimensional data it will take just a few iterations; with more points and features, it can take longer, but the principle is the same.

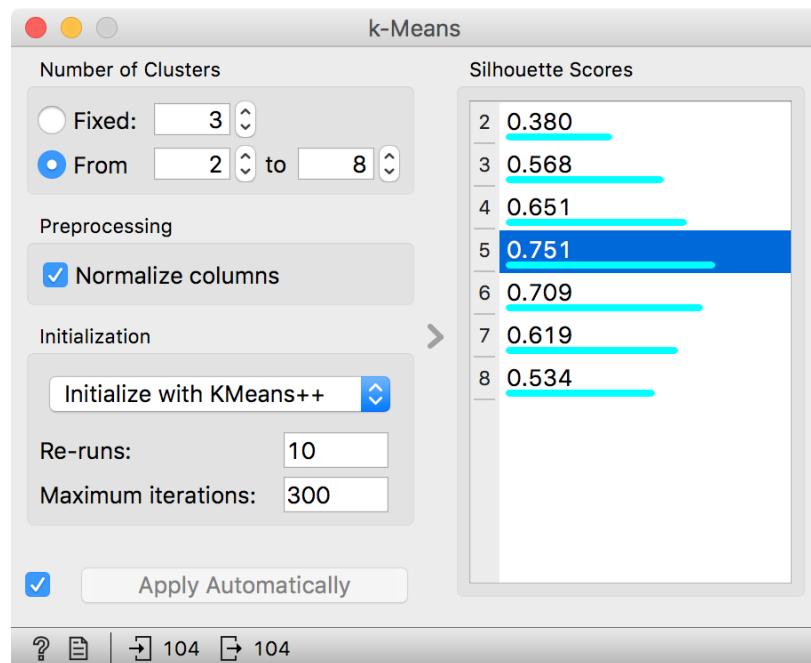
How do we set the initial number of clusters? That's simple: we choose the number that gives the optimal clustering.

Well then, how do we define the optimal clustering? This one is a bit harder. We want small distances between points in the same cluster and large distances between points from different clusters.

Pick one point, and let A be its average distance to the data points in the same cluster and let B represent the average distance to the points from the closest other cluster. (The closest cluster? Just compute B for all other clusters and take the lowest value.) The value  $(B - A) / \max(A, B)$  is called silhouette; the higher the silhouette, the better the point fits into its cluster. The average silhouette across all points is the silhouette of the clustering. The higher the silhouette, the better the clustering.

Now that we can assess the quality of clustering, we can run k-means with different values of parameter k (number of clusters) and select k which gives the largest silhouette.

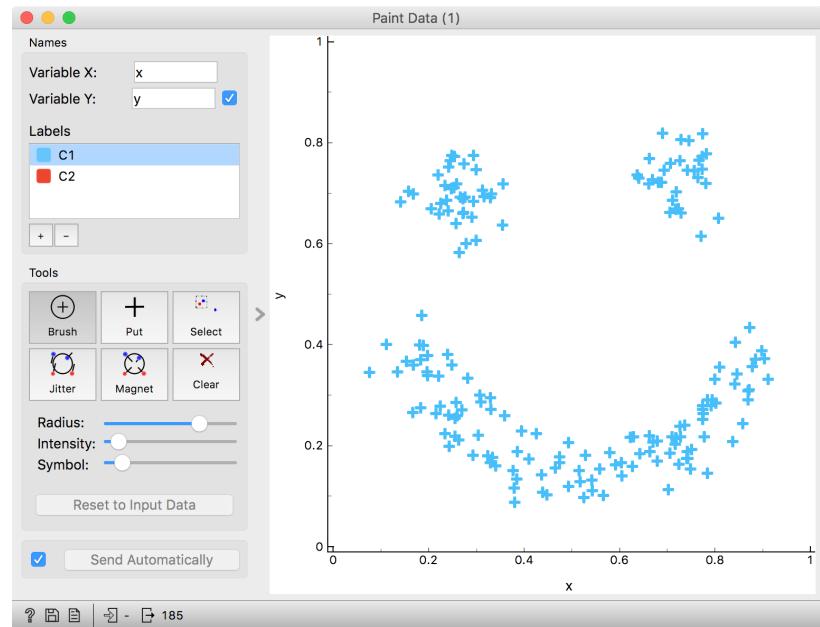
For this, we abandon our educational toy and connect Paint Data to the widget k-Means. We tell it to find the optimal number of clusters between 2 and 8, as scored by the Silhouette.



Works like a charm.

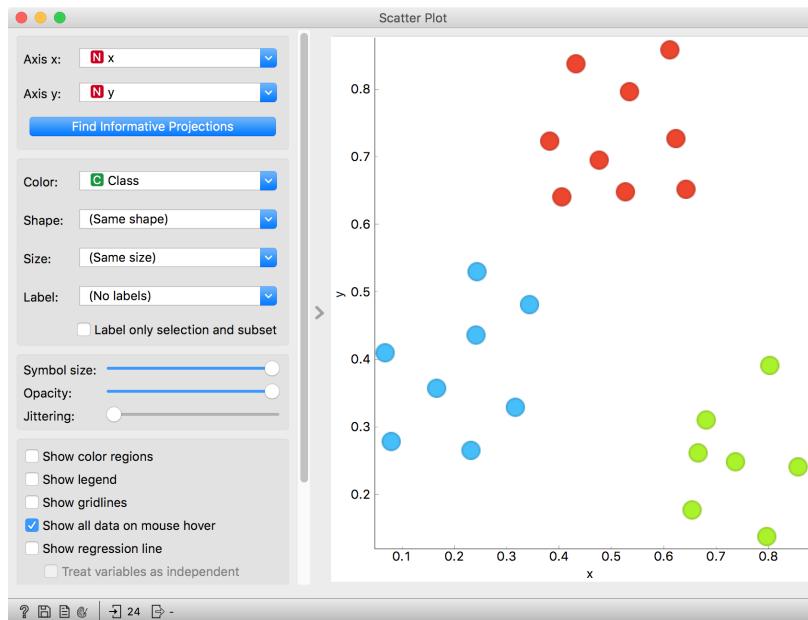
Except that it often doesn't. First, the result of k-means clustering depends on the initial selection of centers. With unfortunate selection, it may get stuck in a local optimum. We solve this by re-running the clustering multiple times from random positions and using the best result. Second, the silhouette sometimes fails to correctly evaluate the clustering. Nobody's perfect.

Time to experiment. Connect the Scatter Plot to k-Means. Change the number of clusters. See if the clusters make sense. Could you paint the data where k-Means fails? Or where it works really well?

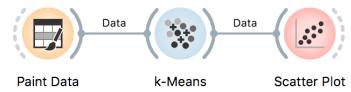


# Silhouettes

CONSIDER A TWO-FEATURE DATA SET which we have painted in the *Paint Data* widget. We send it to the k-means clustering, tell it to find three clusters, and display the clustering in the scatter plot.



Don't get confused: we paint data and/or visualize it with Scatter plots, which show only two features. This is just for an illustration! Most data sets contain many features and methods like k-Means clustering take into account all features, not just two.

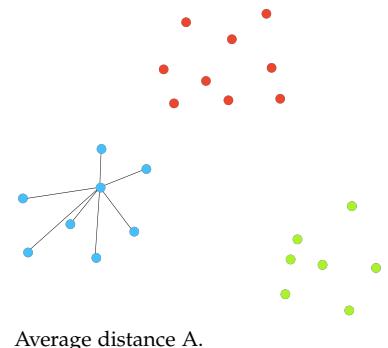


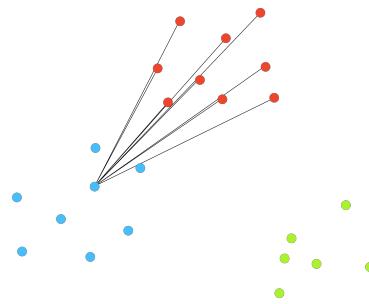
The data points in the green cluster are well separated from those in the other two. Not so for the blue and red points, where several points are on the border between the clusters. We would like to quantify the degree of how well a data point belongs to the cluster to which it is assigned.

We will invent a scoring measure for this and we will call it a silhouette (because this is how it's called). Our goal: a silhouette of 1 (one) will mean that the data instance is well rooted in the cluster, while the score of 0 (zero) will be assigned to data instances on the border between two clusters.

For a given data point (say the blue point in the image on the left), we can measure the distance to all the other points in its cluster and compute the average. Let us denote this average distance with  $A$ . The smaller the  $A$ , the better.

On the other hand, we would like a data point to be far away from the points in the closest neighboring cluster. The closest cluster to our blue data point is the red cluster. We can measure the distances between the blue data point and all the points in the red cluster, and again compute the average. Let us denote this average distance as  $B$ .





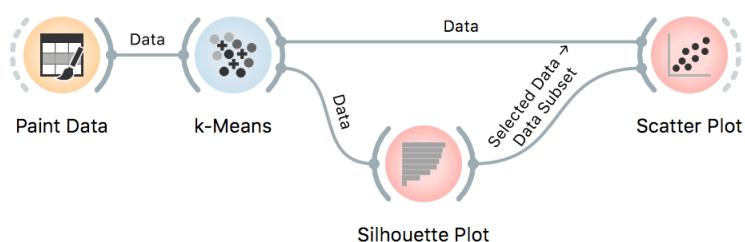
Average distance B.

$C_3$  is the green cluster, and all its points have large silhouettes. Not so for the other two.

The larger the B, the better.

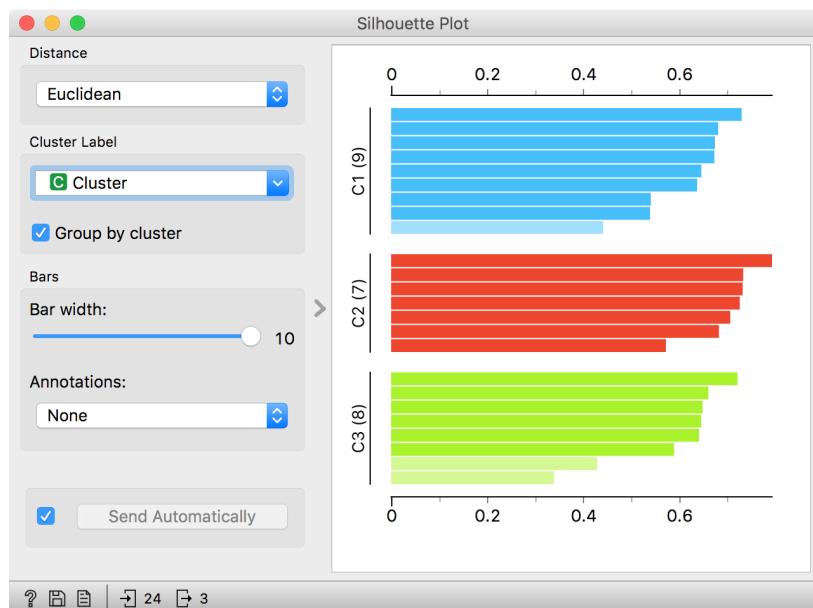
The point is well rooted within its own cluster if the distance to the points from the neighboring cluster (B) is much larger than the distance to the points from its own cluster (A), hence we compute  $B-A$ . We normalize it by dividing it with the larger of these two numbers,  $S = (B - A) / \max A, B$ . Voilà,  $S$  is our silhouette score.

Orange has a *Silhouette Plot* widget that displays the values of the silhouette score for each data instance. We can also choose a particular data instance in the silhouette plot and check out its position in the scatter plot.



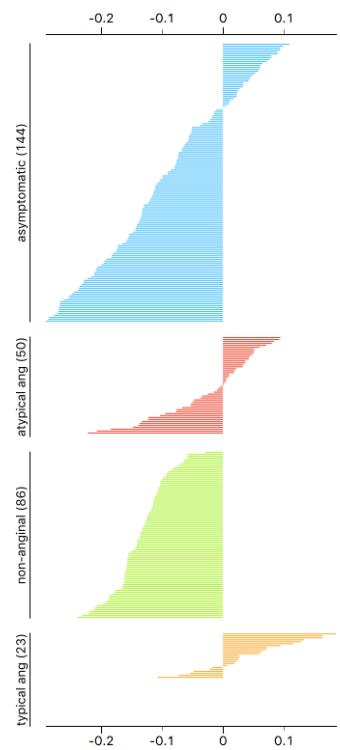
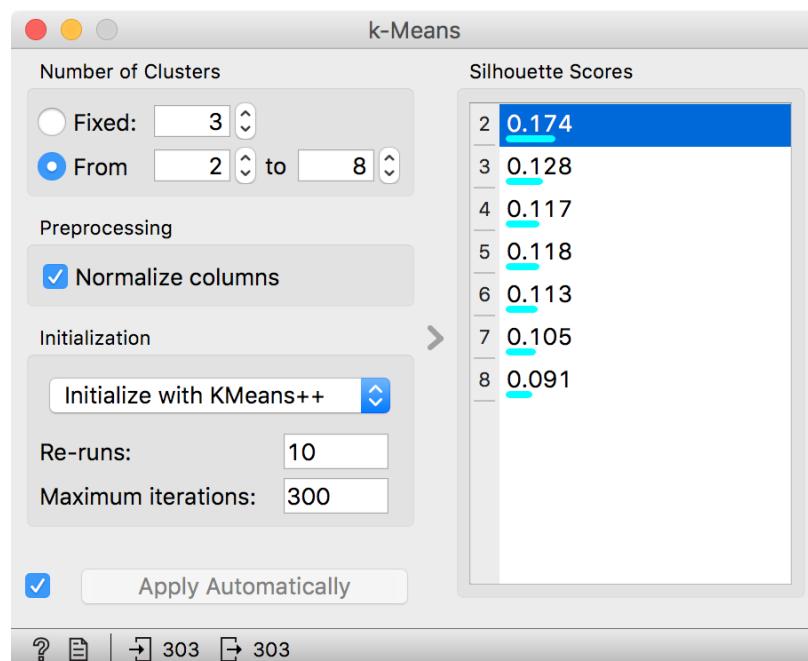
We selected three data instances with the worst silhouette scores. Can you guess where they lie in the scatter plot?

This of course looks great for data sets with two features, where the scatter plot reveals all the information. In higher-dimensional data, the scatter plot shows just two features at a time, so two points that seem close in the scatter plot may be actually far apart when all features - perhaps thousands of gene expressions - are taken into account.



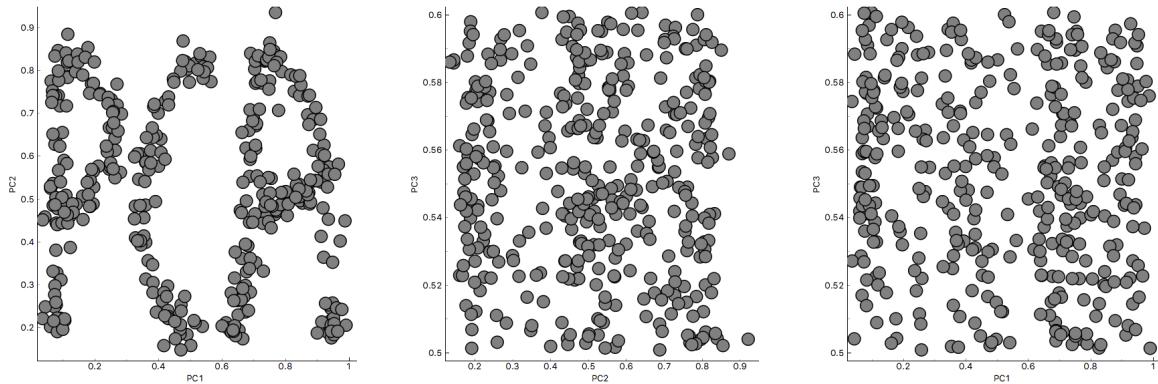
The total quality of clustering - the silhouette of the clustering - is the average silhouette across all points. When the *k-Means* widget searches for the optimal number of clusters, it tries a different number of clusters and displays the corresponding silhouette scores. Ah, one more thing: Silhouette Plot can be used on any data, not just on data sets that are the output of clustering. We could use it with the iris data set and figure out which class is well separated from the other two and, conversely, which data instances from one class are similar to those from another.

We don't have to group the instances by the class. For instance, the silhouette on the left would suggest that the patients from the heart disease data with typical anginal pain are similar to each other (with respect to the distance/similarity computed from all features), while those with other types of pain, especially non-anginal pain are not clustered together at all.



# Principal Component Analysis

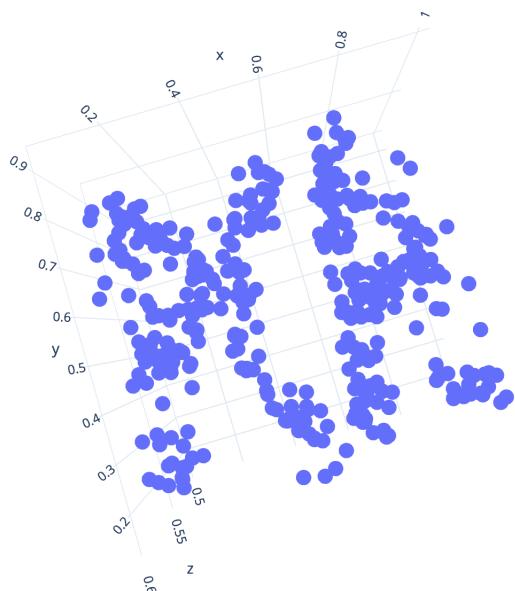
Which of the following three scatter plots (showing x vs. y, x vs. z and y vs. z) for the same three-dimensional data gives us the best picture about the actual layout of the data in space?



Yes, the first scatter plot looks very useful: it tells us that x and y are highly correlated and that we have three clusters of somewhat irregular shape. But remember: this data is three dimensional. What is we saw it from another, perhaps better perspective?

Let's make another experiment. Go to <https://in-the-sky.org/ngc3d.php>, disable Auto-rotate and Show labels and select Zoom to show Local Milky Way. Now let's rotate the picture of the galaxy to find the layout of the stars.

Think about what we've done. What are the properties of the best projection?



We want the data to be as spread out as possible. If we look from the direction parallel to the galactic plane, we see just a line. We lose one dimension, essentially keeping just a single coordinate for each star. (This is unfortunately exactly the perspective we see on the night sky: most stars are in the bright band we call the milky way, and we only see the outliers.) Among all possible projections, we attempt to find the one with the highest spread across the scatter plot. This projection may not be (and usually isn't) orthogonal to any axis; it may be a projection to an arbitrary plane.

We again talk about two dimensional projection only for the sake of illustration. Imagine that we have ten thousand dimensional data and we would like, for some reason, keep just ten features. Yes, we can rank the features and keep the most informative, but what if these are correlated and tell us the same thing? Or what if our data does not have any target variable: with what should the "good features" be correlated? And what if the optimal projection is not aligned with the axes at all, so "good" features are combinations of the original ones?

We can do the same reasoning as above: we want to find a 10-dimensional (for the sake of examples) projection in which the data points are as spread as possible.

How do we do this? Let's go back to our everyday's three dimensional world and think about how to find a two-dimensional projection.

Imagine you are observing a swarm of flies; your data are their exact coordinates in the room, so the position of each fly is described by three numbers. Then you discover that your flies actually fly in a formation: they are (almost) on the same line. You could then describe the position of each fly with a single number that represents the fly's position along the line. Plus, you need to know where in the space the line lies. We call this line the first principal component. By using it, we reduce the three-dimensional space into a single dimension.

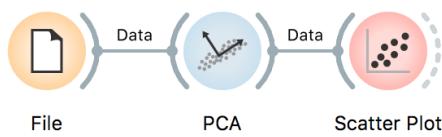
After some careful observation, you notice the flies are a bit spread in one other direction, so they do not fly along a line but along a band. Therefore, we need two numbers, one along the first and one along the — you guessed it — second principal component.

It turns out the flies are actually also spread in the third direction. Thus you need three numbers after all.

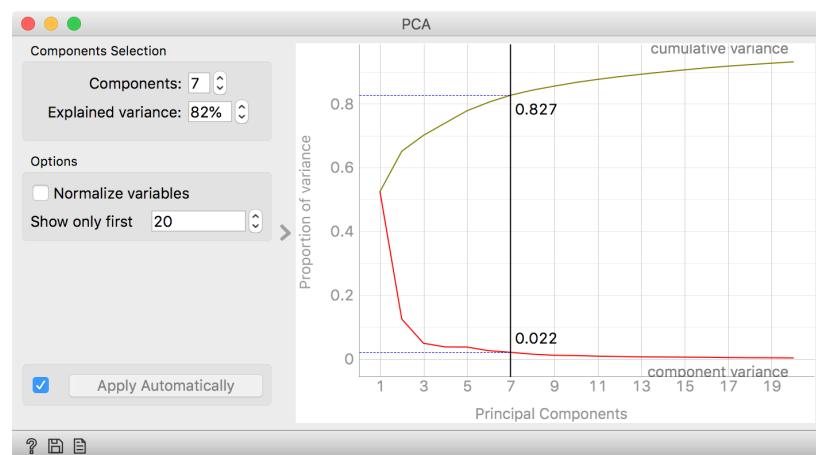
Or do you? It all depends on how spread they are in the second and in the third direction. If the spread along the second is relatively small in comparison with the first, you are fine with a single dimension. If not, you need two, but perhaps still not three.

Let's step back a bit: why would one who carefully measured expressions of ten thousand genes want to throw most data away and reduce it to a dozen dimensions? The data, in general, may not and does not have as many dimensions as there are features. Say you have an experiment in which you spill different amounts of two chemicals over colonies of amoebas and then measure the expressions of 10,000 genes. Instead of flies in a three-dimensional space, you now profile colonies in a 10,000-dimensional space, the coordinates corresponding to gene expressions. Yet if expressions of genes depend only on the concentrations of these two chemicals, you can compute all 10,000 numbers from just two. Your data is then just two-dimensional.

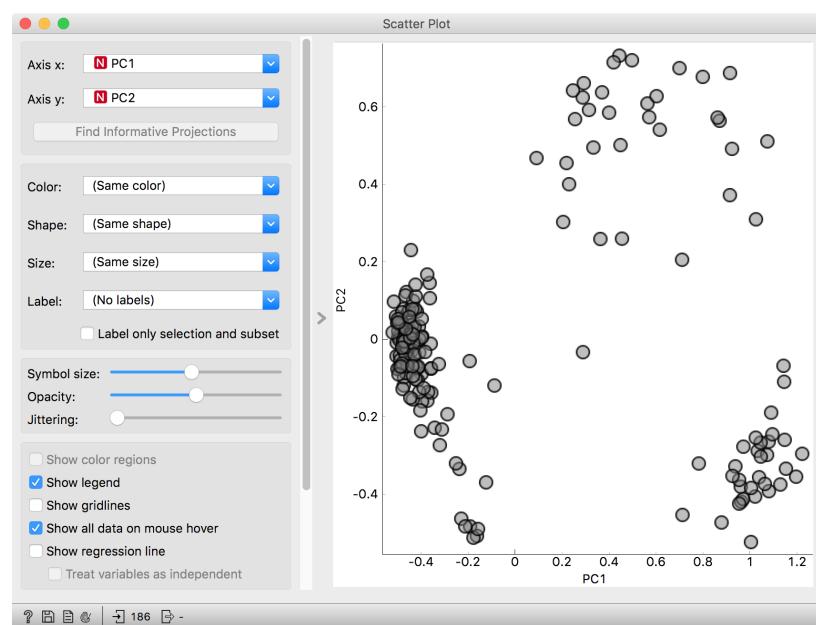
A technique that does this is called Principle Components Analysis, or *PCA*. The corresponding widget is simple: it receives the data and outputs the transformed data.



The widget allows you to select the number of components and helps you by showing how much information (technically: explained variance) you retain with respect to the number of components (brownish line) and the amount of information (explained variance) in each component.

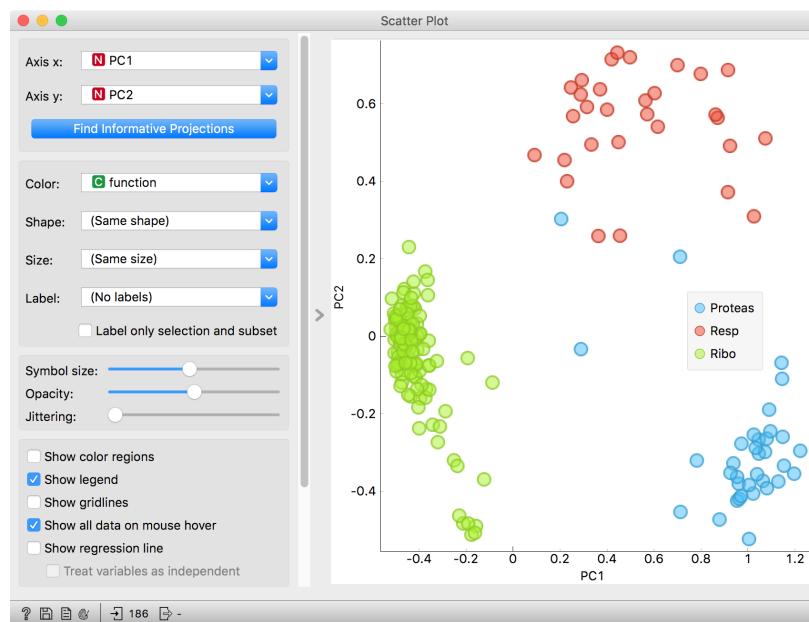


The PCA on the left shows the scree diagram for brown-selected data. Set like this, the widget replaces the 80 features with just seven - and still keeping 82.7% of information. (Note: disable "Normalize data" checkbox to get the same picture.) Let us see a scatter plot for the first two components.

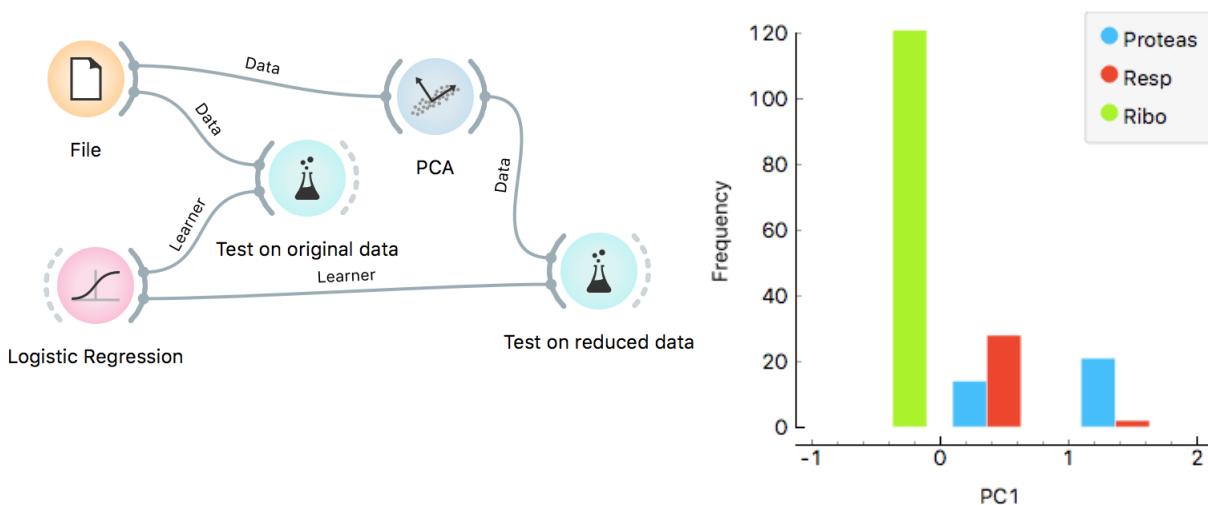


The axes, PC1 and PC2, do not correspond to particular features in the original data, but to their linear combination. What we are looking at is a projection onto the plane, defined by the first two components. When you consider only two components, you can imagine that PCA puts a hyperplane into multidimensional space and projects all data into it.

Note that this is an unsupervised method: it does not care about the class. The classes in the projection may be well separated or not. Let's add some colors to the points and see how lucky we are this time.



The data separated so well that these two dimensions alone may suffice for building a good classifier. No, wait, it gets even better. The data classes are separated well even along the first component. So we should be able to build a classifier from a single feature!



In the above schema we uses the ordinary Test and Score widget, but renamed it to "Test on original data" for better understanding of the workflow.

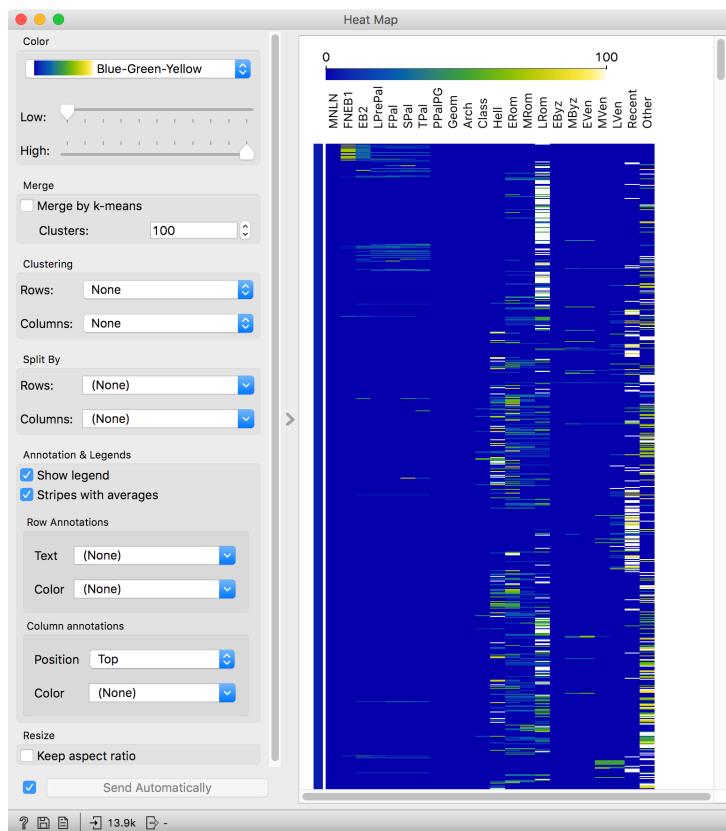
On the original data, logistic regression gets 98% AUC and classification accuracy. If we select just a single component in PCA, we already get a 93%, and if we take two, we get the same result as on the original data.

PCA is thus useful for multiple purposes. It can simplify our data by combining the existing features to a much smaller number of features without losing much information. The directions of these features may tell us something about the data. Finally, it can find us good two-dimensional projections that we can observe in scatter plots.

# Assignment:

## Hierarchical Clustering

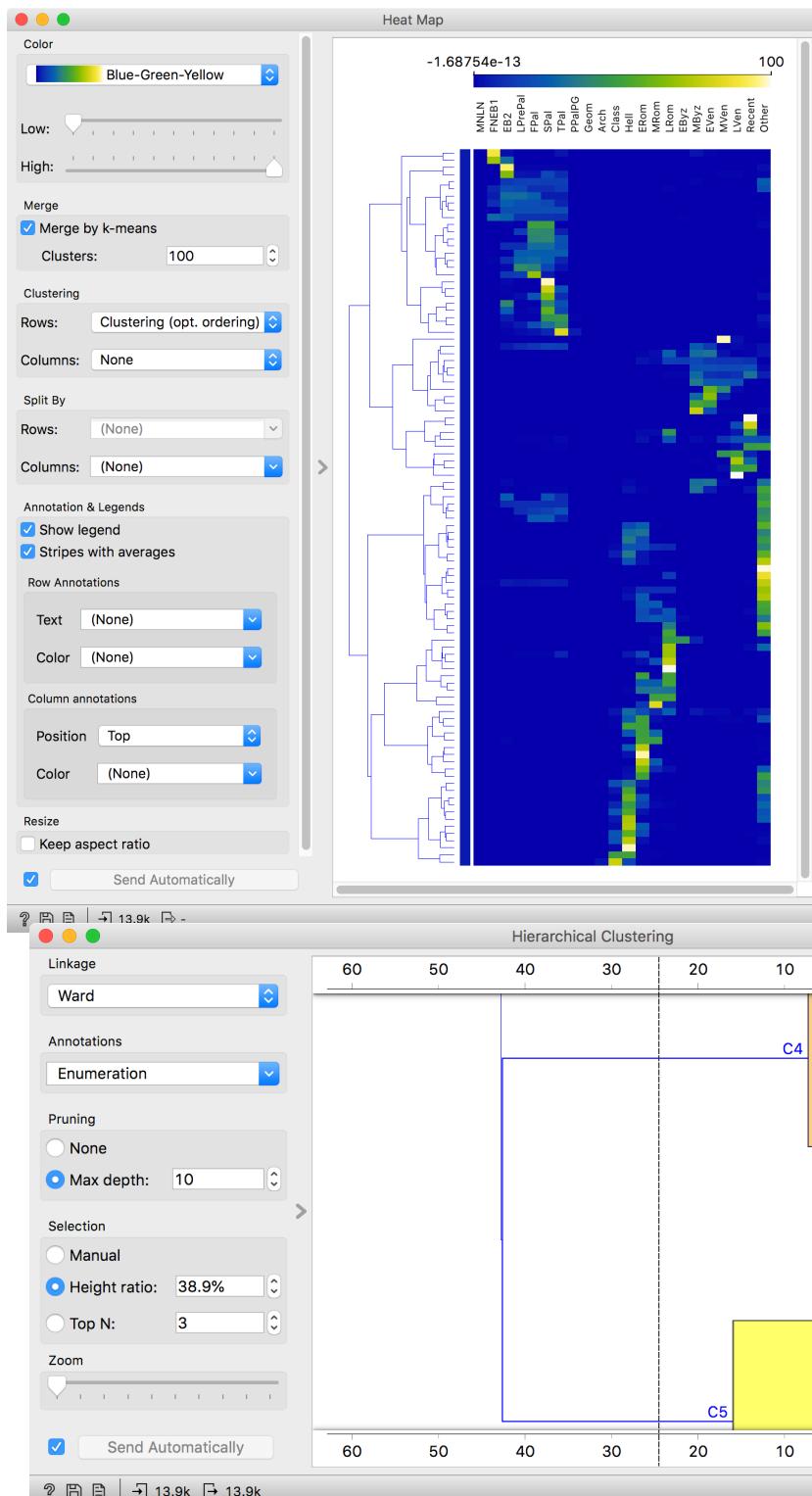
SHARDS BELONG TO DIFFERENT TIME PERIODS. Periods are described with 22 numeric variables, each defining the likelihood of a shard belonging to the said period. First, let us visualize the data in a visualization that shows the value of numeric variables with color - the higher the value, the brighter the number. Such a visualization is called a *Heat Map*.



By default, Heat Map shows each data instance in a row and maps the values to the color scale. But our data is so unorganized, there is no way we can make sense of it! First, since we have many rows in our data set, let us join together rows that are the most similar. We will do this with *Merge by k-means* option and set the number of clusters to 100. This is keep only 100 most different rows.

Visualization got simpler, but it is still unorganized. Would it be nice to have similar rows closer together? Of course it would! We can use hierarchical clustering to re-order rows, so that similar rows will be put next to each other. Use *Clustering (opt. ordering)* - the plot

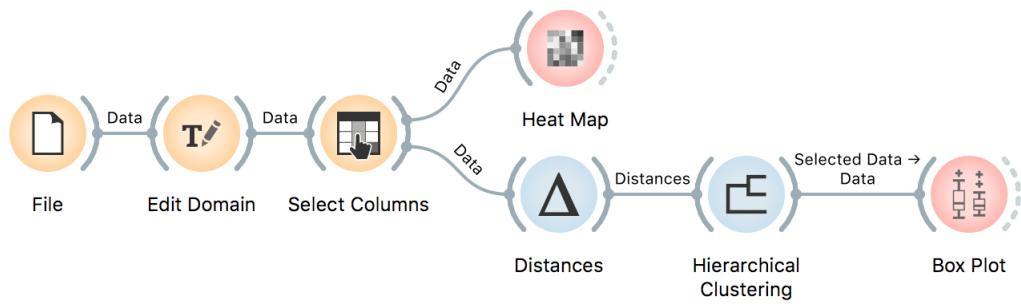
now makes much more sense!



In the top left corner, we have shards from the early periods. In the bottom central part shards from Roman and Hellenic periods. And in the middle right part recent or unidentified shards.

We can repeat the same, but a more exact procedure, with *Hierarchical Clustering*. First, let us pass the data to *Distances* to compute the distance matrix. As our data has a fair number of dimensions (22 to be precise), we will use *cosine distance* and *Ward linkage* for the measure of cluster similarity.

In *Hierachial Clustering*, we see a large dendrogram. To make it easier to read, set *Max depth* to 10 (this will cut the tree at the tenth split). Use *Zoom* to zoom out as much as possible.



### Assignment

Use Box Plot to explore the clusters:

- What is a good number of clusters? Why?
- Cut the dendrogram at the desired level (up to you). Which period(s) does each of the clusters represent? Could you name them?
- What defines each period? Use Box Plot with *Data* output to find out.

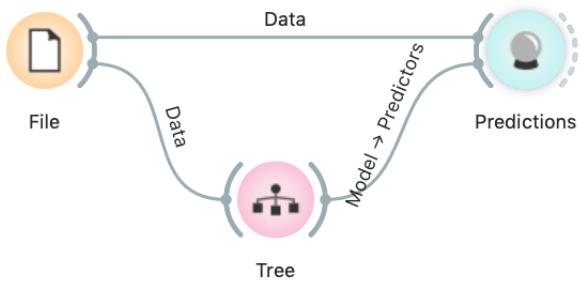
# Classification

We call the variable we wish to predict a target variable, or an outcome or, in traditional machine learning terminology, a class. Hence we talk about classification, classifiers, classification trees...

We have seen the iris data before. We wanted to predict varieties based on measurements—but we actually did not make any predictions. We observed some potentially interesting relations between the features and the varieties, but have never constructed an actual model.

Let us create one now.

Something in this workflow is conceptually wrong. Can you guess what?



Screenshot of the Orange data mining software interface showing the 'Predictions' widget. The left sidebar lists classes: Iris-setosa, Iris-versicolor, Iris-virginica. The main area shows a table of predictions for the Iris dataset. The table has columns: Tree, iris, sepal length, sepal width. Rows 47-55 show the following data:

Tree	iris	sepal length	sepal width
47 1.00 : 0.00 : 0.00 → Iris-setosa	Iris-setosa	5.1	3.8
48 1.00 : 0.00 : 0.00 → Iris-setosa	Iris-setosa	4.6	3.2
49 1.00 : 0.00 : 0.00 → Iris-setosa	Iris-setosa	5.3	3.7
50 1.00 : 0.00 : 0.00 → Iris-setosa	Iris-setosa	5.0	3.3
51 0.00 : 0.98 : 0.02 → Iris-versi...	Iris-versicolor	7.0	3.2
52 0.00 : 0.98 : 0.02 → Iris-versi...	Iris-versicolor	6.4	3.2
53 0.00 : 0.98 : 0.02 → Iris-versi...	Iris-versicolor	6.9	3.1
54 0.00 : 0.98 : 0.02 → Iris-versi...	Iris-versicolor	5.5	2.3
55 0.00 : 0.98 : 0.02 → Iris-versi...	Iris-versicolor	6.5	2.8

Below the table, a summary row shows: Model: Tree, AUC: 0.993, CA: 0.980, F1: 0.980, Precision: 0.980, Recall: 0.980. A blue button at the bottom left says 'Restore Original Order'.

The data is fed into the *Tree* widget, which infers a classification model and gives it to the *Predictions* widget. Note that unlike in our past workflows, in which the communication between widgets included only the data, we here have a channel that carries a predictive model.

The *Predictions* widget also receives the data from the *File* widget. The widget uses the model

to make predictions about the data and shows them in the table.

How correct are these predictions? Do we have a good model?  
How can we tell?

But (and even before answering these very important questions), what is a classification tree? And how does Orange create one? Is this algorithm something we should really use?

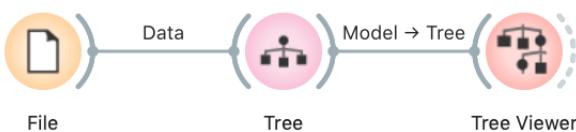
So many questions to answer!

# Classification Trees

In the previous lesson, we used a classification tree, one of the oldest, but still popular, machine learning methods. We like it since the method is easy to explain and gives rise to random forests, one of the most accurate machine learning techniques (more on this later). So, what kind of model is a classification tree?

Let us load *iris* data set, build a tree (widget *Tree*) and visualize it in a *Tree Viewer*.

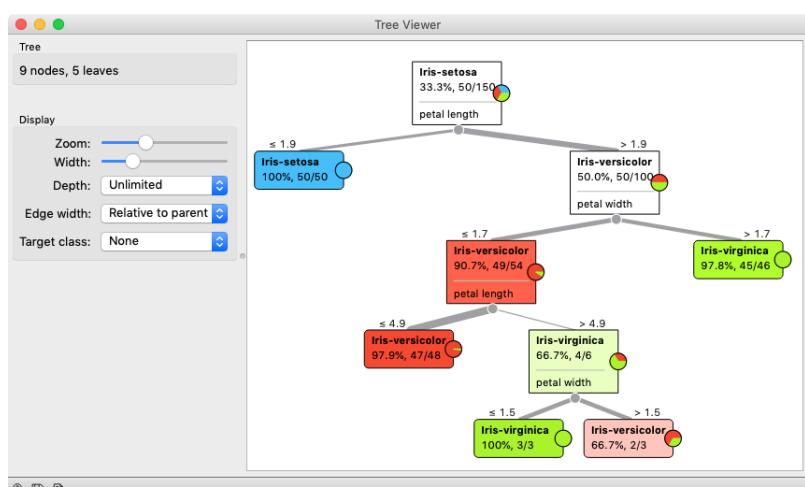
Classification trees were hugely popular in the early years of machine learning, when they were first independently proposed by the engineer Ross Quinlan (C4.5) and a group of statisticians (CART), including the father of random forests Leo Brieman.



	iris	sepal length	sepal width	petal length	petal width
1	Iris-setosa	5.1	3.5	1.4	0.2
2	Iris-setosa	4.9	3.0	1.4	0.2
3	Iris-setosa	4.7	3.2	1.3	0.2
4	Iris-setosa	4.6	3.1	1.5	0.2
5	Iris-setosa	5.0	3.6	1.4	0.2
6	Iris-setosa	5.4	3.9	1.7	0.4
7	Iris-setosa	4.6	3.4	1.4	0.3
8	Iris-setosa	5.0	3.4	1.5	0.2
9	Iris-setosa	4.4	2.9	1.4	0.2
10	Iris-setosa	4.9	3.1	1.5	0.1
11	Iris-setosa	5.4	3.7	1.5	0.2
12	Iris-setosa	4.8	3.4	1.6	0.2
13	Iris-setosa	4.8	3.0	1.4	0.1
14	Iris-setosa	4.3	3.0	1.1	0.1
15	Iris-setosa	5.8	4.0	1.2	0.2
16	Iris-setosa	5.7	4.4	1.5	0.4
17	Iris-setosa	5.4	3.9	1.3	0.4

We read the tree from top to bottom. Looks like the column *petal length* best separates the iris variety *setosa* from the others, and in the next step, *petal width* then almost perfectly separates the remaining two varieties.

Trees place the most useful feature at the root. What would be the most useful feature? The feature that splits the data into two purest possible subsets. It then splits both subsets further, again by their most useful features, and keeps doing so until it reaches subsets in which all data belongs to the same class (leaf nodes in strong blue or red) or until it runs out of data instances to

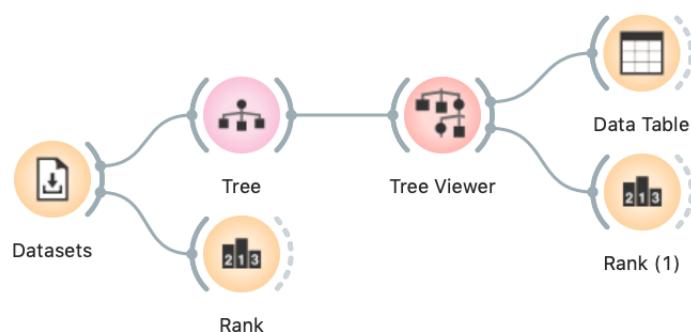


The *Rank* widget can be used on its own to show the best predicting features. Say, to figure out which genes are best predictors of the phenotype in some gene expression data set.

The *Datasets* widget is set to load the *Sailing* data set. To use the second *Rank*, select a node in the *Tree Viewer*.

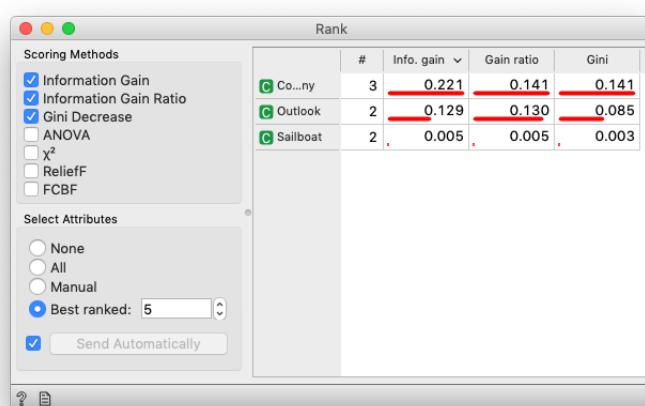
split or out of useful features (the two leaf nodes in white).

We still have not been very explicit about what we mean by "the most useful" feature. There are many ways to measure the quality of features, based on how well they distinguish between classes. We will illustrate the general idea with information gain. We can compute this measure in Orange using the *Rank* widget, which estimates the quality of data features and ranks them according to how informative they are about the class. We can either estimate the information gain from the whole data set, or compute it on data corresponding to an internal node of the classification tree in the *Tree Viewer*. In the following example we use the *Sailing* data set.

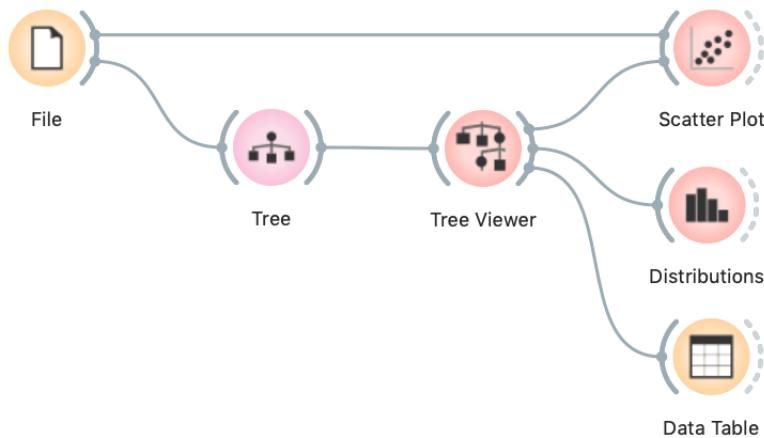


Besides the information gain, *Rank* displays several other measures (including Gain Ratio and Gini), which are often quite in agreement and were invented to better handle discrete features with many different values.

For the whole *Sailing* data set, *Company* is the most class-informative feature according to all measures shown.

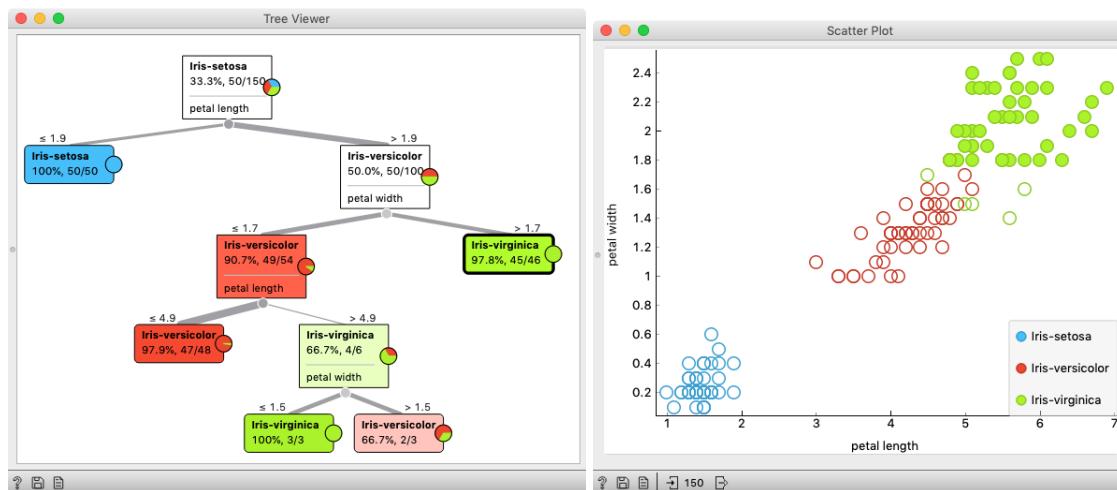


Here is an interesting combination of a *Tree Viewer* and a *Scatter Plot*. This time, use the *Iris* data set. In the *Scatter Plot*, we first find the best visualization of this data set, that is, the one that best separates the instances from different classes. Then we connect the *Tree Viewer* to the *Scatter Plot*. Data instances (particular irises) from the selected node in the *Tree Viewer* are shown in the *Scatter Plot*.



Careful, the *Data* widget needs to be connected to the *Scatter Plot*'s *Data* input, and *Tree Viewer* to the *Scatter Plot*'s *Data Subset* input.

Just for fun, we have included a few other widgets in this workflow. In a way, a *Tree Viewer* behaves like *Select Rows*, except that the rules used to filter the data are inferred from the data itself and optimized to obtain purer data subsets.



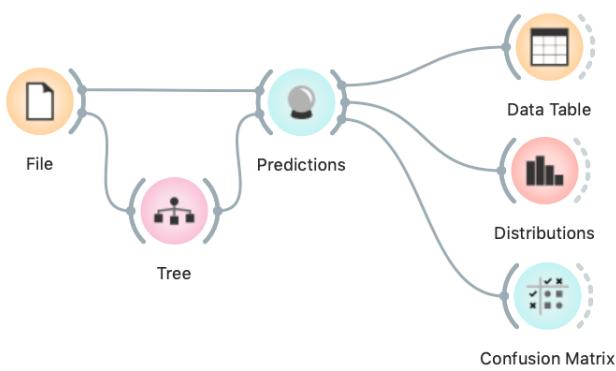
Wherever possible, visualizations in Orange are designed to support selection and passing of the data that applies to it. Finding interesting data subsets and analyzing their commonalities is a central part of explorative data analysis, a data analysis approach favored by the data visualization guru Edward Tufte.

In the *Tree Viewer* we selected the right-most node. All data instances coming to the selected node are highlighted in *Scatter Plot*.

# Classification Accuracy

$$\text{accuracy} = \frac{\#\{\text{correct}\}}{\#\{\text{all}\}}$$

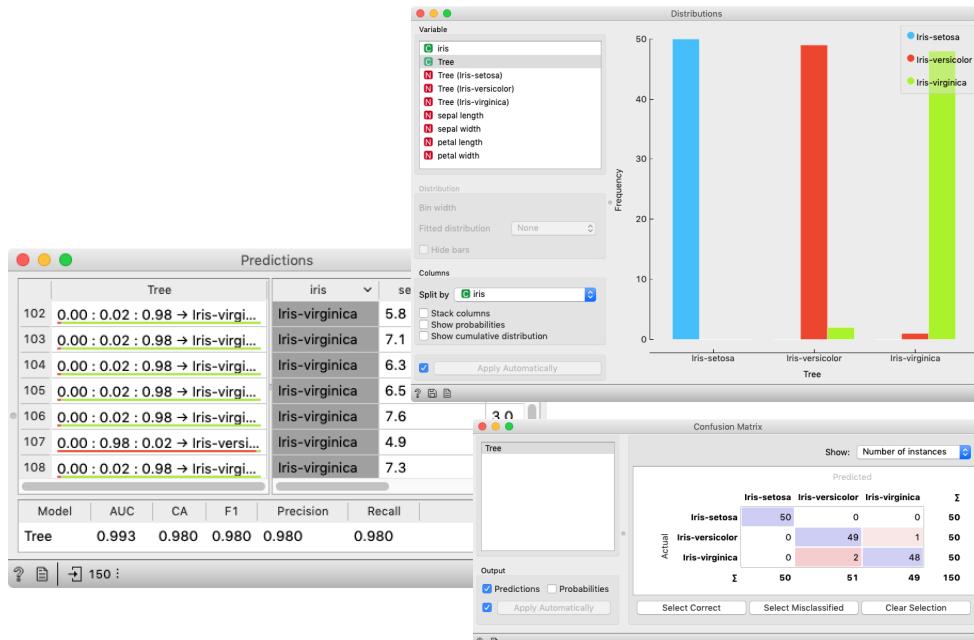
Now that we know what classification trees are, the next question is what is the quality of their predictions. For beginning, we need to define what we mean by quality. In classification, the simplest measure of quality is classification accuracy expressed as the proportion of data instances for which the classifier correctly guessed the value of the class. Let's see if we can estimate, or at least get a feeling for, classification accuracy with the widgets we already know.



features in view appropriately.

For precise statistics of correctly and incorrectly classified examples open the *Confusion Matrix* widget.

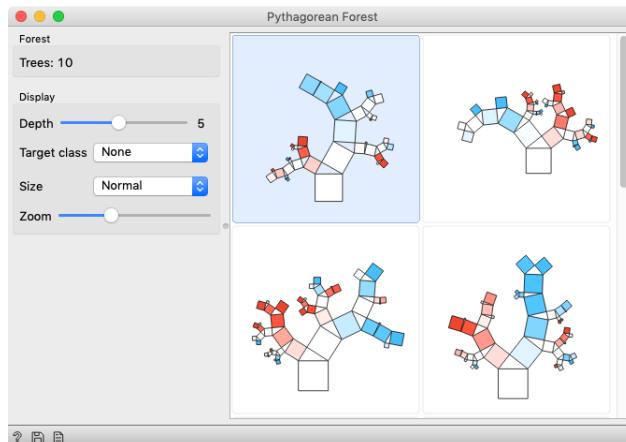
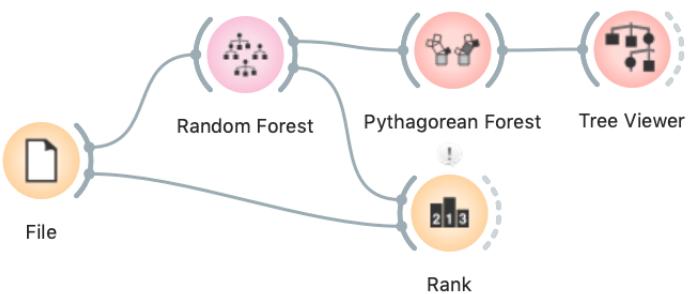
Let us try this schema with the *iris* data set. The *Predictions* widget outputs a data table augmented with a column that includes predictions. In the *Data Table* widget, we can sort the data by any of these two columns, and manually select data instances where the values of these two features are different (this would not work on big data). Roughly, visually estimating the accuracy of predictions is straightforward in the *Distribution* widget, if we set the fea-



The *Confusion Matrix* shows 3 incorrectly classified examples, which makes the accuracy  $(150 - 3)/150 = 98\%$ .

# Random Forests

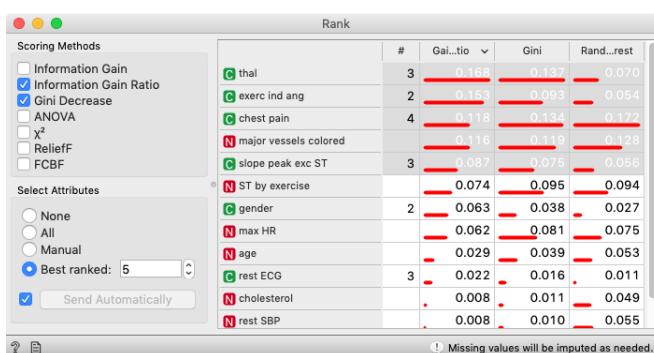
Random forests, a modeling technique introduced in 2001, is still one of the best performing classification and regression techniques. Instead of building a tree by always choosing the one feature that seems to separate best at that time, it builds many trees in slightly random ways. Therefore the induced trees are different. For the final prediction the trees vote for the best class.



The *Pythagorean Forest* widget shows us how random the trees are. If we select a tree, we can observe it in a *Tree Viewer*.

There are two sources of randomness: (1) training data is sampled with replacement, and (2) the best feature for a split is chosen among a subset of randomly chosen features.

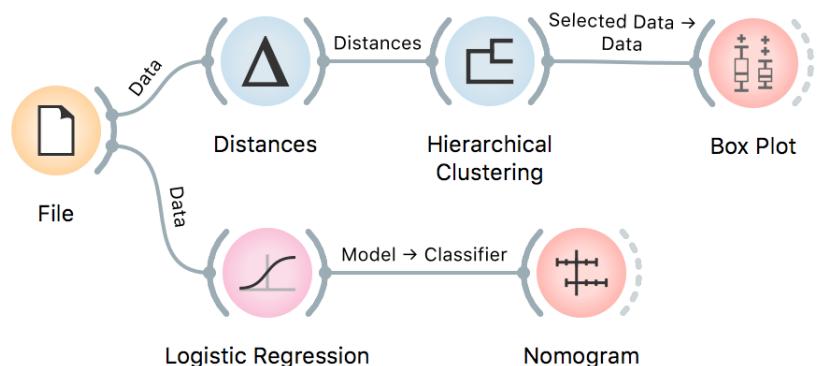
Which features are the most important? The creators of random forests also defined a procedure for computing feature importances from random forests. In Orange, you can use it with the *Rank* widget.



Feature importance according to two univariate measures (gain ratio and gini index) and random forests. Random forests also consider combinations of features when evaluating their importance.

# *Assignment:* *Clustering vs. Classification*

CLUSTERING AND CLASSIFICATION ARE FUNDAMENTALLY DIFFERENT TASKS. The former tries to find similar data instances and puts them into groups or clusters. The latter looks for the patterns in the data and infers correlations between the features and the target variable. Clustering is a part of unsupervised learning and doesn't require a target variable. Classification is a part of supervised learning and requires a target variable. A target variable is the thing we want to predict. With clustering, we are not predicting anything, but trying to find groups of similar data instances.



For this task, use *heart-disease* data and build a simple hierarchical clustering workflow (with Euclidean distance and Ward linkage) and a logistic regression model and inspect it in a nomogram. In clustering, cut at two clusters. Refer to the workflow above and answer the following questions:

1. Explore the groups with *Box Plot*. What is the characteristic of cluster 1 ( $C_1$ )? What is the characteristic of cluster 2 ( $C_2$ )?
2. Which are the top three attributes distinguishing between clusters? Write them down.
3. In *Nomogram*, which are the three most important attributes for the model?
4. Are the attributes from clustering the same as those from logistic regression? Why (not)?
5. *Major vessels colored* is the most important attribute for logistic regression. How well does it split between the clusters?

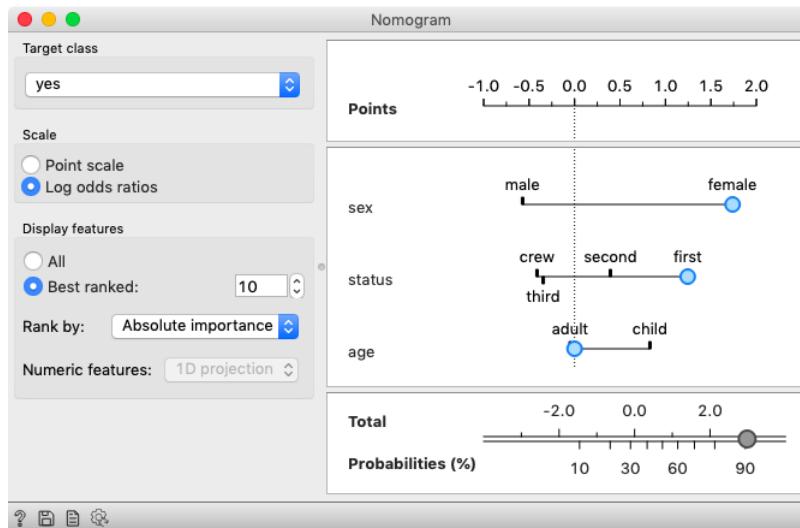
# Naive Bayes

Naive Bayes is also a classification method. To see how naive Bayes works, we will use a data set on passengers' survival in the Titanic disaster of 1912. The *Titanic* data set describes 2201 passengers, with their tickets (first, second, thirds class or crew), age and gender.



We inspect naive Bayes models with the *Nomogram* widget. There, we see a scale 'Points' and scales for each feature. Below we can see probabilities. Note the 'Target class' in upper left corner. If it is set to 'yes', the widget will show the probability that a passenger survived.

The nomogram shows that gender was the most important feature for survival. If we move the blue dot to 'female', the survival probability increases to 73%. Furthermore, if that woman also travelled in the first class, she survived with probability of 90%. The bottom scales show the conversion from feature contributions to probability.



Naive Bayes assumes class-wise independent features. For a data set where features would actually be independent, which rarely happens in practice, the naive Bayes would be the ideal classifier.

According to the probability theory individual contributions should be multiplied. Nomograms get around this by working in a log-space: a sum in the log-space is equivalent to multiplication in the original space. Therefore nomograms sum contributions (in the log-space) of all feature values and then convert them back to probability.

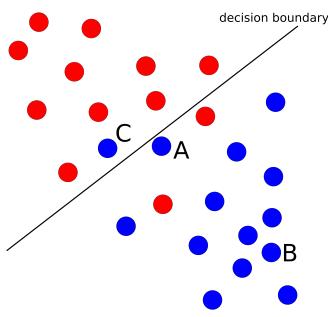
# Logistic Regression

Logistic regression is one of the best-known classifiers. The model returns the probability of a class variable, based on input features. First, it computes probabilities with a one-versus-all approach, meaning that for a multiclass problem, it will take one target value and treat all the rest as "other", effectively transforming the problem to binary classification.

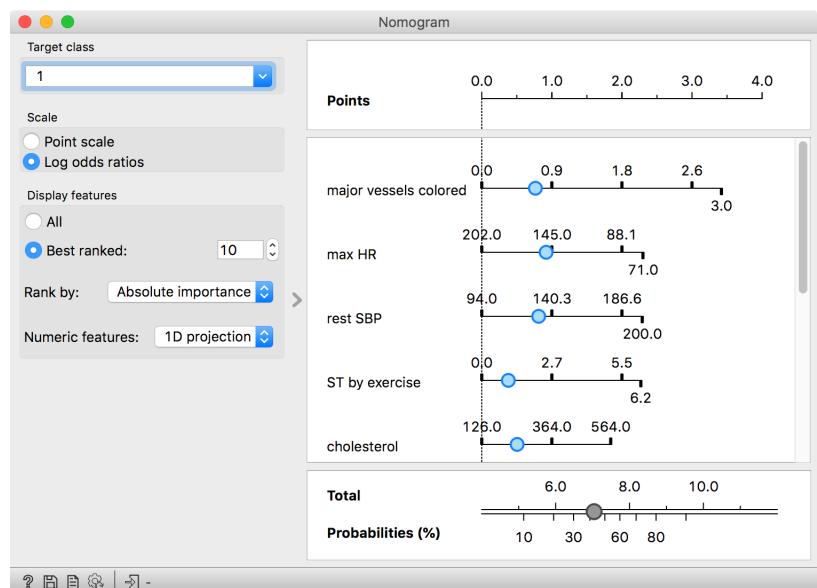
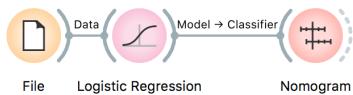
Second, it tries to find an optimal plane that separates instances with the target value from the rest. Then it uses logistic function to transform the distance to the plane into probabilities. The further away from the plane an instance will be, the higher the probability it belongs to the class on that side of the plane. The closer it is to the decision boundary (the plane), the more uncertain the prediction becomes (i.e. it gets close to 0.5).

Logistic regression tries to find such a plane that all points from one class are as far away from the boundary (in the correct direction) as possible.

A great thing about *Logistic Regression* is that we can interpret it with a *Nomogram*. Nomogram shows the importance of variables for the model. The higher the variable is in the list, the greater its importance. Also, the longer the line, the greater the importance. The line corresponds to the coefficient of the variable, which is then mapped to the probability. You can drag the blue point on the line left or right, decreasing or increasing the probability of the target class. This will show you how different values affect the outcome of the model.



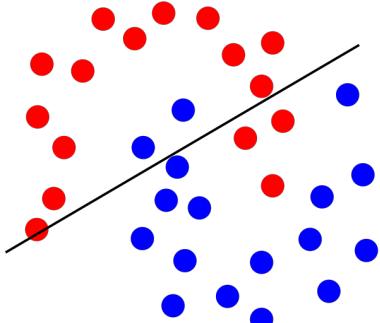
Can you guess what would the probability for belonging to the blue class be for A, B, and C?



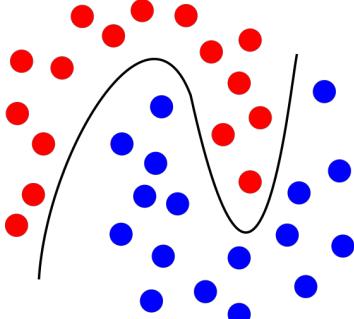
Another characteristic of logistic regression is that it observes all variables at once and takes the correlation into account. If some variables are correlated, their importance will be spread among them.

A not so great thing about logistic regression is that it operates with planes, meaning that the model won't work when the data cannot be separated in such a way. Can you think of such a data set?

# Support Vector Machines



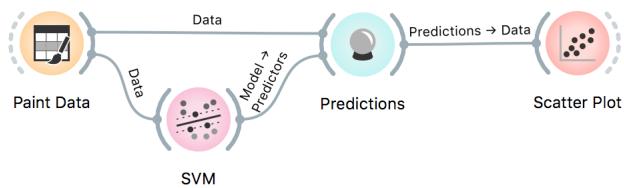
Decision boundary of a linear regression classifier.



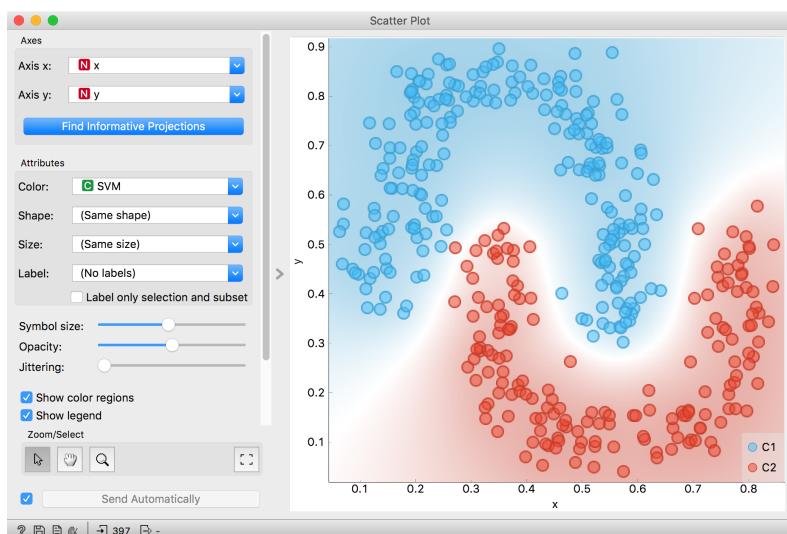
Decision boundary of a support vector machine classifier with an RBF kernel.

Support vector machines (SVM) are another example of linear classifiers, similar to logistic or linear regression. However, SVM can overcome splitting the data by a plane by using the so-called *kernel trick*. This means the hyperplane (decision boundary) can be transformed to a higher-dimensional space, which can fit the data nicely. In such a way, SVM becomes a non-linear classifier and can fit more complex data sets.

The magic of SVM (and other methods that can use kernels, and are thus called kernel methods) is that they will implicitly find a transformation into a (usually infinite-dimensional) space, in which the distances between objects are such as prescribed by the kernel, and draw a hyperplane in this space.



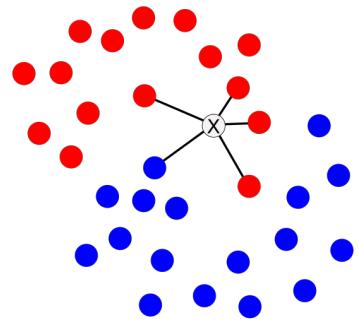
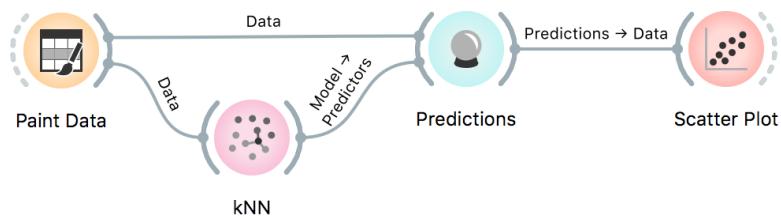
Abstract talking aside, SVM with different kernels can split the data not by ordinary hyperplanes, but with more complex curves. The complexity of the curve is decided by the kernel type and by the arguments given to the algorithm, like the degree and coefficients, and the penalty for misclassifications.



# *k*-Nearest Neighbors

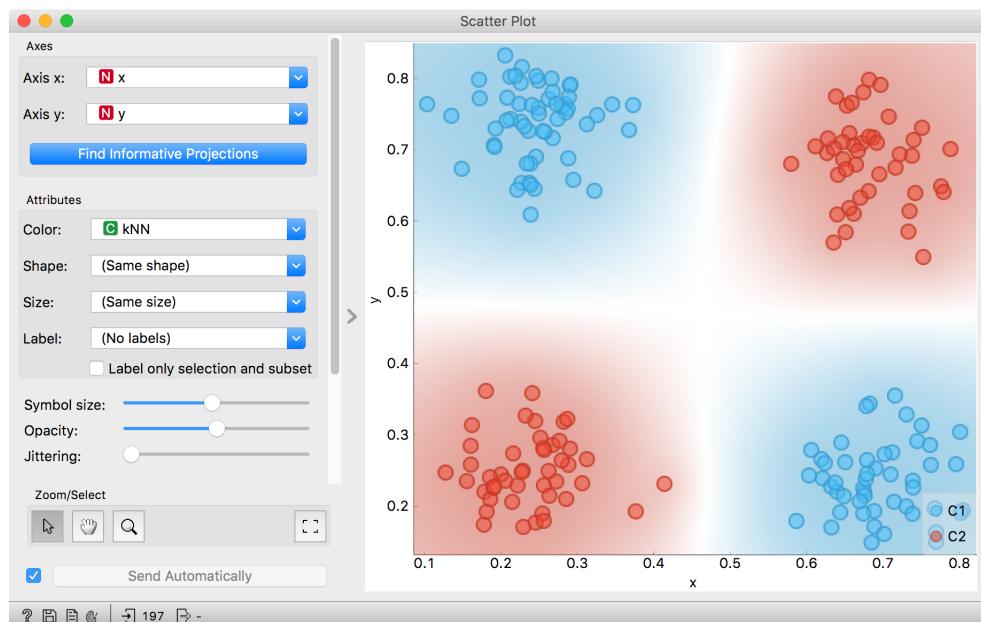
The idea of *k*-nearest neighbors is simple - find *k* instances that are the most similar to each data instance. We make the prediction or estimate probabilities based on the classes of these *k* instances. For classification, the final label is the majority label of *k* nearest instances. For regression, the final value is the average value of *k* nearest instances.

Unlike most other algorithms, *kNN* does not construct a model but just stores the data. This kind of learning is called *lazy learning*.



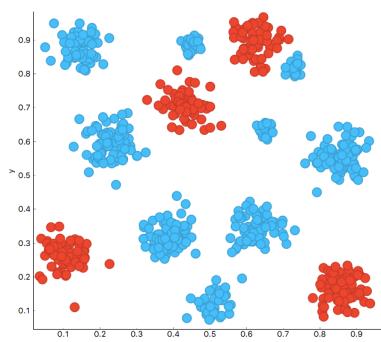
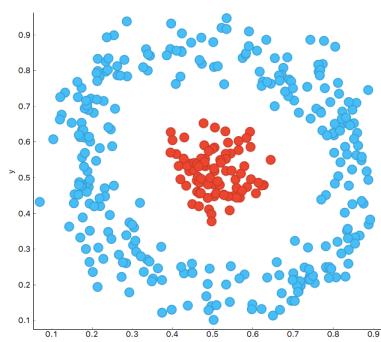
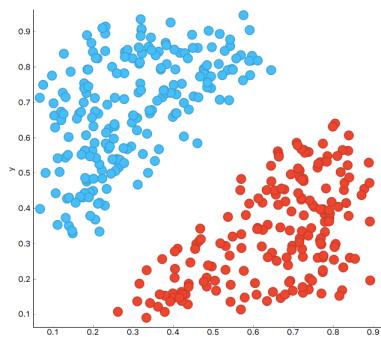
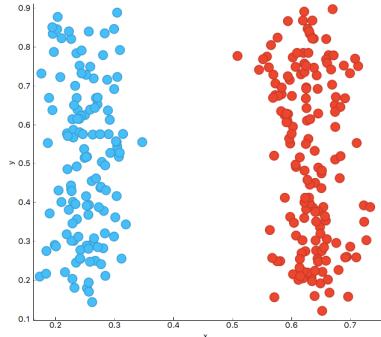
*kNN* classifier looks at *k* nearest neighbors, say 5, of instance X. 4 neighbors belong to the red class and 1 to the blue class. X will thus be classified as red with 80% probability.

The advantage of *kNN* algorithm is that it can successfully model the data, where classes are not linearly separable. It can also be re-trained quickly, because new data instances effect model only locally. However, the first training is can be slow for large data sets, as the model has to estimate *k* distances for data instance.

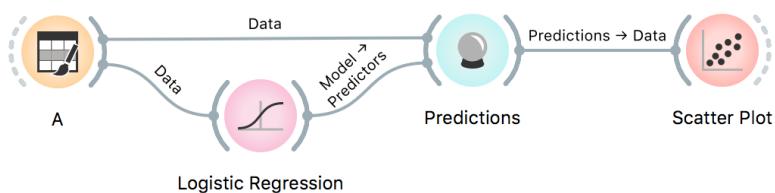


# Assignment: Decision Boundaries

You can try painting the data yourself or download it from [here](#).



CLASSIFIERS COME IN ALL SHAPES AND SIZES. What we mean by that is that each has its own way of learning from the data, its own strengths and weaknesses. Knowing how classifiers work is crucial for selecting the right algorithm for your task.



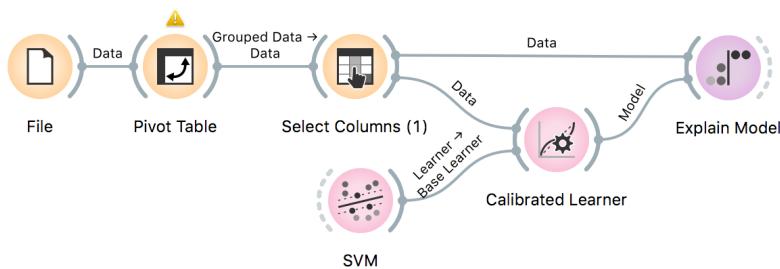
Try *Tree*, *Logistic Regression*, *SVM*, *Random Forest*, and *kNN* classifiers:

1. Which classifiers work well with data set A? Which with B, C, and D?
2. Which classifier is struggling the most? Which one the least? Why?
3. Look at the Tree with *Tree Viewer* for data set C. What do you notice?
4. In the above example, you can separate classes with a single stroke of a pen. Now limit Tree depth by setting *Limit maximal tree depth to 2*, which replicates drawing a single line to separate the classes. What do you notice? What happens, when you increase the depth of the tree?

# Assignment: Model Explanation

## UNDERSTANDING THE MODEL IS ESSENTIAL FOR DECISION-MAKING.

We are back to FTO<sub>3</sub> data set. We have already built several models and evaluated their performance. But for reaching any kind of decisions, it is important to understand the model - what it does, which features are important, and in what way. We will continue working with the mean batch data.



1. What is the first thing you notice about the model? Hint: look at the distribution of the data.
2. What are the top three attributes of the model? How are they interpreted?
3. What is the main production decision that you can make after observing the model?

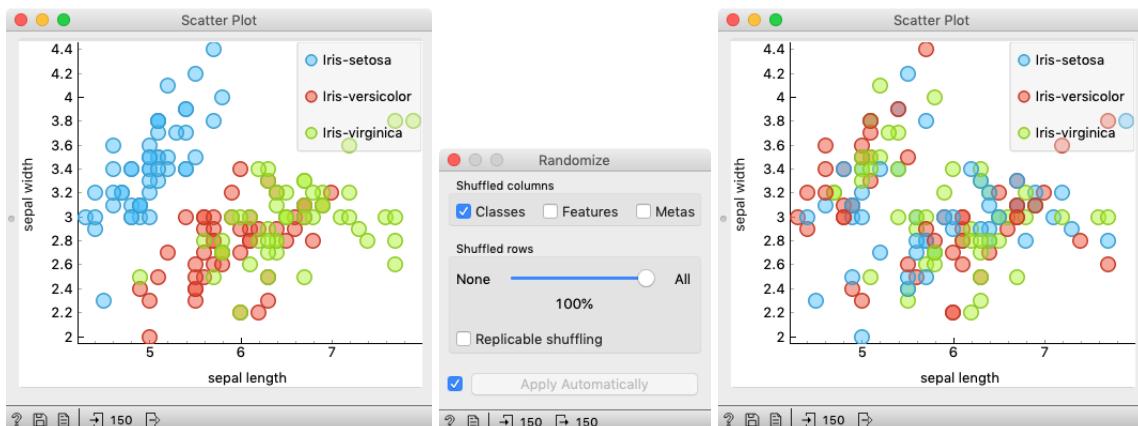
This lesson has a strange title and it is not obvious why it was chosen. Maybe you, the reader, should tell us what this lesson has to do with cheating.

## How to Cheat



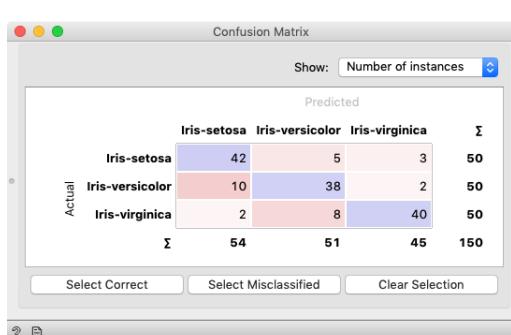
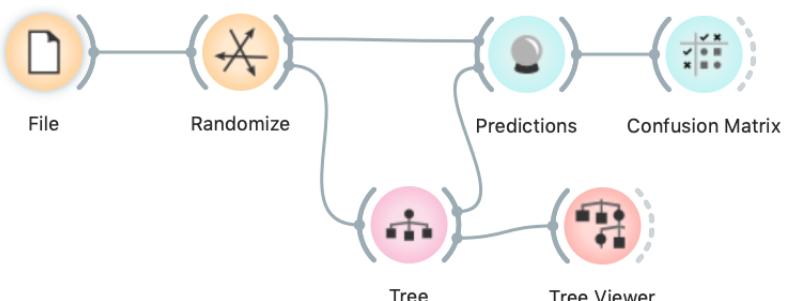
At this stage, the classification tree looks very good. There's only one data point where it makes a mistake. Can we mess up the data set so bad that the trees will ultimately fail? Like, remove any existing correlation between features and the class?

We can! There's the *Randomize* widget with class shuffling. Check out the chaos it creates in the *Scatter Plot* visualization where there were nice clusters before randomization!



Left: scatter plot of the *Iris* data set before randomization; right: scatter plot after shuffling 100% of rows.

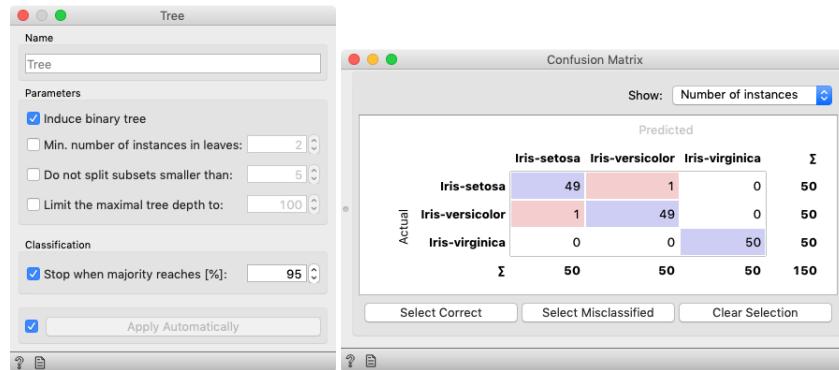
Fine. There can be no classifier that can model this mess, right? Let's make sure.



And the result? Here is a screenshot of the *Confusion Matrix*.

Most unusual. Despite shuffling all the classes, which destroyed any connection between features and the class variable, about 80% of predictions were still correct.

Can we further improve accuracy on the shuffled data? Let us try to change some properties of the induced trees: in the *Tree* widget, disable all early stopping criteria.



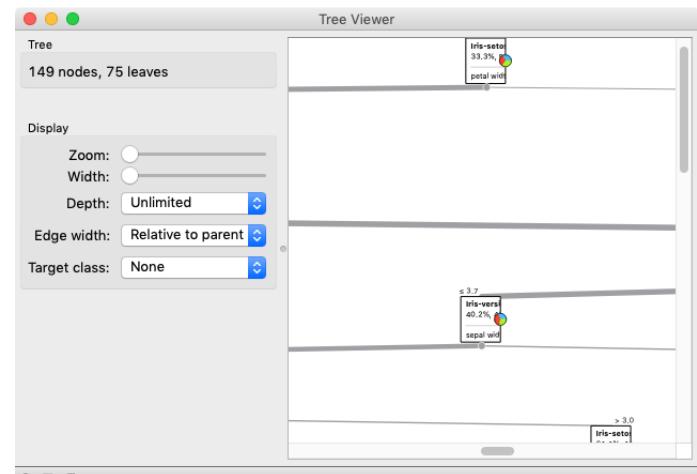
After we disable 2-4 check box in the *Tree* widget, our classifier starts behaving almost perfectly.

Wow, almost no mistakes now. How is this possible? On a class-randomized data set?

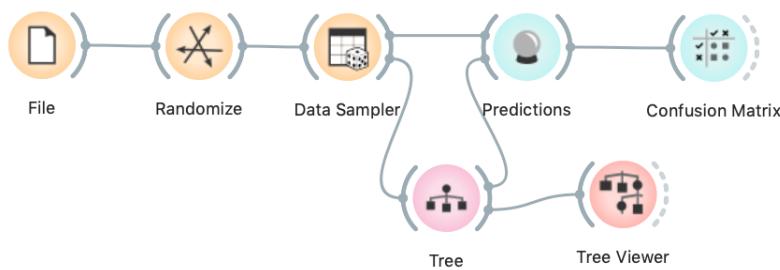
To find the answer to this riddle, open the *Tree Viewer* and check out the tree. How many nodes does it have? Are there many data instances in the leaf nodes?

Looks like the tree just memorized every data instance from the data set. No wonder the predictions were right. The tree makes no sense, and it is complex because it simply remembered everything.

Ha, if this is so, if a classifier remembers everything from a data set but without discovering any general patterns, it should perform miserably on any new data set. Let us check this out. We will split our data set into two sets, training and testing, train the classification tree on the training data set and then estimate its accuracy on the test data set.



In the build tree, there are 75 leaves. Remember, there are only 150 rows in the Iris data set.



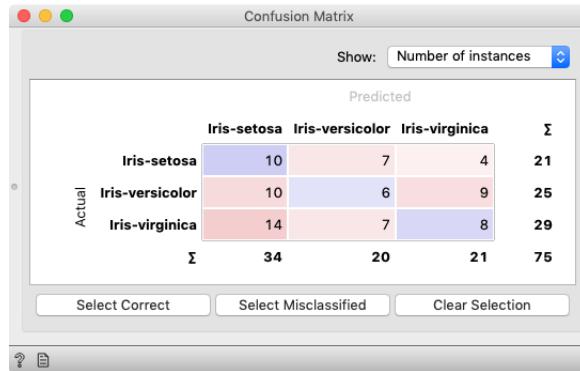
Let's check how the *Confusion Matrix* looks after testing the classifier on the test data.

The first two classes are a complete fail. The predictions for ribosomal genes are a bit better, but still with lots of mistakes. On the

Connect the *Data Sampler* widget carefully. The *Data Sampler* splits the data to a sample and out-of-sample (so called remaining data). The sample was given to the *Tree* widget, while the remaining data was handed to the *Predictions* widget. Set the *Data Sampler* so that the size of these two data sets is about equal.

class-randomized training data our classifier fails miserably. Finally, just as we would expect.

Confusion matrix if we estimate accuracy on a data set that was not used in learning.



We have just learned that we need to train the classifiers on the training set and then test it on a separate test set to really measure performance of a classification technique. With this test, we can distinguish between those classifiers that just memorize the training data and those that actually learn a general model.

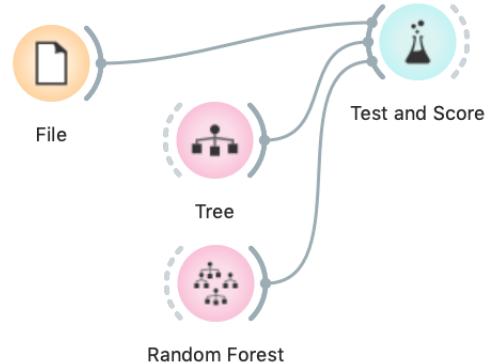
Learning is not only memorizing. Rather, it is discovering patterns that govern the data and apply to new data as well. To estimate the accuracy of a classifier, we therefore need a separate test set. This estimate should not depend on just one division of the input data set to training and test set (here's a place for cheating as well). Instead, we need to repeat the process of estimation several times, each time on a different train/test set and report on the average score.

# Cross-Validation

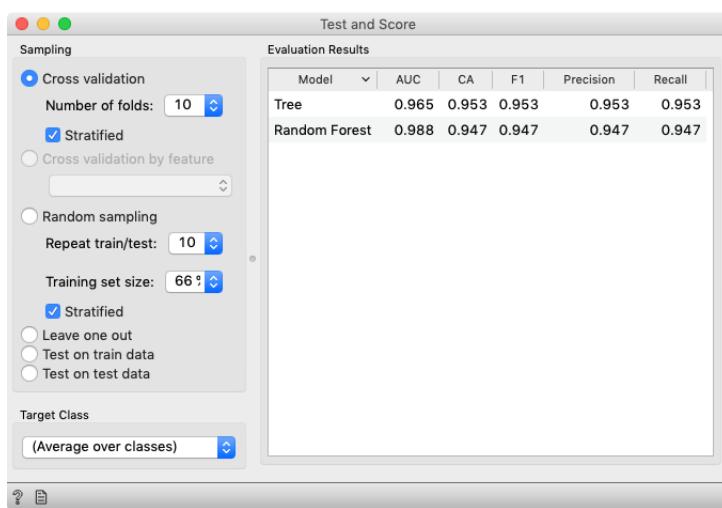
Estimating the accuracy may depend on a particular split of the data set. To increase robustness, we can repeat the measurement several times, each time choosing a different subset of the data for training. One such method is cross-validation. It is available in Orange in the *Test and Score* widget.

Note that in each iteration, *Test and Score* will pick a part of the data for training, learn the predictive model on this data using some machine learning method, and then test the accuracy of the resulting model on the remaining, test data set. For this, the widget will need on its input a data set from which it will sample the data for training and testing, and a learning method which it will use on the training data set to construct a predictive model. In Orange, the learning method is simply called a learner. Hence, *Test and Score* needs a learner on its input.

This is another way to use the *Tree* widget. In the workflows from the previous lessons we have used another of its outputs, called *Model*; its construction required data. This time, no data is needed for *Tree*, because all that we need from it is a *Learner*.



For geeks: a learner is an object that, given the data, outputs a classifier. Just what *Test and Score* needs.



In the *Test and Score* widget, the second column, CA, stands for classification accuracy, and this is what we really care for now.

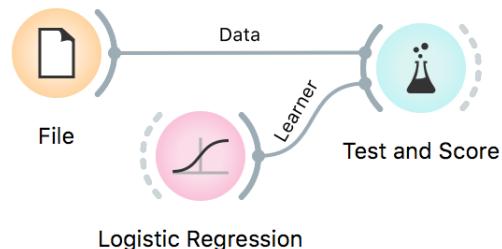
Cross validation splits the data sets into, say, 10 different non-overlapping subsets we call folds. In each iteration, one fold will be used for testing, while the data from all other folds will be used for training. In this way, each data instance will be used for testing exactly once.

# Assignment: Overfitting

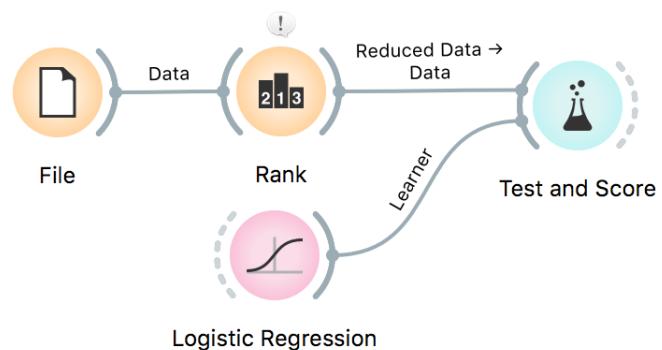
To load the blood loneliness data set copy and paste the below URL to the URL field of the File widget.

<http://file.biolab.si/datasets/blood-loneliness-GDS3898.tab>

OVERFITTING IS SOMETHING WE TRY TO AVOID AT ALL TIMES. But overfitting comes in many shapes and sizes. For this exercise we will use a *blood-loneliness* data set with the File widget. This data set relates gene expressions in blood with a measure of loneliness obtained from a sample of elderly persons. Let's try to model loneliness with logistic regression and see how well the model performs.



1. Train the Logistic Regression model on the data and observe its performance. What is the result?
2. We have many features in our data. What if we select only the most relevant ones, the genes that actually matter? Use Rank to select the top 20 best performing features.



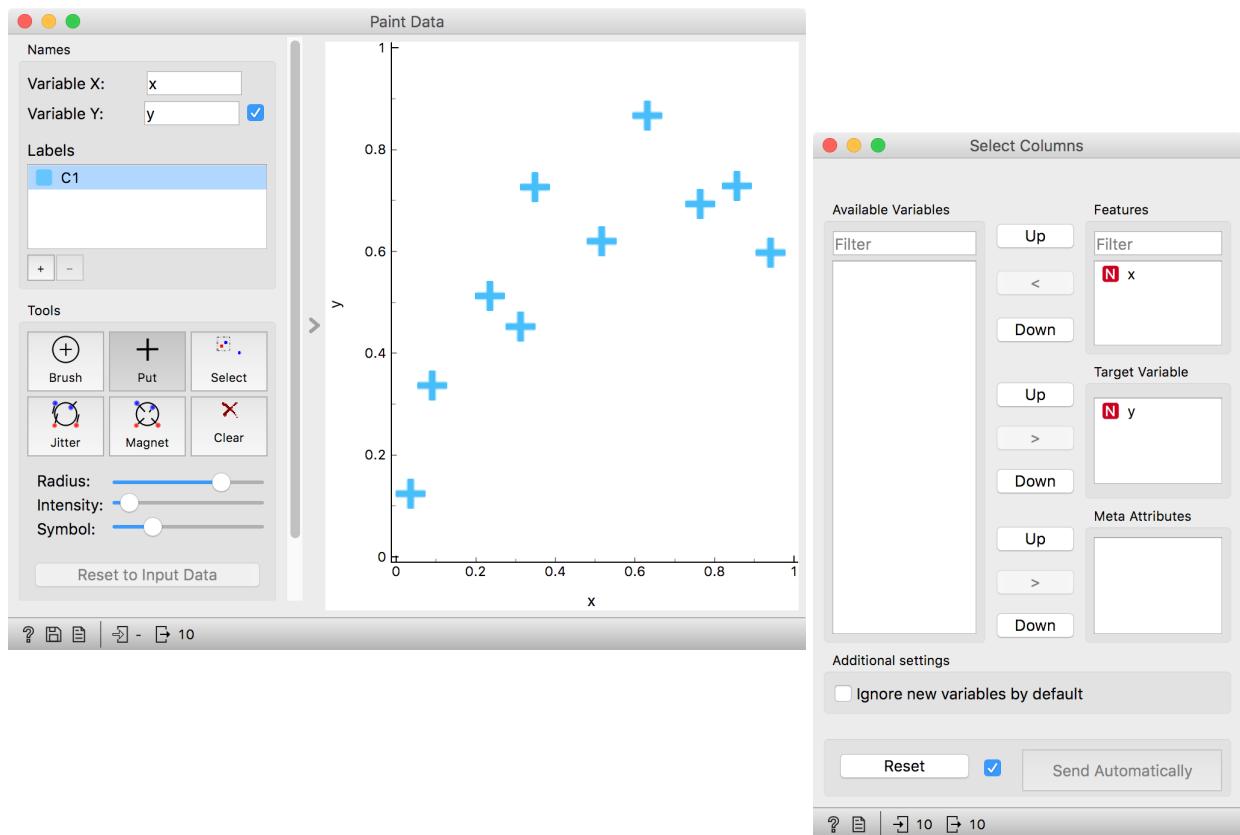
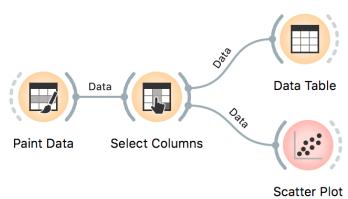
3. How are the results now? What happened? Is there a better way of performing feature selection?

# Linear Regression

For a start, let us construct a very simple data set. It will contain just one continuous input feature (let's call it  $x$ ) and a continuous class (let's call it  $y$ ). We will use Paint Data, and then reassign one of the features to be a class using *Select Columns* and moving the feature  $y$  from "Features" to "Target Variable". It is always good to check the results, so we are including *Data Table* and *Scatter Plot* in the workflow at this stage. We will be modest this time and only paint 10 points and will use Put instead of the Brush tool.

We would like to build a model that predicts the value of target variable  $y$  from the feature  $x$ . Say that we would like our model to be linear, to mathematically express it as  $h(x) = \theta_0 + \theta_1x$ . Oh, this is the equation of a line. So we would like to draw a line through our data points. The  $\theta_0$  is then an intercept and  $\theta_1$  is a slope. But there are many different lines we could draw. Which one is the best one? Which one is the one that fits our data the most? Are they the same?

In the *Paint Data* widget, remove the  $C_2$  label from the list. If you have accidentally left it while painting, don't despair. The class variable will appear in the *Select Columns* widget, but you can "remove" it by dragging it into the Available Variables list.

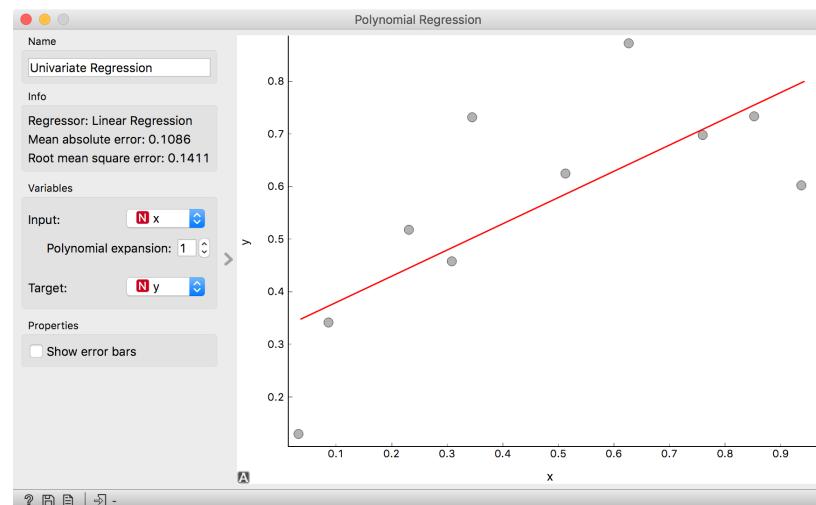


The question above requires us to define what a good fit is. Say, this could be the error the fitted model (the line) makes when it predicts the value of  $y$  for a given data point (value of  $x$ ). The prediction is  $h(x)$ , so the error is  $h(x) - y$ . We should treat the negative and

Do not worry about the strange name of the *Polynomial Regression*, we will get there in a moment.

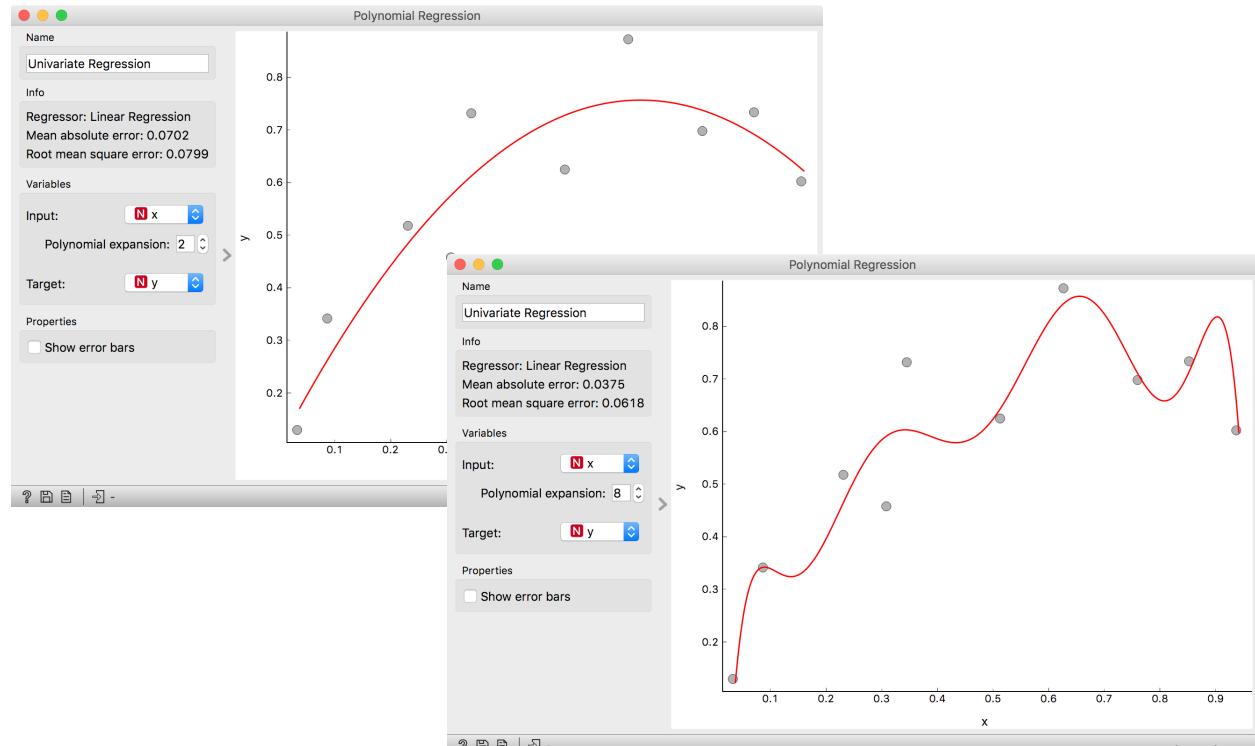


positive errors equally, plus, let us agree, we would prefer punishing larger errors more severely than smaller ones. Therefore, we should square the errors for each data point and sum them up. We got our objective function! Turns out that there is only one line that minimizes this function. The procedure that finds it is called linear regression. For cases where we have only one input feature, Orange has a special widget in the Educational add-on called *Polynomial Regression*.



Looks ok. Except that these data points do not appear exactly on the line. We could say that the linear model is perhaps too simple for our data set. Here is a trick: besides the column  $x$ , the widget *Polynomial Regression* can add columns  $x_2, x_3 \dots x_n$  to our data set. The number  $n$  is a degree of polynomial expansion the widget performs. Try setting this number to higher values, say to 2, and then 3, and then, say, to 8. With the degree of 3, we are then fitting the data to a linear function  $h(x) = \theta_0 + \theta_1x + \theta_2x^2 + \theta_3x^3$ .

The trick we have just performed (adding higher order features to the data table and then performing linear regression) is called polynomial regression. Hence the name of the widget. We get something reasonable with polynomials of degree 2 or 3, but then the results get really wild. With higher degree polynomials, we totally overfit our data.

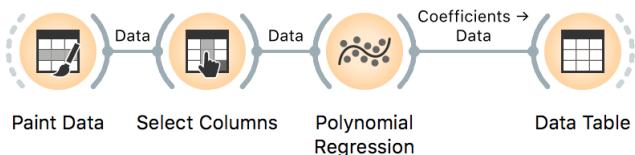


Overfitting is related to the complexity of the model. In polynomial regression, the models are defined through parameters  $\theta$ . The more parameters, the more complex the model. Obviously, the simplest model has just one parameter (an intercept), ordinary linear regression has two (an intercept and a slope), and polynomial regression models have as many parameters as is the degree of the polynomial. It is easier to overfit with a more complex model, as this can adjust to the data better. But is the overfitted model really discovering the true data patterns? Which of the two models depicted in the figures above would you trust more?

It is quite surprising to see that linear regression model can result in fitting non-linear (univariate) functions. That is, the functions with curves, such as those on the figures. How is this possible? Notice though that the model is actually a hyperplane (a flat surface) in the space of many features (columns) that are the powers of  $x$ . So for the degree 2,  $h(x) = \theta_0 + \theta_1 x + \theta_2 x^2$  is a (flat) hyperplane. The visualization gets curvy only once we plot  $h(x)$  as a function of  $x$ .

# Regularization

There has to be some cure for overfitting. Something that helps us control it. To find it, let's check what are the values of the parameters  $\theta$  under different degrees of polynomials.



With smaller degree polynomials values of  $\theta$  stay small, but then as the degree goes up, the numbers get really large.

Data Table		
	name	coef
1	1	0.106121
2	x	1.90152
3	x^2	-1.21305
4	x^3	-0.244903

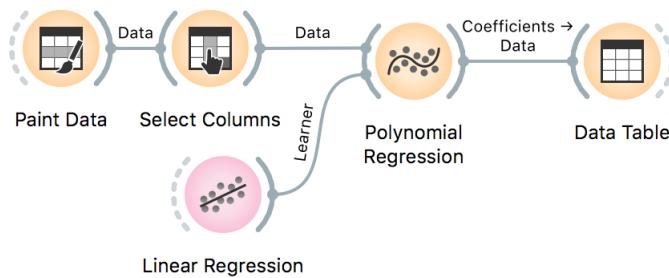
Data Table		
	name	coef
1	1	-0.787028
2	x	40.3077
3	x^2	-553.499
4	x^3	3756.01
5	x^4	-13830.3
6	x^5	29051.4
7	x^6	-34730.1
8	x^7	21961.7
9	x^8	-5696.56

Which inference of linear model would overfit more, the one with high  $\lambda$  or the one with low  $\lambda$ ? What should the value of  $\lambda$  be to cancel regularization? What if the value of  $\lambda$  is really high, say 1000?

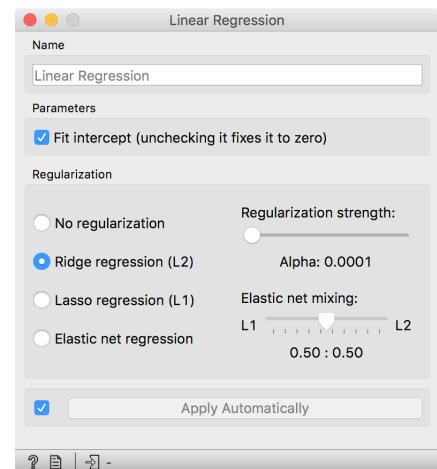
More complex models can fit the training data better. The fitted curve can wiggle sharply. The derivatives of such functions are high, and so need be the coefficients  $\theta$ . If only we could force the linear regression to infer models with a small value of coefficients. Oh, but we can. Remember, we have started with the optimization function the linear regression minimizes — the sum of squared errors. We could simply add to this a sum of all  $\theta$  squared. And ask the linear regression to minimize both terms. Perhaps we should weigh the part with  $\theta$  squared, say, with some coefficient  $\lambda$ , just to control the level of regularization.

Here we go: we just reinvented regularization, a procedure that helps machine learning models not to overfit the training data. To observe the effects of regularization, we can give Polynomial Regression our own learner, which supports these kind of settings.

The Linear Regression widget provides two types of regularization. Ridge regression is the one we have talked about and minimizes the sum of squared coefficients  $\theta$ . Lasso regression minimizes the sum of absolute value of coefficients. Although the difference may seem negligible, the consequences are that lasso regression may result in a large proportion of coefficients  $\theta$  being zero, in this way performing feature subset selection.



Internally, if no learner is present on its input, the Polynomial Regression widget would use just ordinary, non-regularized linear regression.

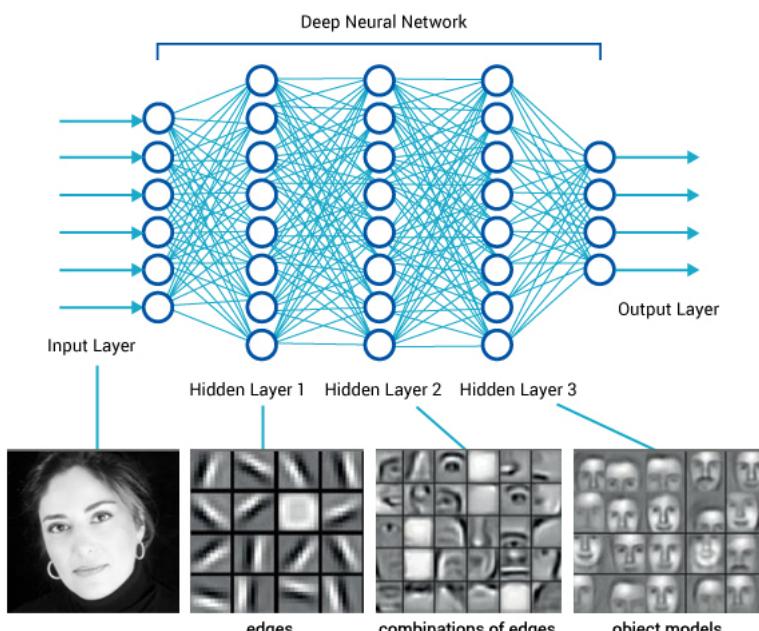


Now for the test. Increase the degree of polynomial to the max. Use Ridge Regression. Does the inferred model overfit the data? How does the degree of overfitting depend on regularization strength?

# Image Embedding

This depiction of deep learning network was borrowed from <http://www.amax.com/blog/?p=804>

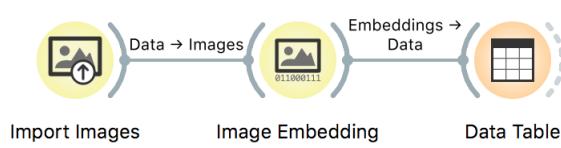
Every data set so far came in the matrix (tabular) form: objects (say, tissue samples, students, flowers) were described by row vectors representing a number of features. Not all the data is like this; think about collections of text articles, nucleotide sequences, voice recordings or images. It would be great if we could represent them in the same matrix format we have used so far. We would turn collections of, say, images, into matrices and explore them with the familiar prediction or clustering techniques.



Until very recently, finding useful representation of complex objects such as images was a real pain. Now, technology called deep learning is used to develop models that transform complex objects to vectors of numbers.

Consider images. When we, humans, see an image, our neural networks go from pixels, to spots, to patches, and to some higher order representations like squares, triangles, frames, all the way to representation of complex objects. Artificial neural networks used for deep learning emulate these through layers of computational units (essentially, logistic regression models and some other stuff we will ignore here).

If we put an image to an input of such a network and collect the outputs from the higher levels, we get vectors containing an abstraction of the image. This is called embedding.



Deep learning requires a lot of data (thousands, possibly millions of data instances) and processing power to prepare the network. We will use one which is already prepared. Even so, embedding takes time, so Orange doesn't do it locally but uses a server invoked through the *Image Embedding* widget.

Image embedding describes the images with a set of 2048 features appended to the table with meta features of images.

The screenshot shows a Data Table window with the following details:

- Info:** 19 instances (no missing data), 2048 features, No target variable, 5 meta attributes.
- Variables:**
  - Show variable labels (if present)
  - Visualize numeric values
  - Color by instance classes
- Selection:**  Select full rows
- Buttons:** Restore Original Order, Send Automatically

The main area displays a table with the following data:

hidden origin type	image name	image	size	width	height	n0 True	n1 True	n2 True	n3 True
1	duck	duck.png	39583	158	172	0.117296	0.0176356	0.106513	0.0349388
2	dog	dog.png	28745	129	125	0.0331701	0.199738	0.169014	0.215931
3	horse	horse.png	69109	285	195	0.409667	0.32446	0.0968603	0.152301
4	rabbit	rabbit.png	24294	97	174	0.229838	0.0299793	0.269629	0.00803636
5	ox	ox.png	56401	191	189	0.532831	0.00481718	0.166275	0.183236
6	turkey	turkey.png	55072	171	182	0.303828	0.0414567	0.287995	0.0756877
7	sheep	sheep.png	58022	214	181	0.232301	0.217721	0.0137615	0.0388852
8	cow	cow.png	62159	210	189	0.533668	0.127438	0.097885	0.0660376
9	calf	calf.png	45538	191	152	0.210981	0.13404	0.0892528	0.0554134
10	hen	hen.png	41716	134	168	0.571451	0.0563081	0.0841946	0.0408928
11	foal	foal.png	39210	147	177	0.105666	0.217703	0.014116	0.182355
12	cat	cat.png	22193	105	137	0.0566888	0.211721	0.671922	0
13	goose	goose.png	34442	141	202	0.246742	0.165718	0.134696	0
..	..	..	..	..	..	..	..	..	..

We have no idea what these features are, except that they represent some higher-abstraction concepts in the deep neural network (ok, this is not very helpful in terms of interpretation). Yet, we have just described images with vectors that we can compare and measure their similarities and distances. Distances? Right, we could do clustering. Let's cluster the images of animals and see what happens.

Images are available at <http://file.biolab.si/images/domestic-animals.zip>



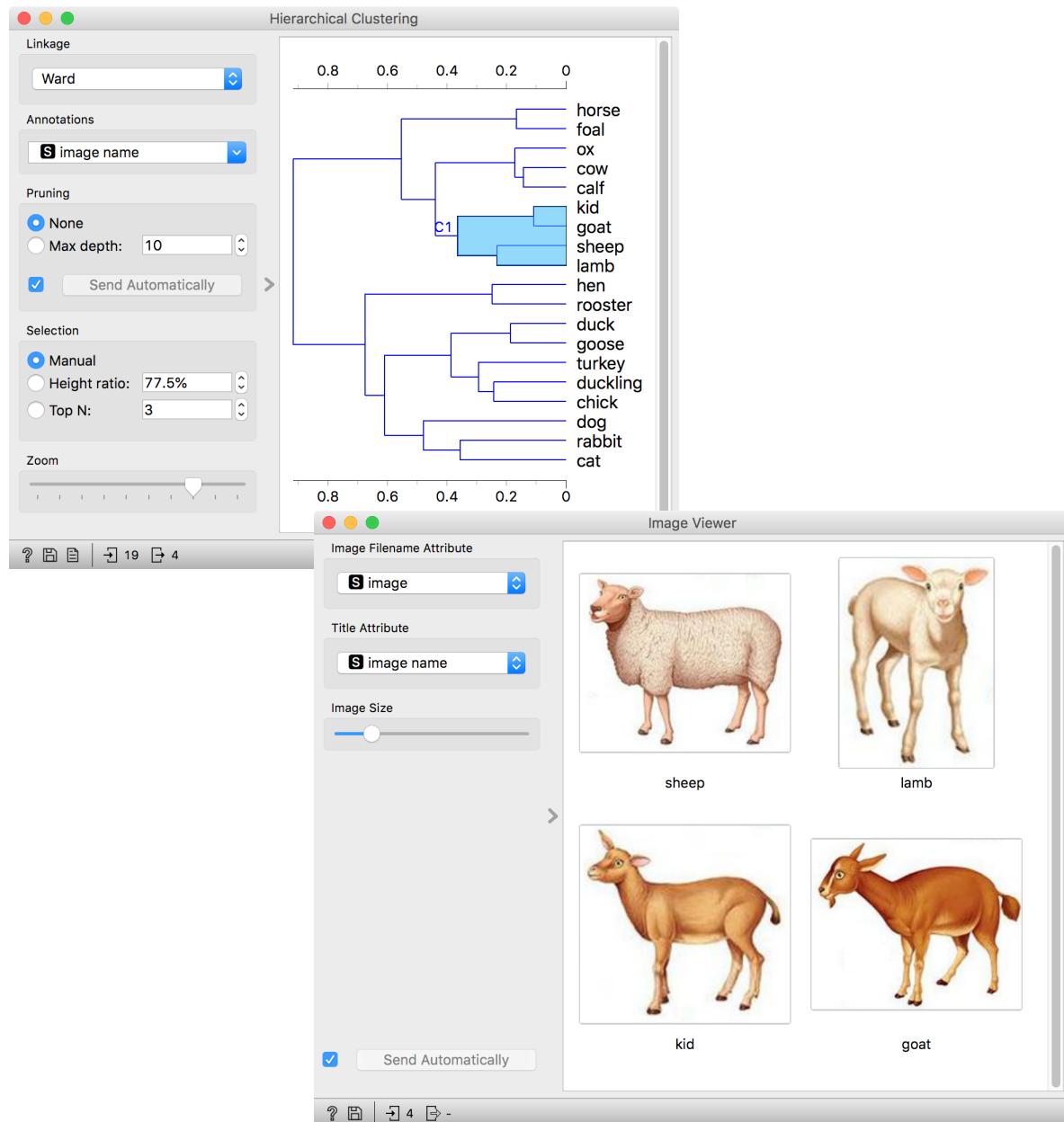
To recap: in the workflow above we have loaded the images from the local disk, turned them into numbers, computed the distance matrix containing distances between all pairs of images, used the distances for hierarchical clustering, and displayed the images that correspond to the selected branch of the dendrogram in the *Image Viewer*. We used cosine similarity to assess the distances (simply because the dendrogram looked better than with the Euclidean distance).

The screenshot shows a Distances window with the following settings:

- Distances between:** Rows (selected radio button)
- Distance Metric:** Cosine
- Normalized:** Unchecked
- Apply Automatically:** Checked

The bottom of the window shows standard file navigation buttons: ? (Help), ⌘ (File), ⌘ (Edit), 19 (Rows), ⌘ (Columns), and 19x19.

Even the lecturers of this course were surprised at the result. Beautiful!



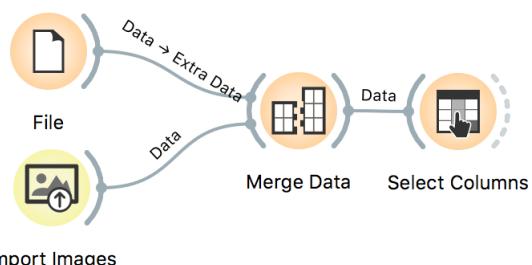
# Assignment: Image Analytics

ARCHAEOLOGISTS OFTEN ANALYZE IMAGES OF ARTEFACTS, from sketches of items to photographs of sites and findings. We retrieved 164 digital sketches of amphorae belonging to 3 different types - Dressel, Gauloise, and Keay. Consider we have an archive of such images and we wish to automatically label new sketches with their corresponding amphora type.

First, we load images with *Import Images* from the Image Analytics add-on. Our images are organized, each type of amphora being in its own folder. Folders will be treated as categories upon loading. This means each image will get a special label with the name of the folder - the target variable.

Second, we have metadata belonging to each subtype of amphora (Dressel 6b, Gauloise 13, etc.). Load this with *File* and join the two tables with *Merge Data*. The widget should automatically recognize the attribute to merge by, but if it doesn't, set both to *image name*. Finally, we will use *Select Columns* to put all the metadata to metas. Remember, our aim is to classify amphorae based on their image, not their description.

Images are available at <http://file.biolab.si/datasets/amphorae.zip>



The screenshot shows the Orange data mining interface. On the left, the 'Info' panel displays: 164 instances, 2048 features, Target with 3 values, and 18 meta attributes (5.9 % missing data). Under 'Variables', there are checkboxes for 'Show variable labels (if present)', 'Visualize numeric values', and 'Color by instance classes', all of which are checked. Under 'Selection', 'Select full rows' is checked. At the bottom, there are buttons for 'Restore Original Order' and 'Send Automatically'. The main area is a 'Data Table' with 164 rows and 8 columns. The columns are: hidden origin type, Base type, Capacity (avg l), n0, n1, n2, n3, and True. The 'Base type' column contains values like 'ringed', 'flat', and 'spike/tapered'. The 'Capacity (avg l)' column contains numerical values. The 'n0', 'n1', 'n2', and 'n3' columns contain True or False values. The 'True' column contains numerical values. The bottom status bar shows 164 instances and 164 features.

But machine learning works with numbers and images are not numbers, but complex objects. We need a way to transform images to a numeric description. We will use *Image Embedding*, a widget that uses pre-trained deep models to infer image profiles. A common embedder is Google's Inception v3, which we will use for this exer-

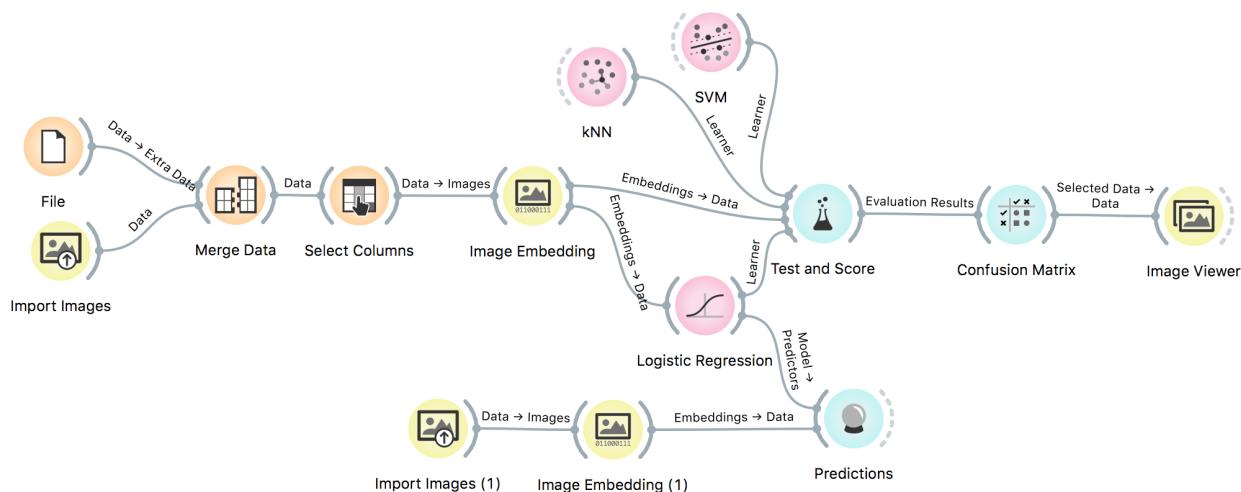
cise. Finally, we have our numbers! Specifically, a 2048-dimensional vector for each image.

Now, we can train a few classifiers. Common models for image classification are *kNN*, *Logistic Regression*, and *SVM*, so let's use these. It looks like *SVM* has the highest AUC score, but *Logistic Regression* performs the best in other metrics.

Evaluation Results						
Model	AUC	CA	F1	Precision	Recall	
kNN	0.942	0.823	0.818	0.828	0.823	
SVM	0.961	0.854	0.850	0.865	0.854	
Logistic Regression	0.949	0.878	0.878	0.879	0.878	

### Assignment

- Observe *Confusion Matrix* and determine which model is better to use in practice and why.
- Which type of amphorae is the most difficult to predict? How could we improve the performance? Select different misclassifications in Confusion Matrix and explore them in Image Viewer.
- There are three new images to predict. Use amphorae-predict and load it with Import Images. Pass it to Image Embedding, then use Predictions and the chosen model to predict image type. How accurate is the model? Would you use it in practice? Why (not)?



## *Bibliography*