DATA MINING:

A HANDS-ON COURSE AT BAYLOR COLLEGE OF MEDICINE

BIOINFORMATICS LAB, LJUBLJANA

We have designed this course for students and researchers of life sciences. These working notes include Orange workflows and visualizations we will construct during the lectures. Throughout our training, you will see how to accomplish various data mining tasks through visual programming and use Orange to build visual data mining workflows. Many similar data mining environments exist, but the lecturers prefer Orange for one simple reason—they are its authors.

If you haven't already installed Orange, please download the installation package from http://orangedatamining.com.

The notes were written by Blaž Zupan and Janez Demšar with massive help from the members of the Bioinformatics Lab in Ljubljana, Slovenia, that developed Orange. We would specifically like to thank Ajda Pretnar Žagar and Marko Toplak for proofreading, editing, and converting our previous documents in Pages to LaTeX.

PUBLISHED BY BIOINFORMATICS LAB, LJUBLJANA

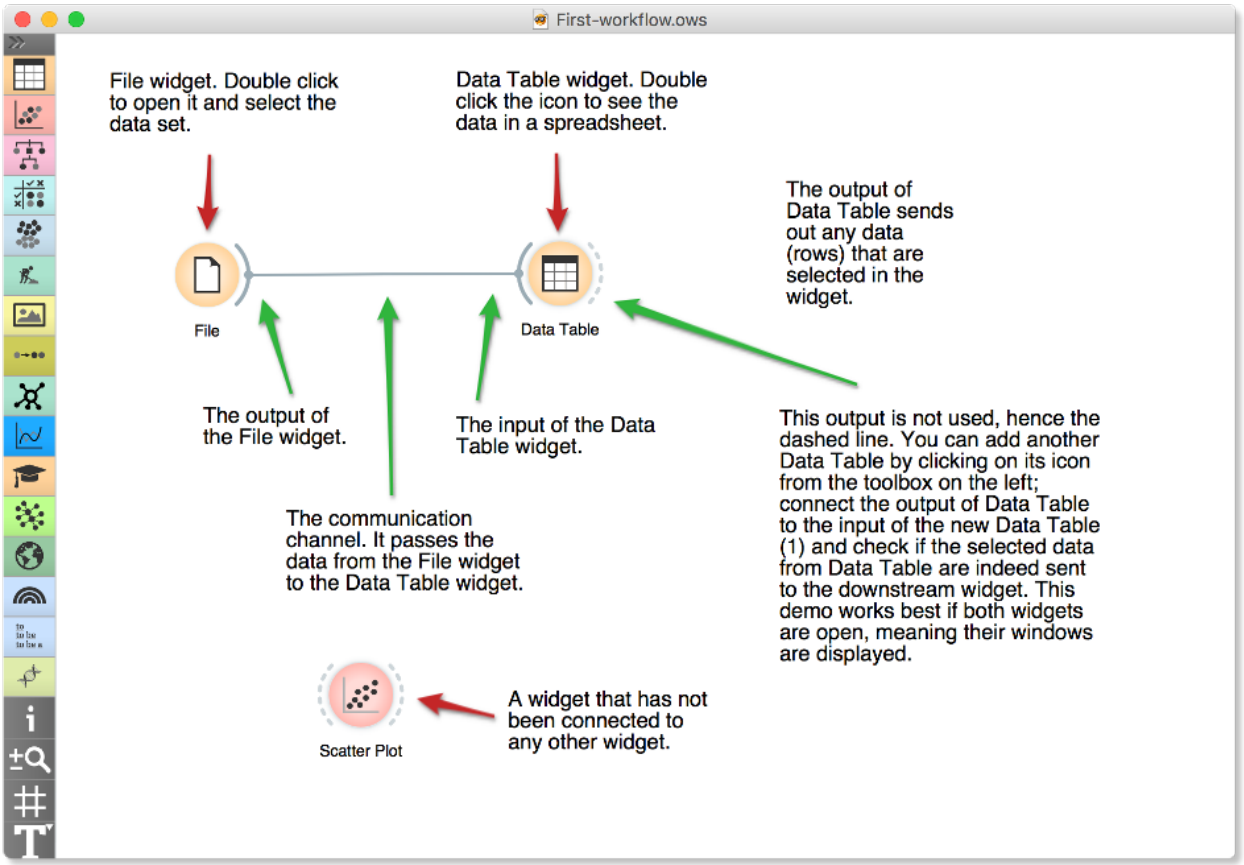
TUFTE-LATEX.GOOGLECODE.COM

First printing, February 2022

Contents

Workflows in Orange

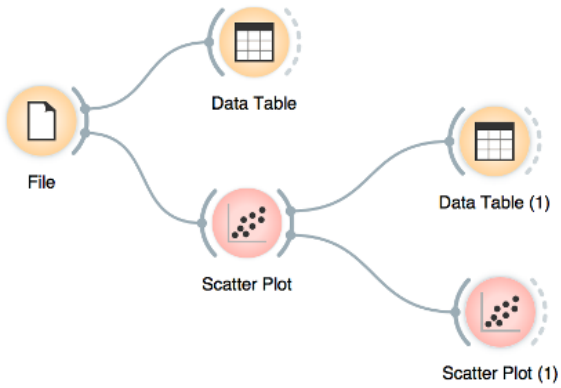
ORANGE WORKFLOWS consist of components that read, process, and visualize data. We refer to these components as "widgets" We place the widgets on a drawing board called the "canvas" to design a workflow. Widgets communicate by sending information along their communication channel. Output from one widget can be used as input to another.



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.

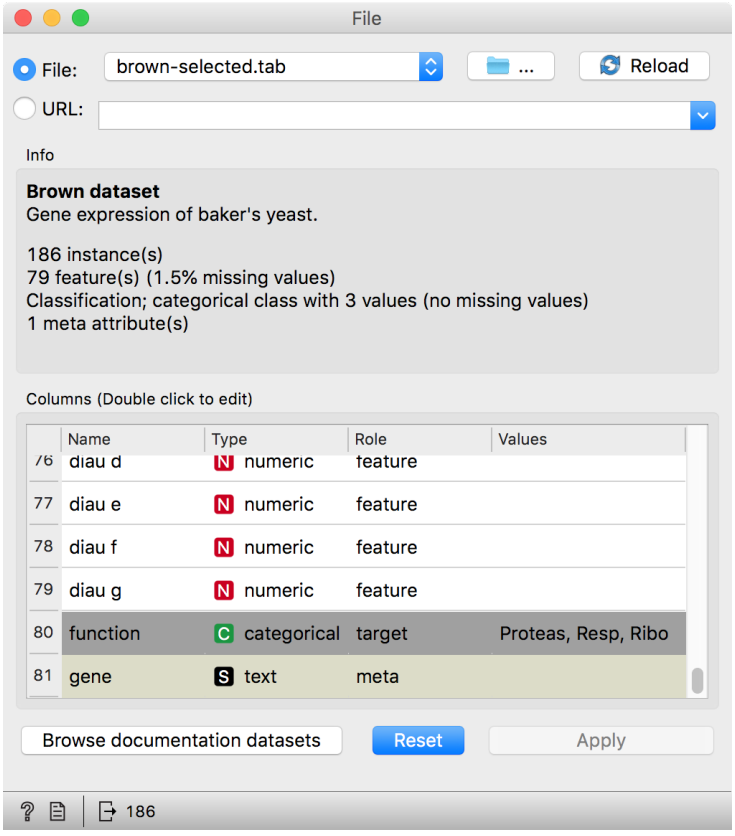
We construct workflows by dragging widgets onto the canvas and connecting them by drawing a line from the transmitting widget to the receiving widget. The widget's outputs are on the right, and the inputs on the left. In the workflow above, the *File* widget sends data to the *Data Table* widget.

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



A 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. The plot's selected data points 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 preloaded data sets. From these ("Browse documentation data sets..."), choose *brown-selected.tab*, a yeast gene expression data set.



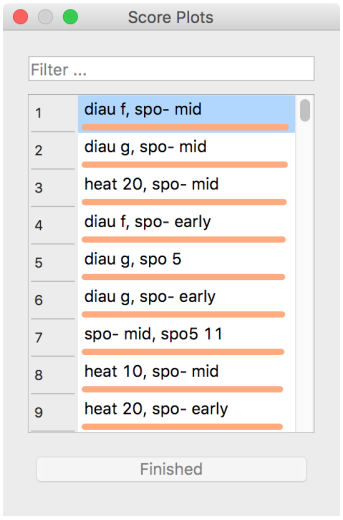
Orange workflows often start with a File widget. The brown-selected data set comprises 186 rows (genes) and 81 columns. Out of the 81 columns, 79 contain gene expressions of baker's yeast under various conditions, one column (marked as a "meta attribute") provides gene names, and one column contains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

After you load the data:

1. Open the other widgets.
2. Select a few data points in the *Scatter Plot* widget and watch as they appear in the *Data Table (1)*.

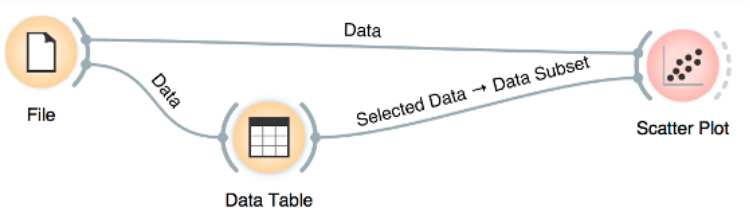
3. Use a combination of two *Scatter Plot* widgets, where the second scatter plot shows a detail from a smaller region selected in the first scatter plot.

The following is 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* at 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 helpful to a biologist?



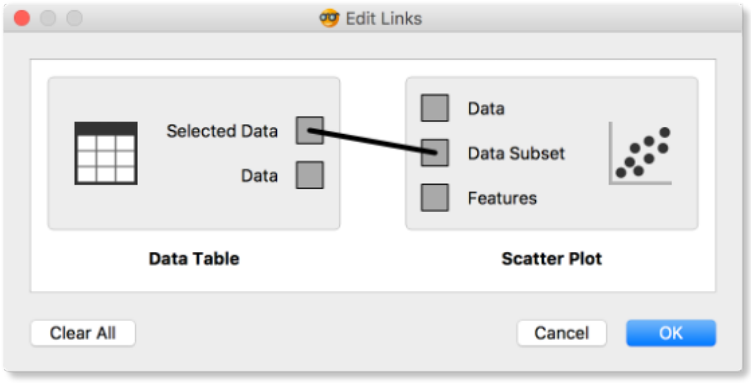
Scatter Plot and Ranking

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.



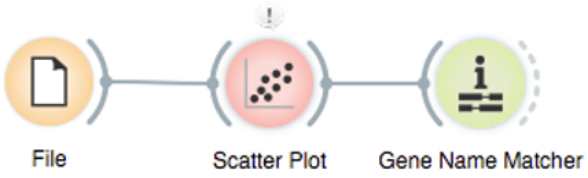
In this workflow, we have switched on the option "Show channel names between widgets" in File/Preferences.

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 add-ons. Check them out by selecting Add-ons... from the Options menu.

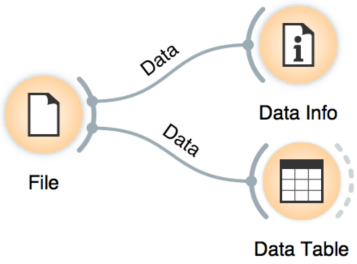


Basic data exploration

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.

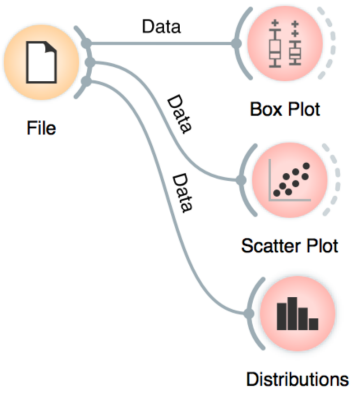
We will start with an empty canvas; to clean it from our previous lesson, use either File/New or select all the widgets and remove them (use the backspace/delete key).

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.

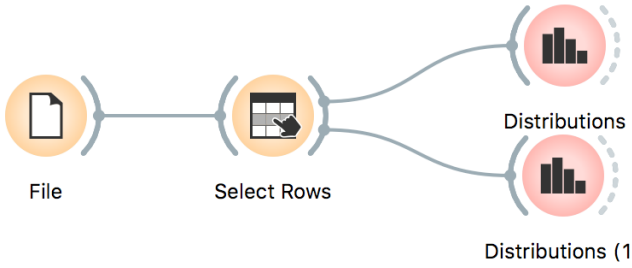
Let us check whether common visualizations tell us anything interesting. (Hint: look for gender differences. These are always interesting and occasionally even real.)



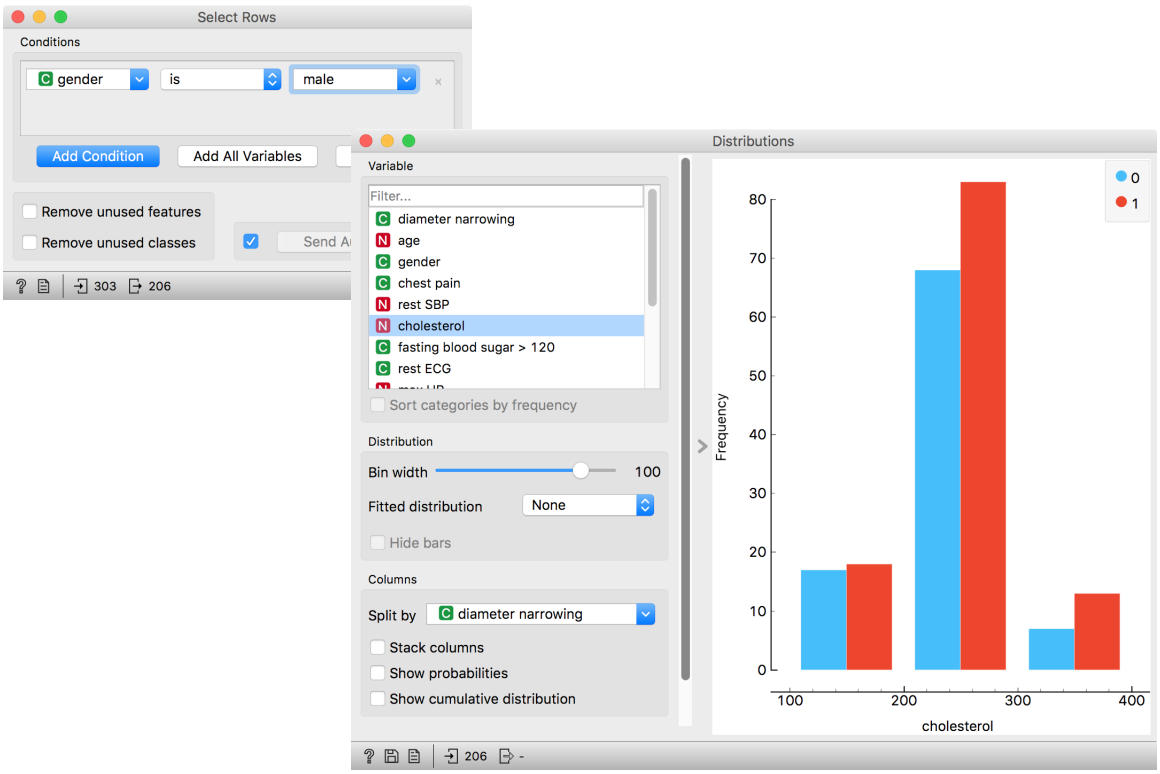
Quick check with common statistics and visualization widgets.

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.



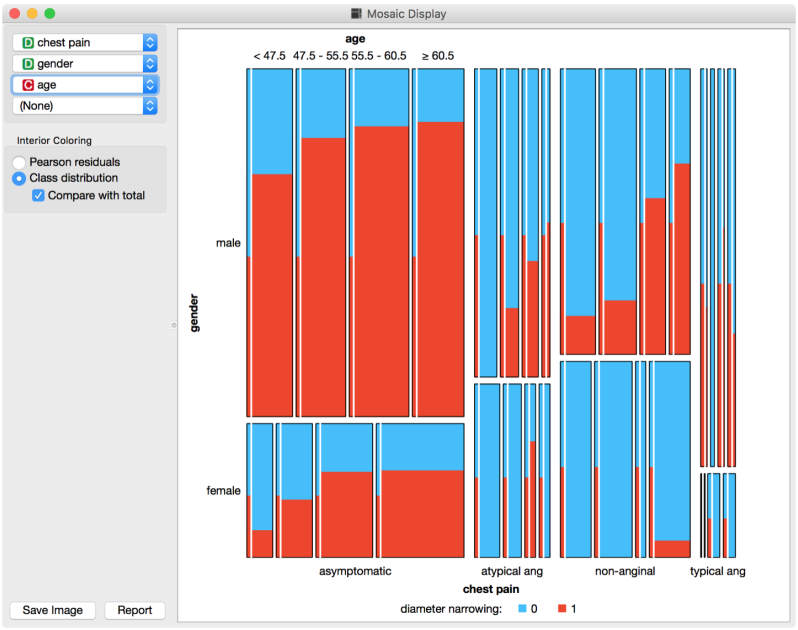
In the *Select Rows* widget, we select the female patients. You can also add other conditions. The selection of data instances provides a powerful combination with visualization of data distribution. Try having at least two widgets open simultaneously and explore the data.



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

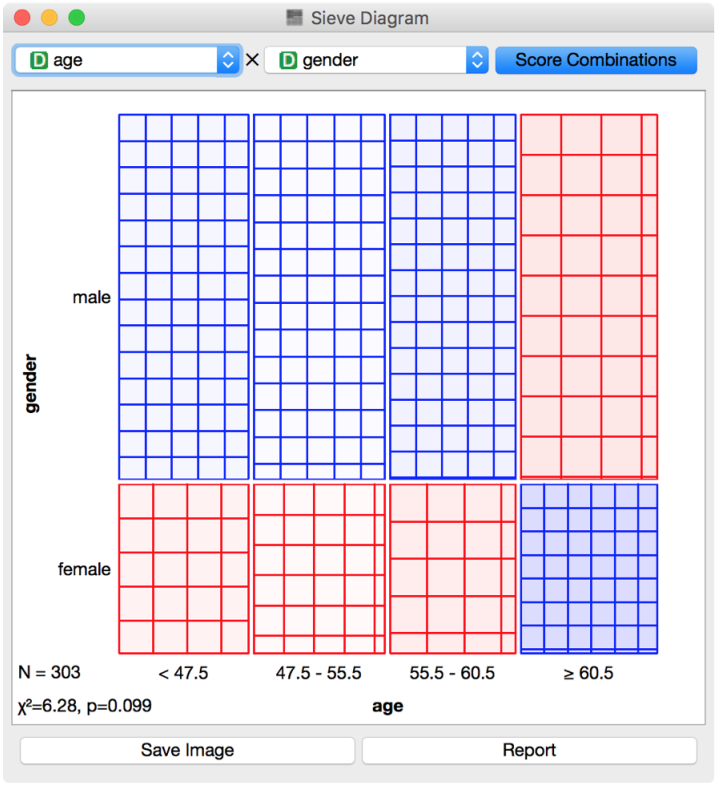
The mosaic display shows a rectangle split into columns with widths reflecting the prevalence of different types of chest pain. Each column is then further split vertically according to gender distributions within the column. The resulting rectangles are split again horizontally according to age group sizes. The red and blue areas represent each group's outcome distribution within the resulting bars, 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, the 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. Instead, there are 34. The sieve diagram shows the difference between the expected and the observed frequencies by the grid density and the color of the field.

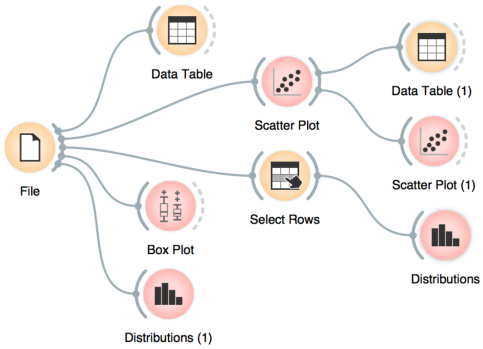


See the Score Combinations button? Try to guess what it does? And how does it score the combinations? Hint: there are some Greek letters at the bottom of the widget.

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.

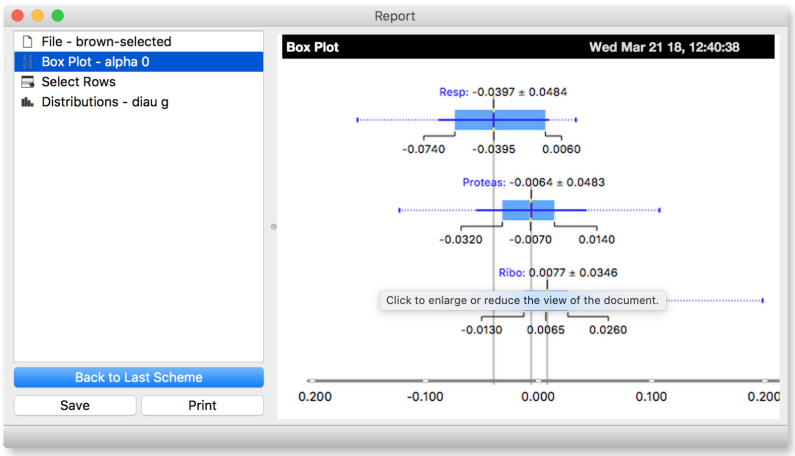


You can save this workflow using the File/Save menu and share it with your colleagues. Just don't forget to put the data files in the same directory as the file with the workflow.

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.

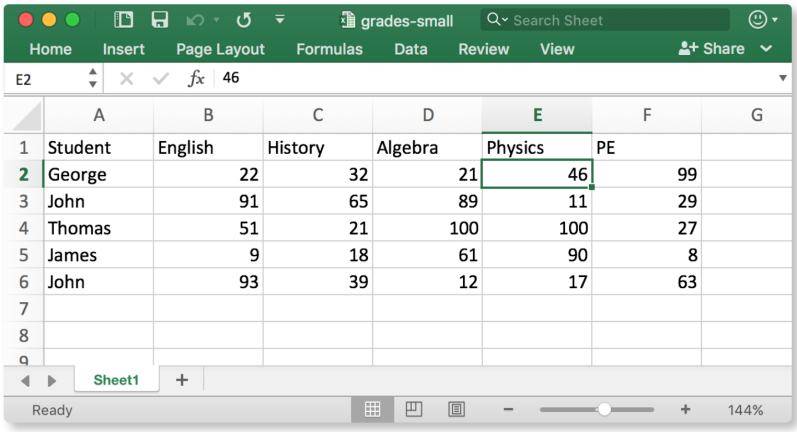
The report window and the additional text input box (bootom).



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

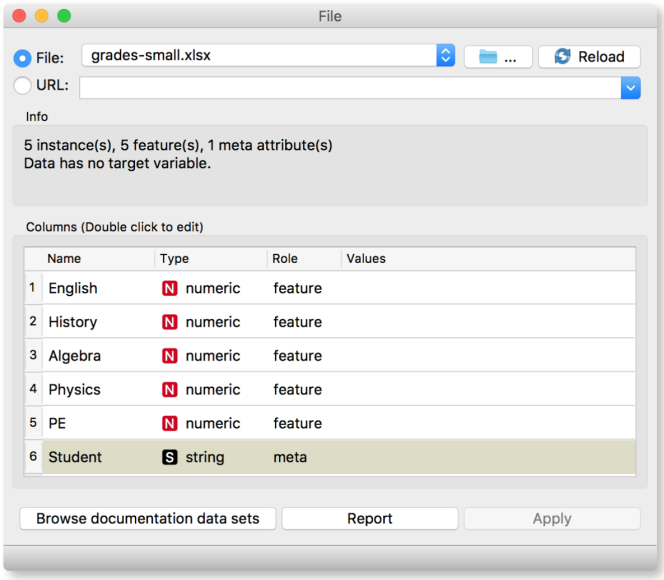
The data sets we have worked with in the previous lesson come with the Orange installation. Orange can read data from many file formats which include tab and comma separated and Excel files. To see how this works, let's prepare a data set (with school subjects and grades) in Excel and save it on a local disk.



	A	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							

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.

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



The File widget allows you to select a local file or even paste a URL to a Google Spreadsheet. In the Info box, you will see a quick summary about the data you loaded. By double clicking the fields, you can also edit the types of entries and their role, that will be relevant for further processing.

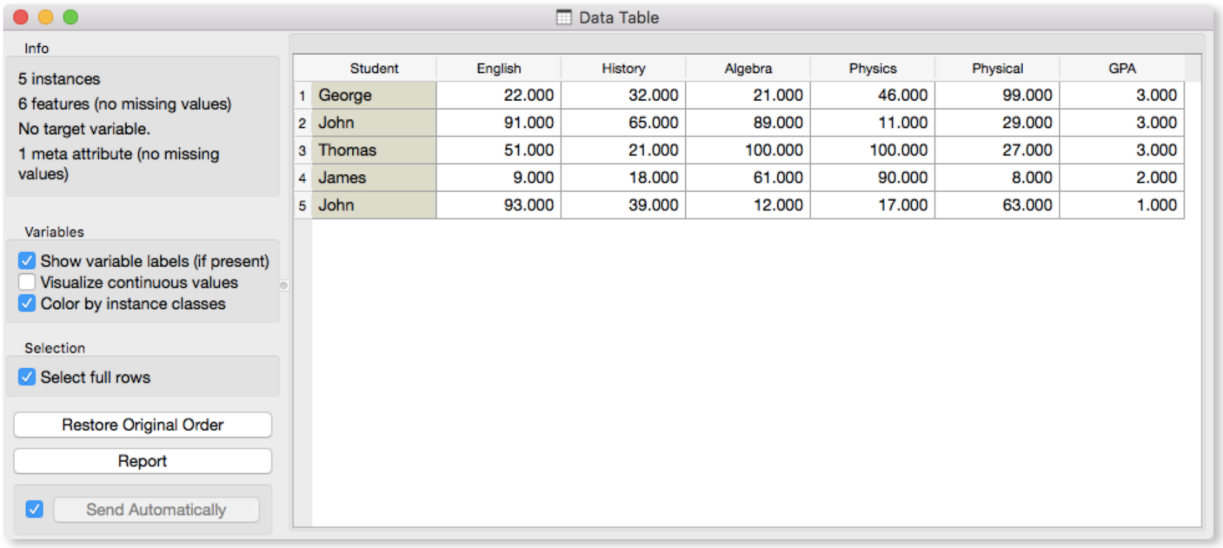
Looks good! Orange has correctly guessed that student names are character strings and that this column in the data set is special, meant

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,

Construct a 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.



	Student	English	History	Algebra	Physics	Physical	GPA
1	George	22.000	32.000	21.000	46.000	99.000	3.000
2	John	91.000	65.000	89.000	11.000	29.000	3.000
3	Thomas	51.000	21.000	100.000	100.000	27.000	3.000
4	James	9.000	18.000	61.000	90.000	8.000	2.000
5	John	93.000	39.000	12.000	17.000	63.000	1.000

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.

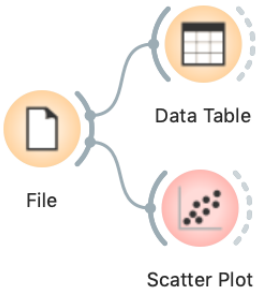
Orange's legacy native data format is a tab-delimited text file with three header rows. The first row lists the attribute names, the second row defines their type (continuous, discrete, time and string, or abbreviated c, d, t, and s), and the third row an optional role (class, meta, weight, or ignore).

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.

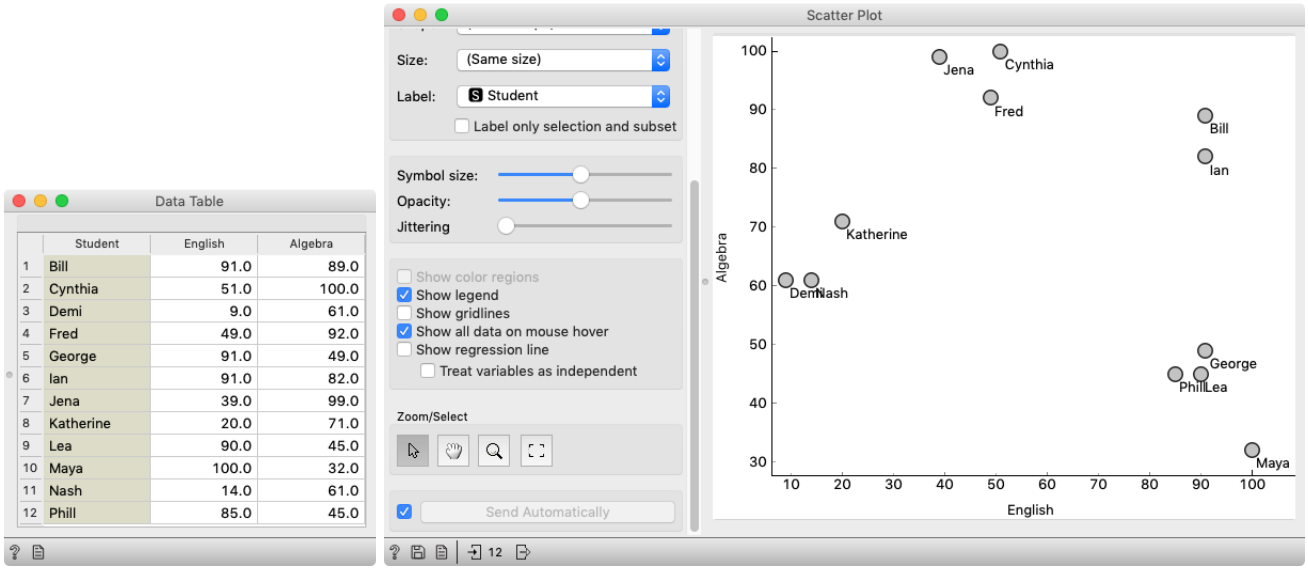
Hierarchical Clustering

We are interested in finding clusters in our data. We want to identify groups of data instances 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 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.

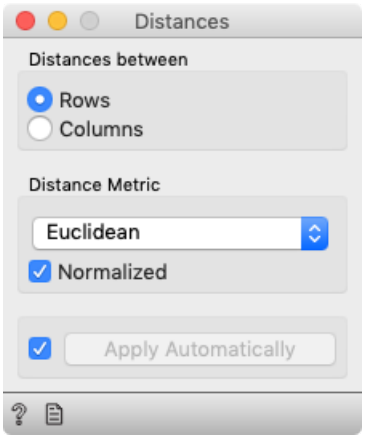


We will introduce clustering with a simple data set on students and their grades in English and Algebra. Load the data set from `http://file.biolab.si/text/grades.tab`.

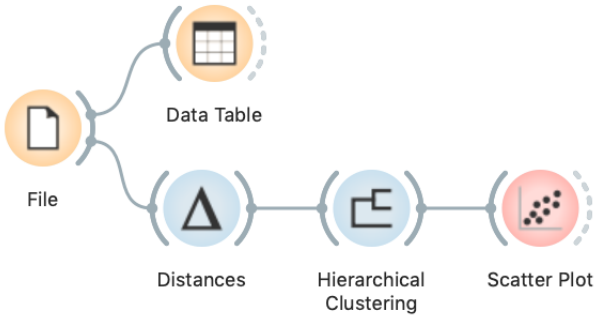


Next, we need to define a clustering algorithm. Say that we start with each data instance being its cluster, and then, at each step, we join the closest clusters. We estimate the distance between the clusters with the average distance between all their pairs of data points. This algorithm is called hierarchical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

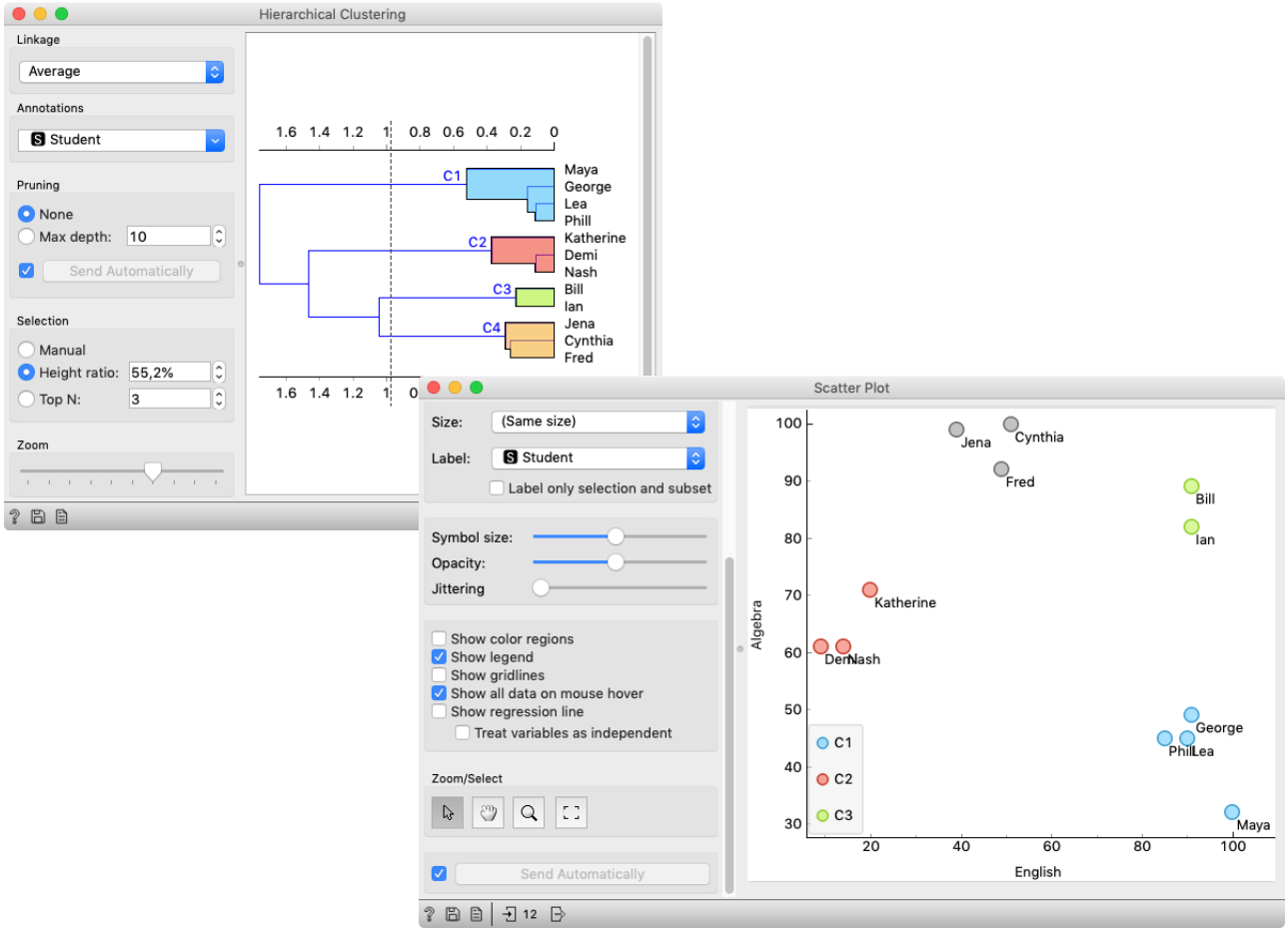
There are different ways to measure the similarity between clusters. The estimate we have described is called average linkage. We could also estimate the distance through the two closest points in each group (single linkage) or through the two points that are furthest away (complete linkage).



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



It 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 specific distance score and output the corresponding clusters:

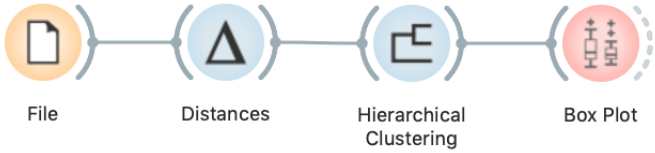


Animal Kingdom

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.

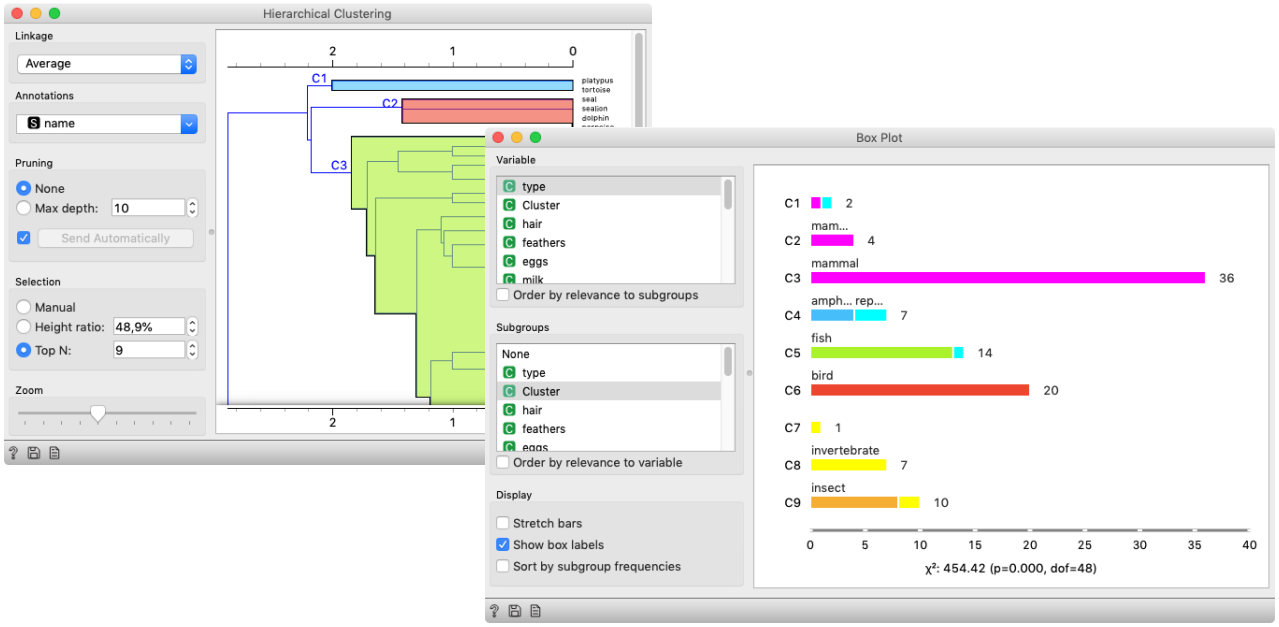
Funny stuff was we never understood the order in which the cards were laid out in the album. We later learned about taxonomy, but being more inclined to engineering we never mastered learning it in our biology classes. Luckily, there's data mining and the idea that taxonomy simply stems from measuring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



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

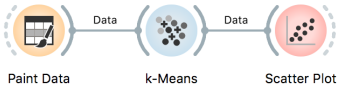
To split the data into clusters, let us manually set a threshold by dragging the vertical line left or right in the visualization. Can you say what is the appropriate number of groups?



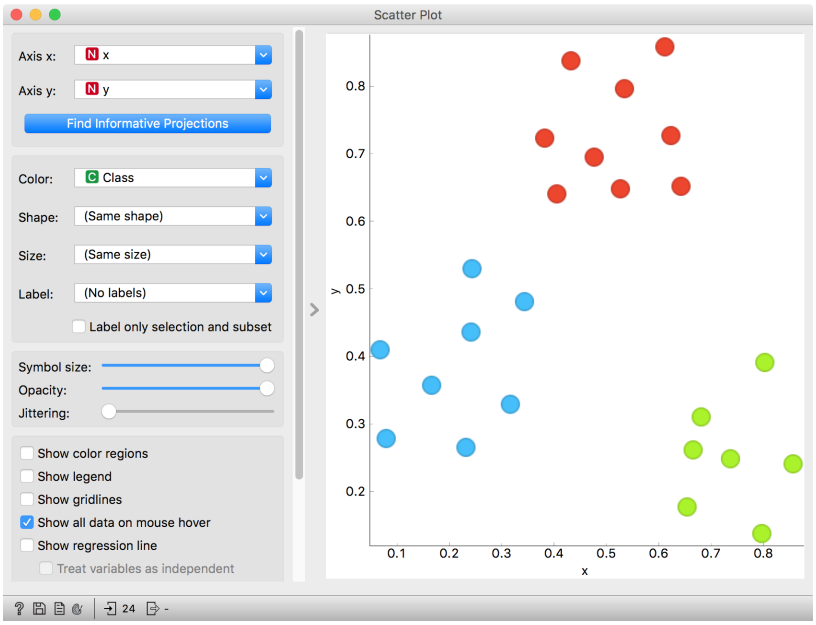
What is wrong with those mammals? Why can't they be in one single cluster? Two reasons. First, they represent 40% of the data instances. Second, they include some weirdos. Who are they?

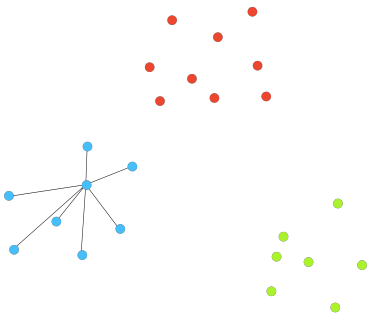
Silhouettes

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.



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.





Average distance A.

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.

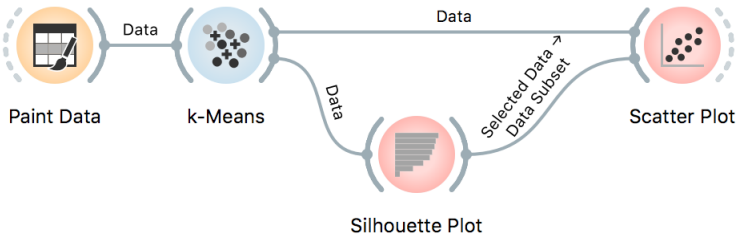
For a given data point (say the blue point in the image on the left), we can measure the distance to all the other points in its cluster and compute the average. Let us denote this average distance with A. The smaller the A, the better.

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.

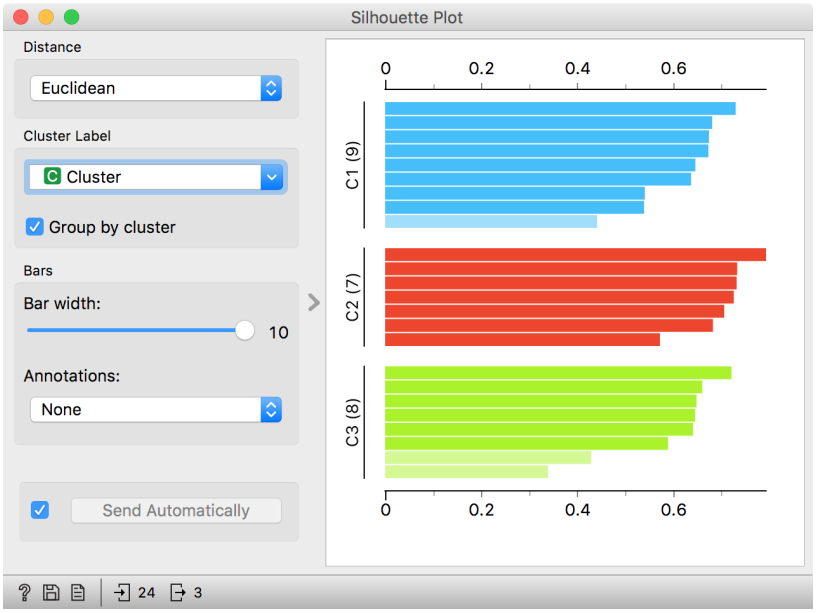
The larger the B, the better.

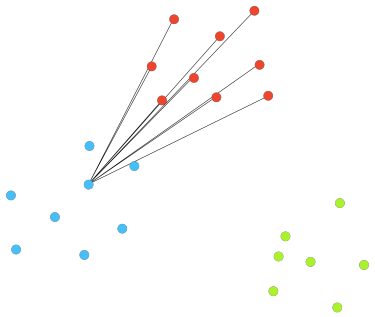
The point is well rooted within its own cluster if the distance to the points from the neighboring cluster (B) is much larger than the distance to the points from its own cluster (A), hence we compute B-A. We normalize it by dividing it with the larger of these two numbers, S = (B -A) / maxA, B. Voilá,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



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.

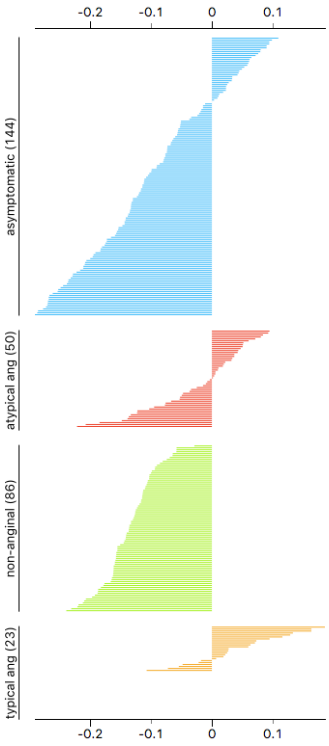




Average distance B.

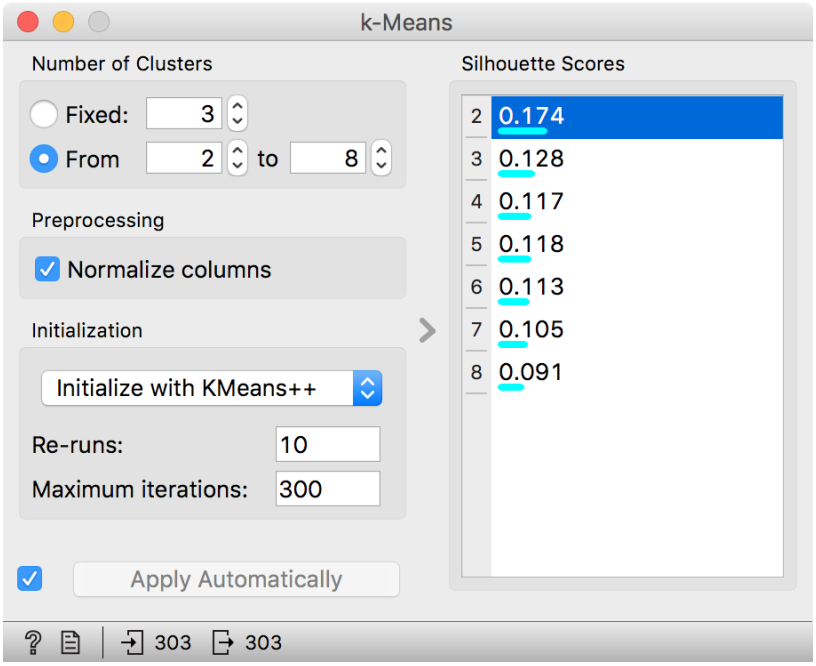
C3 is the green cluster, and all its points have large silhouettes. Not so for the other two.

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



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.



k-Means Clustering

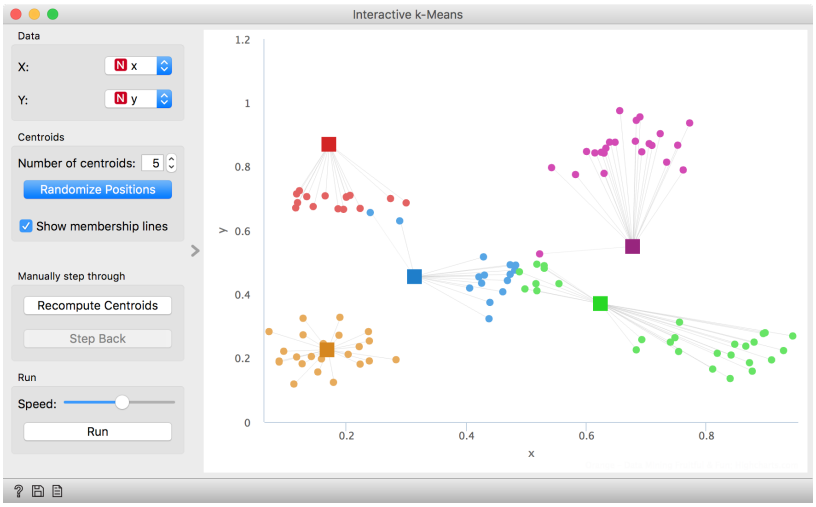
Hierarchical clustering is not suitable for larger data sets due to the prohibitive size of the distance matrix: with 30 thousand objects, the distance matrix already has almost one billion elements. An alternative approach that avoids using the distance matrix is k-means clustering.

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.

Try rerunning the clustering from new random positions and observe how the centers conquer the territory. Exciting, isn't it?



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.

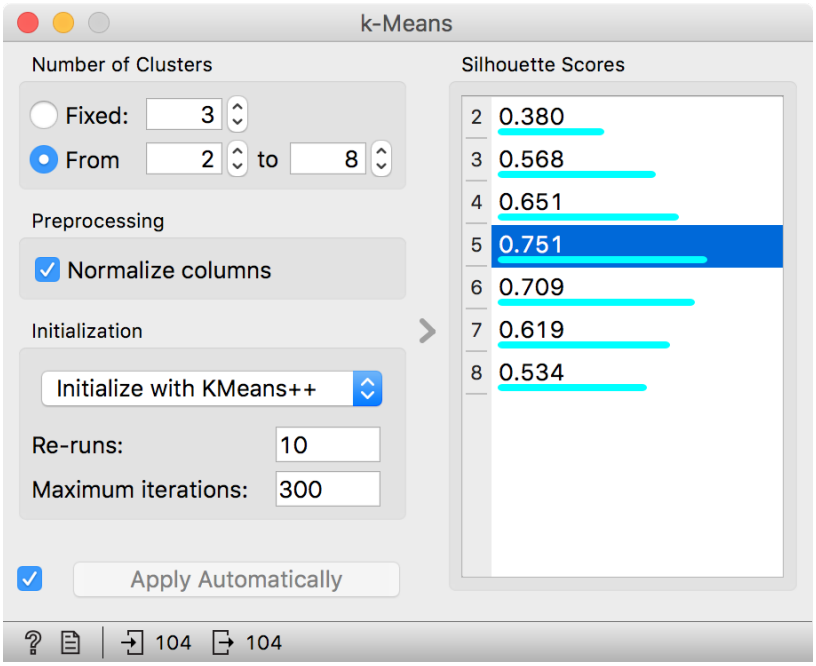
How do we set the initial number of clusters? That's simple: we choose the number that gives the optimal clustering.

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.

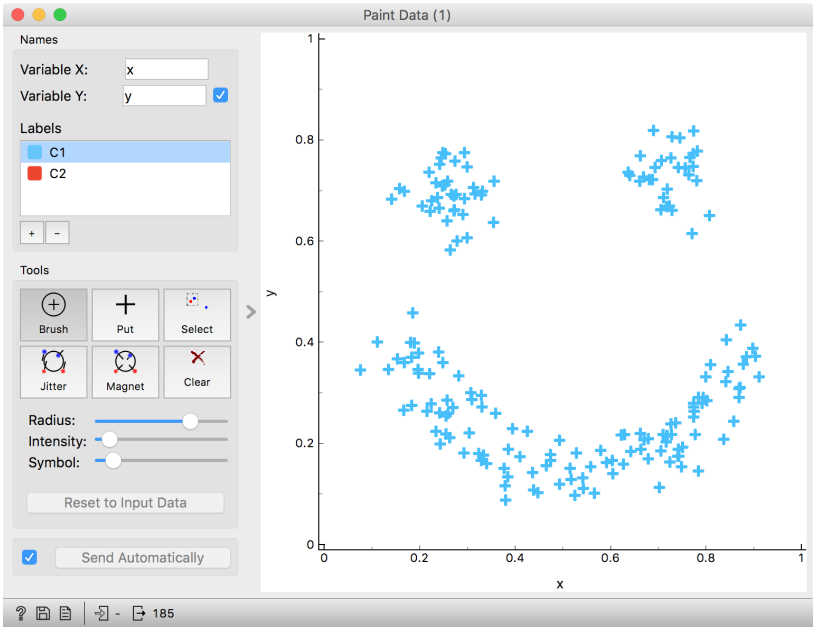
For this, we abandon our educational toy and connect Paint Data to the widget k-Means. We tell it to find the optimal number of clusters between 2 and 8, as scored by the Silhouette.



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?



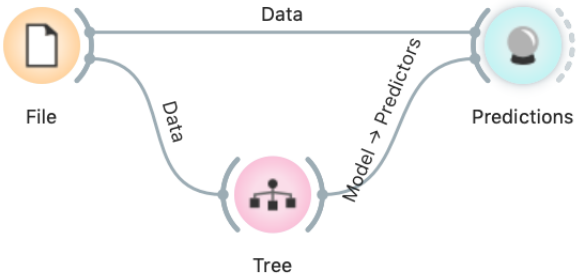
Classification

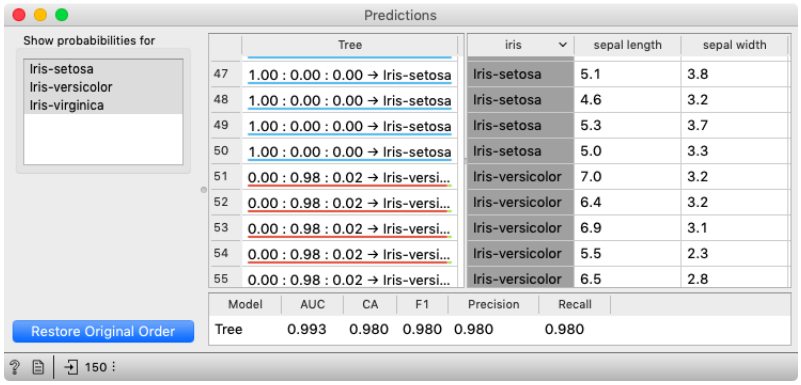
We call the variable we wish to predict a target variable, or an outcome or, in traditional machine learning terminology, a class. Hence we talk about classification, classifiers, classification trees...

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

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.





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?

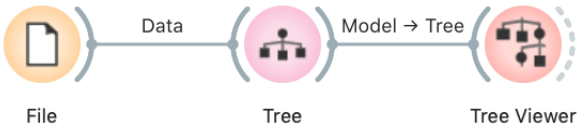
So many questions to answer!

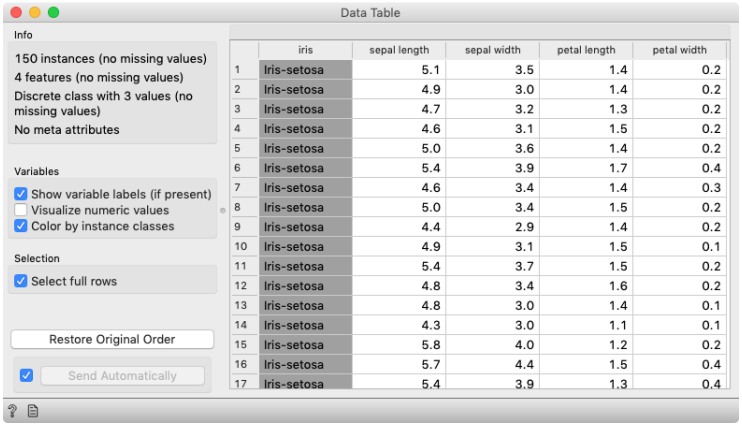
Classification Trees

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?

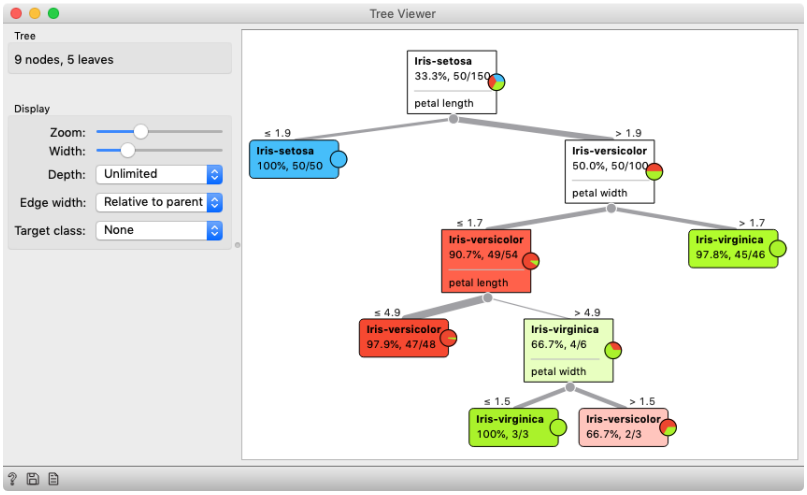
Classification trees were hugely popular in the early years of machine learning, when they were first independently proposed by the engineer Ross Quinlan (C4.5) and a group of statisticians (CART), including the father of random forests Leo Brieman.

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





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.



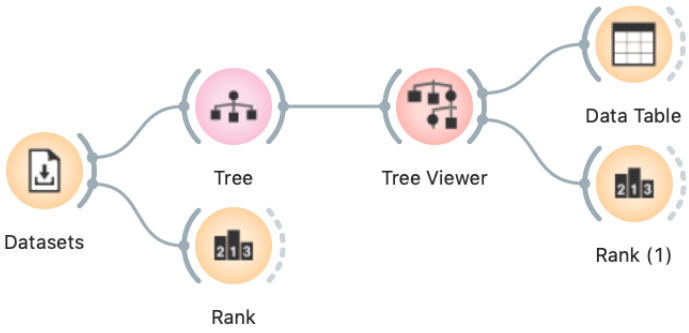
Trees place the most useful feature at the root. What would be the most useful feature? The feature that splits the data into two purest possible subsets. It then splits both subsets further, again by their most useful features, and keeps doing so until it reaches subsets in which all data belongs to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

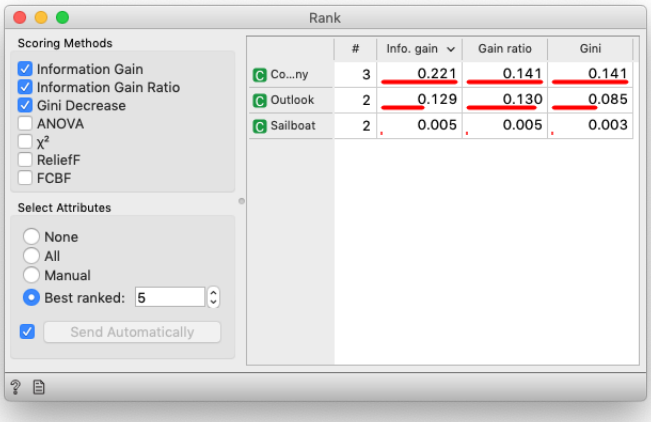
The Rank widget can be used on its own to show the best predicting features. Say, to figure out which genes are best predictors of the phenotype in some gene expression data set.

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

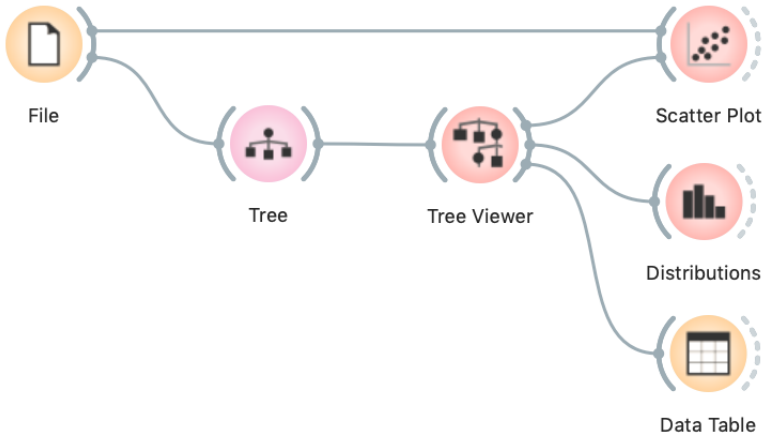


Besides the information gain, *Rank* displays several other measures (including Gain Ratio and Gini), which are often quite in agreement and were invented to better handle discrete features with many different values.

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

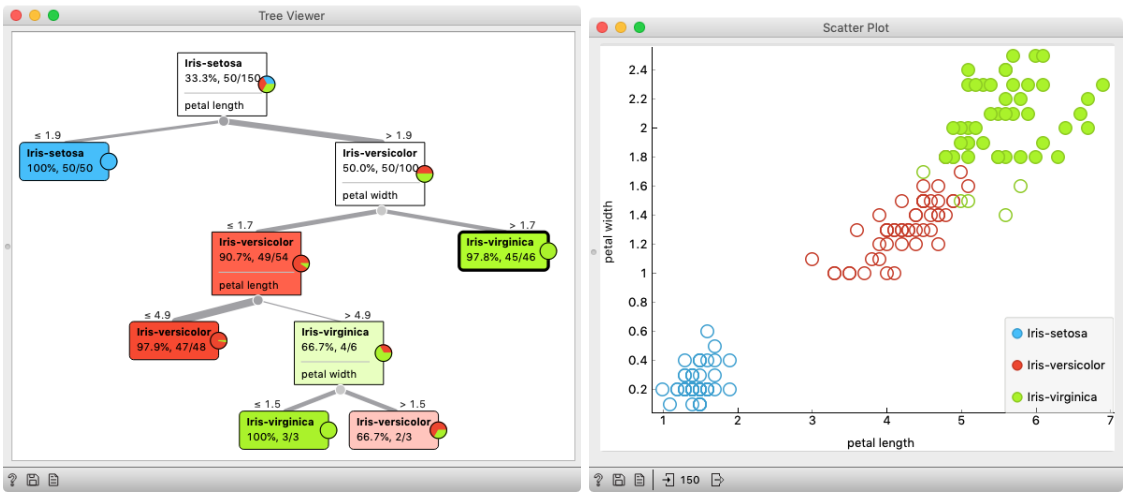


Here is an interesting combination of a *Tree Viewer* and a *Scatter Plot*. This time, use the *Iris* data set. In the *Scatter Plot*, we first find the best visualization of this data set, that is, the one that best separates the instances from different classes. Then we connect the *Tree Viewer* to 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.

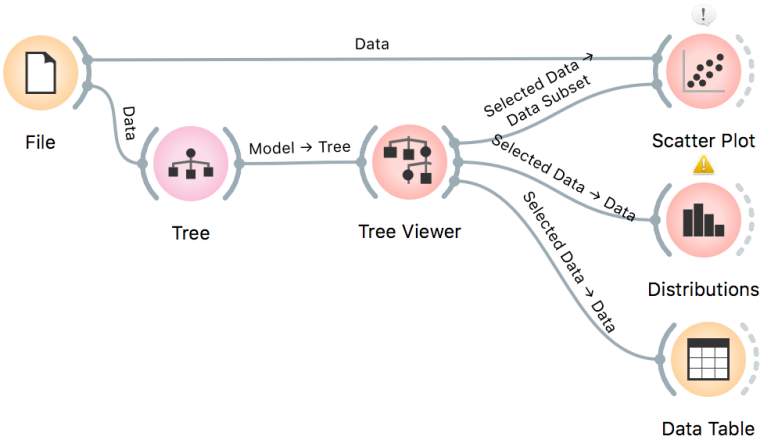


In the Tree Viewer we selected the rightmost node. All data instances coming to the selected node are highlighted in Scatter Plot.

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.

Model Inspection

Here's another interesting combination of widgets: *Tree Viewer* and *Scatter Plot*. In Scatter Plot, find the best visualization of this data set, that is, the one that best separates instances from different classes. Then connect Tree Viewer to Scatter Plot. Selecting any node of the tree will output the corresponding data subset, which will be shown in the scatter plot.



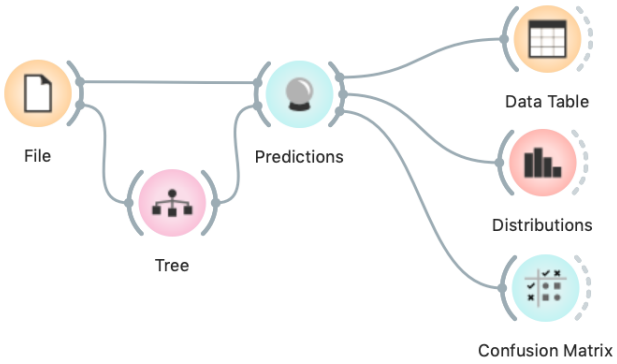
Just for fun, we have included a few other widgets in this workflow. The Tree Viewer selects data instances by inferring rules from the data itself and optimizing to obtain purer data subsets.

Classification Accuracy

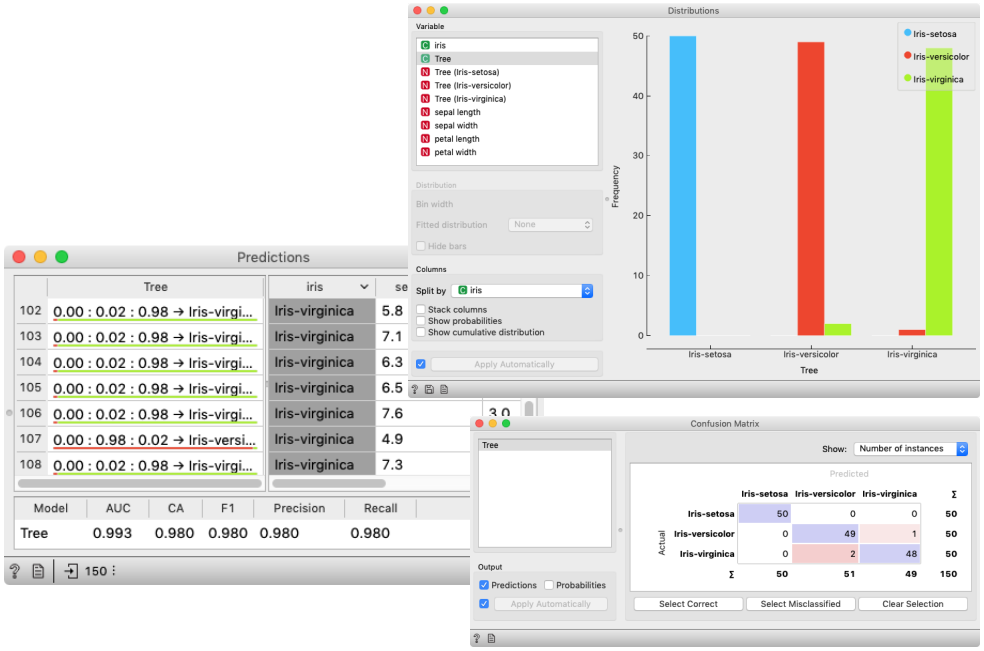
Now that we know what classification trees are, the next question is what is the quality of their predictions. For beginning, we need to define what we mean by quality. In classification, the simplest measure of quality is classification accuracy expressed as the proportion of data instances for which 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.

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

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 features in view appropriately.



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

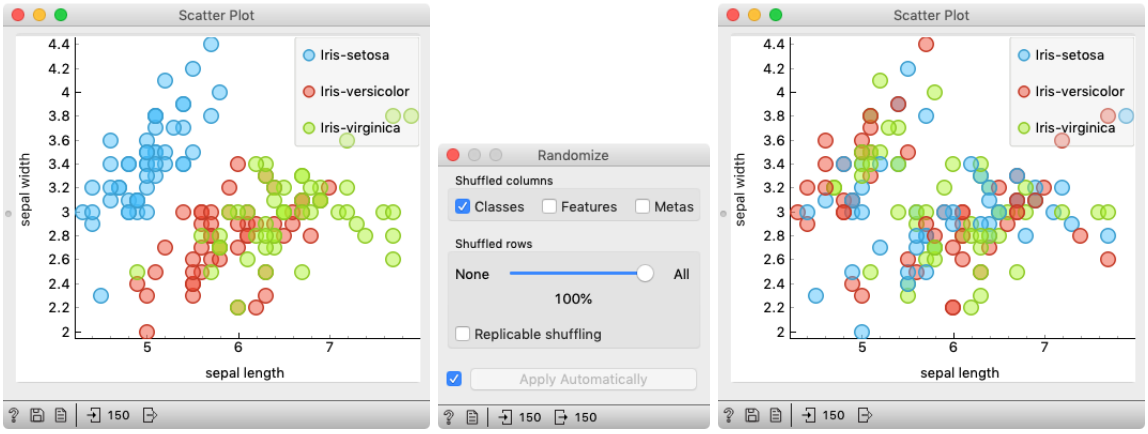


The Confusion Matrix shows 3 incorrectly classified examples, which makes the accuracy $(150 - 3)/150 = 98\%$.

How to Cheat

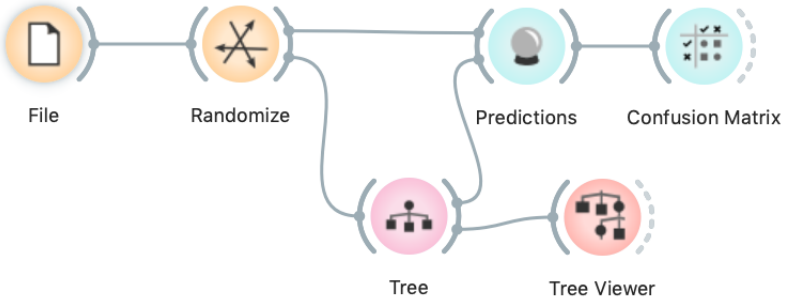
This lesson has a strange title and it is not obvious why it was chosen. Maybe you, the reader, should tell us what this lesson has to do with cheating.

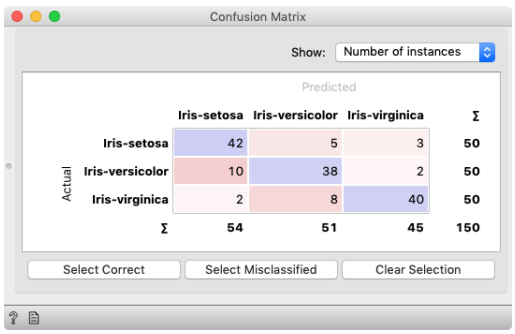
At this stage, the classification tree looks very good. There's only one data point where it makes a mistake. Can we mess up the data set so bad that the trees will ultimately fail? Like, remove any existing correlation between features and the class? We can! There's the *Randomize* widget with class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

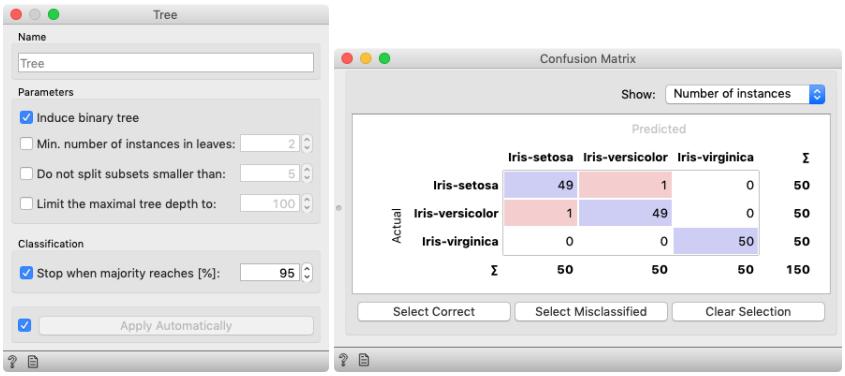




And the result? Here is a screenshot of the *Confusion Matrix*.

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.



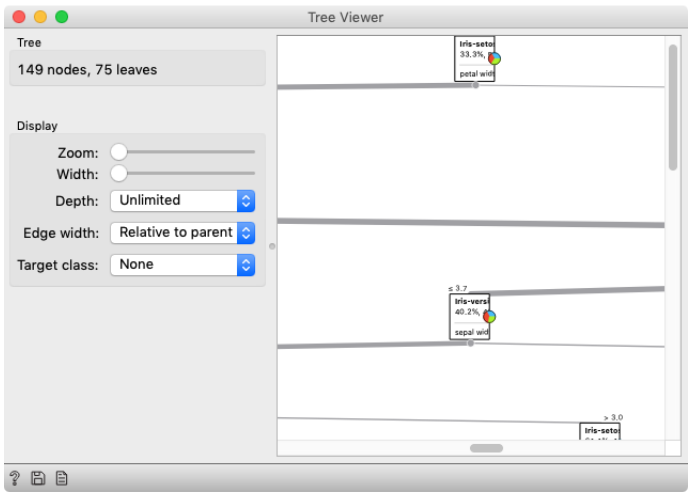
After we disable 2–4 check box in the Tree widget, our classifier starts behaving almost perfectly.

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

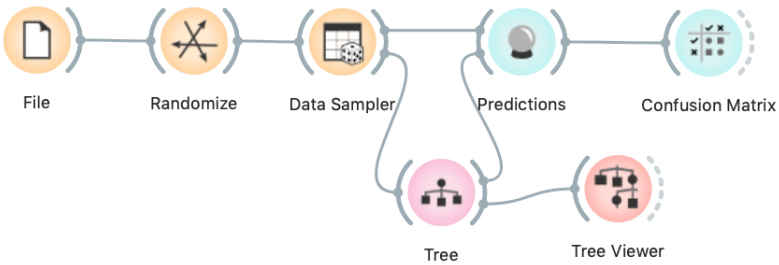
To find the answer to this riddle, open the *Tree Viewer* and check out the tree. How many nodes does it have? Are there many data instances in the leaf nodes?

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.



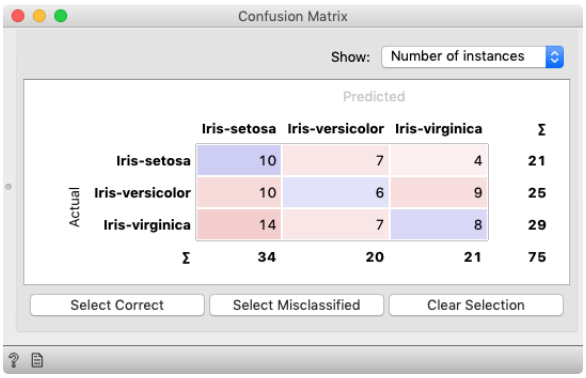
Connect the Data Sampler widget carefully. The Data Sampler splits the data to a sample and out-of-sample (so called remaining data). The sample was given to the Tree widget, while the remaining data was handed to the Predictions widget. Set the Data Sampler so that the size of these two data sets is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

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

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.

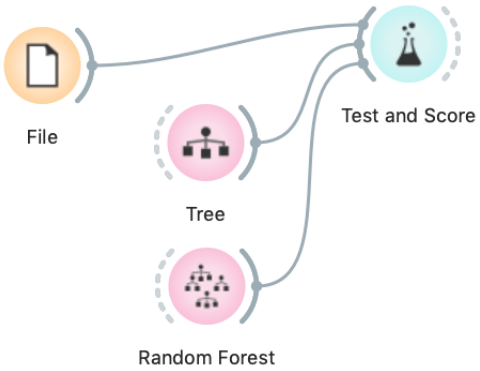


We have just learned that we need to train the classifiers on the training set and then test it on a separate test set to really measure performance of a classification technique. With this test, we can distinguish between those classifiers that just memorize the training data and those that actually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

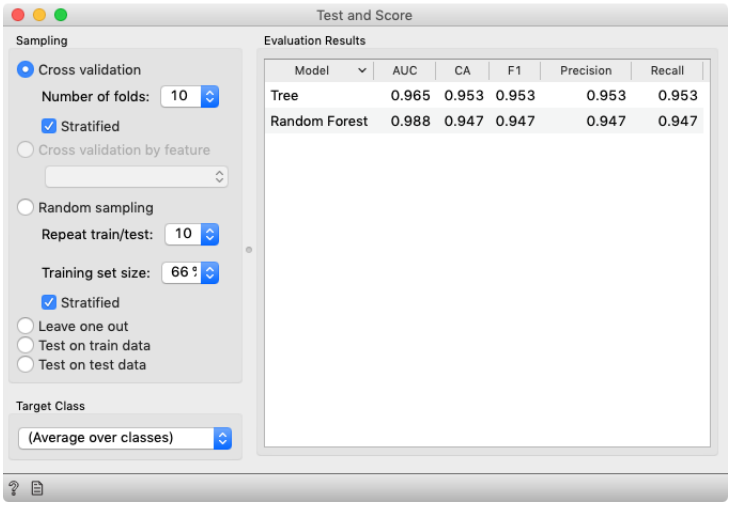
Estimating the accuracy may depend on a particular split of the data set. To increase robustness, we can repeat the measurement several times, each time choosing a different subset of the data for training. One such method is cross-validation. It is available in Orange in the *Test and Score* widget.



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.

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

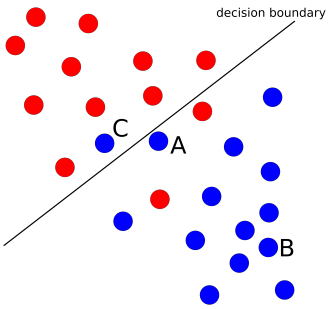
This is another way to use the *Tree* widget. In the workflows from the previous lessons we have used another of its outputs, called *Model*; its construction required data. This time, no data is needed for *Tree*, because all that we need from it is a *Learner*.



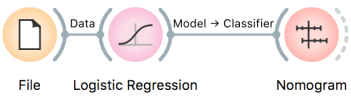
Cross validation splits the data sets into, say, 10 different non-overlapping subsets we call folds. In each iteration, one fold will be used for testing, while the data from all other folds will be used for training. In this way, each data instance will be used for testing exactly once.

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

Logistic Regression



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

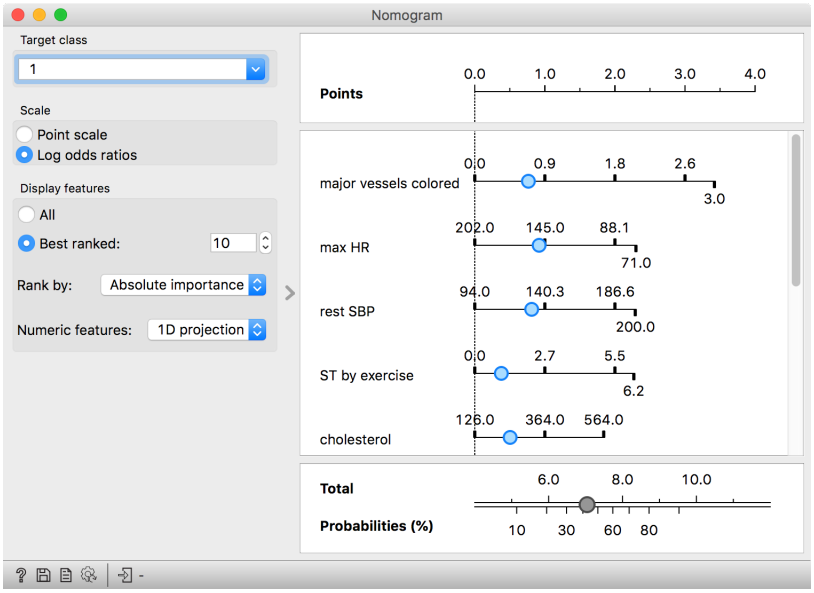


Logistic regression is one of the best-known classifiers. The model returns the probability of a class variable, based on input features. First, it computes probabilities with a one-versus-all approach, meaning that for a multiclass problem, it will take one target value and treat all the rest as "other",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

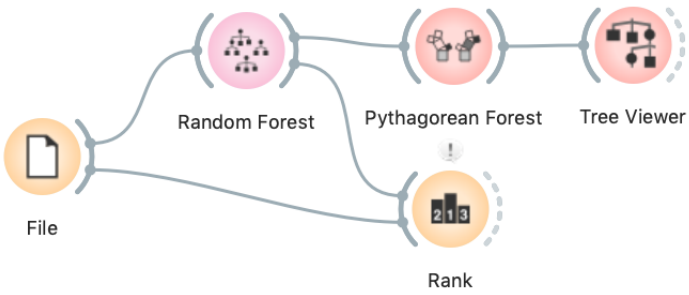
A great thing about *Logistic Regression* is that we can interpret it with a *Nomogram*. Nomogram shows the importance of variables for the model. The higher the variable is in the list, the greater its importance. Also, the longer the line, the greater the importance. The line corresponds to the coefficient 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.



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.

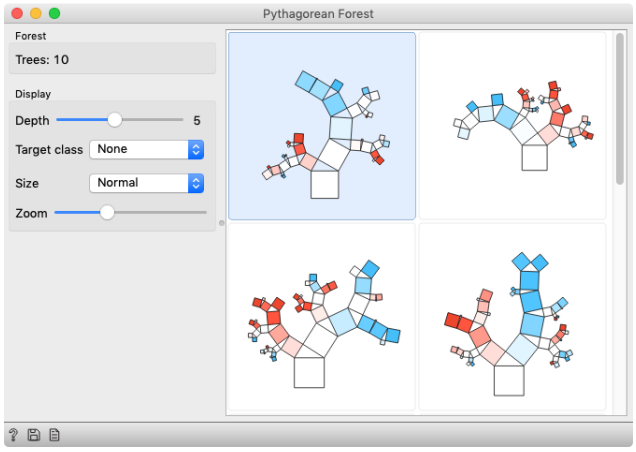
A not so great thing about logistic regression is that it operates with planes, meaning that the model won't work when the data cannot be separated in such a way. Can you think of such a data set?

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.

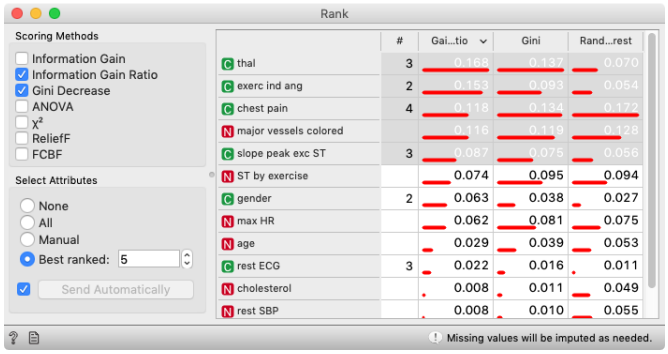
The *Pythagorean Forest* widget shows us how random the trees are. If we select a tree, we can observe it in a *Tree Viewer*.



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.

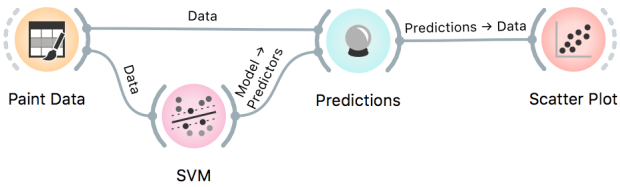
Feature importance according to two univariate measures (gain ratio and gini index) and random forests. Random forests also consider combinations of features when evaluating their importance.



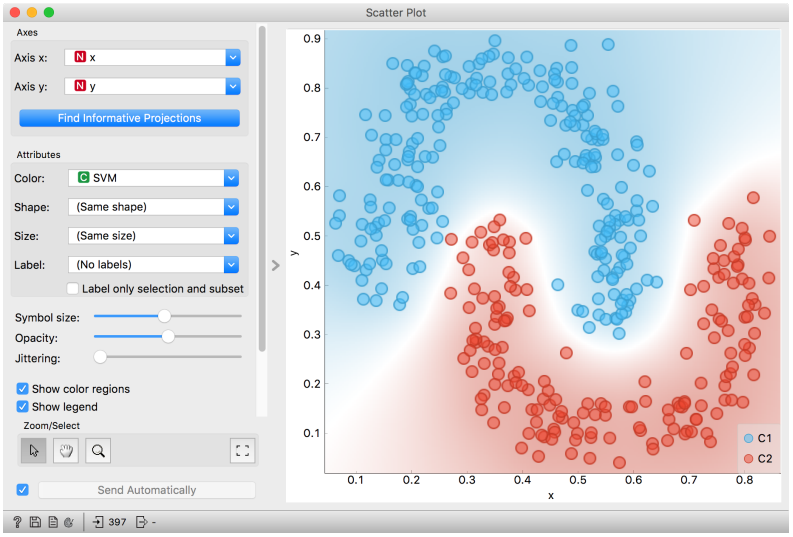
Support Vector Machines

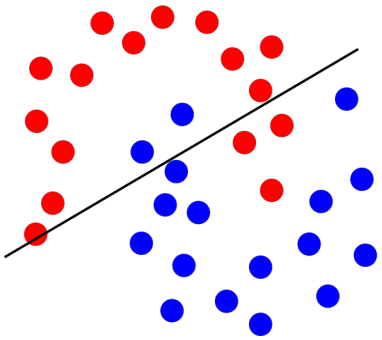
Support vector machines (SVM) are another example of linear classifiers, similar to logistic or linear regression. However, SVM can overcome splitting the data by a plane by using the so-called *kernel trick*. This means the hyperplane (decision boundary) can be transformed to a higher-dimensional space, 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.

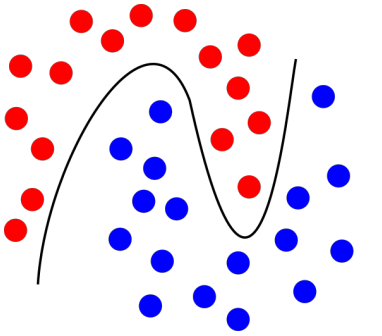


Abstract talking aside, SVM with different kernels can split the data not by ordinary hyperplanes, but with more complex curves. The complexity of the curve is decided by the kernel type and by the arguments given to the algorithm, like the degree and coefficients, and the penalty for misclassifications.



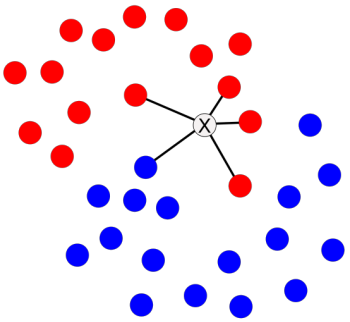


Decision boundary of a linear regression classifier.



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

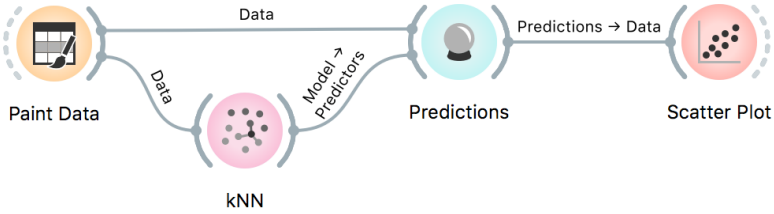
k-Nearest Neighbors



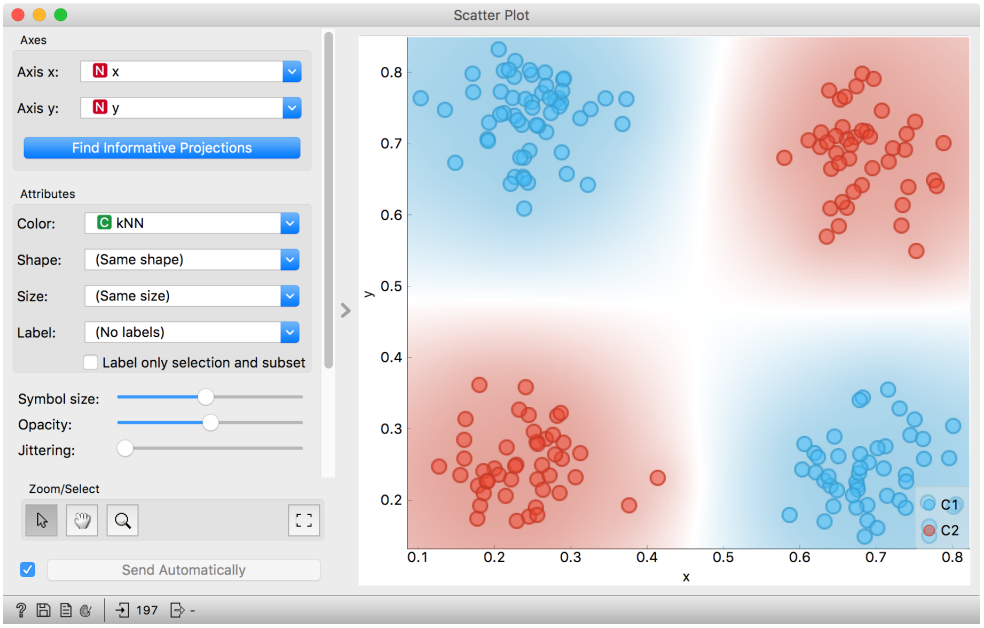
kNN classifier looks at k nearest neighbors, say 5, of instance X. 4 neighbors belong to the red class and 1 to the blue class. X will thus be classified as red with 80% probability.

The 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*.



The advantage of kNN algorithm is that it can successfully model the data, where classes are not linearly separably. It can also be retrained 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.



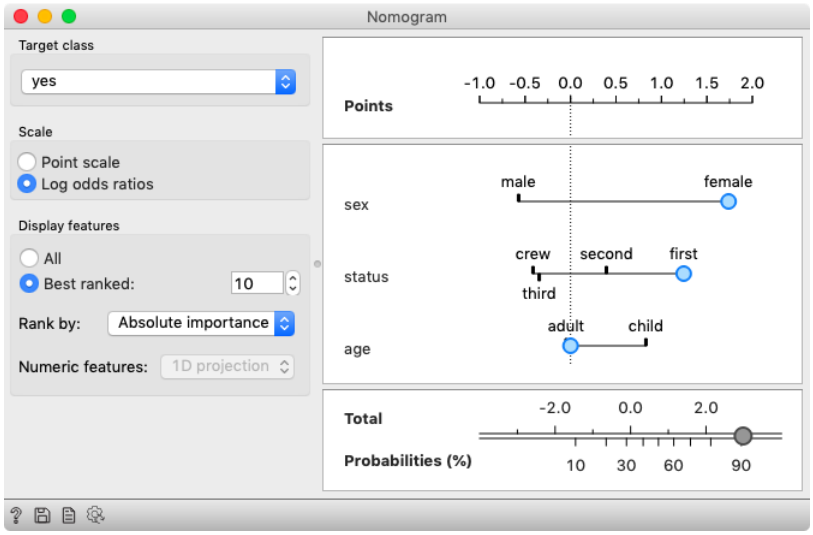
Naive Bayes

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.

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.

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.

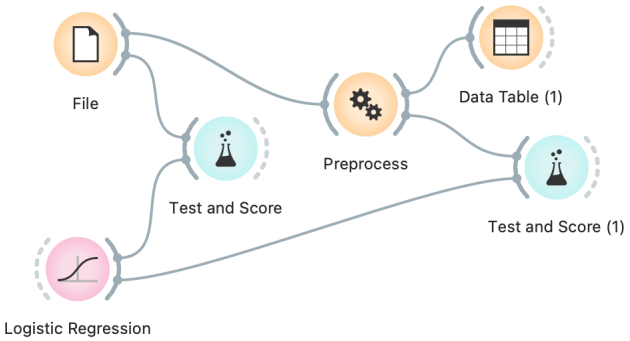


According to the probability theory individual contributions should be multiplied. Nomograms get around this by working in a log-space: a sum in the log-space is equivalent to multiplication in the original space. Therefore nomograms sum contributions (in the log-space) of all feature values and then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

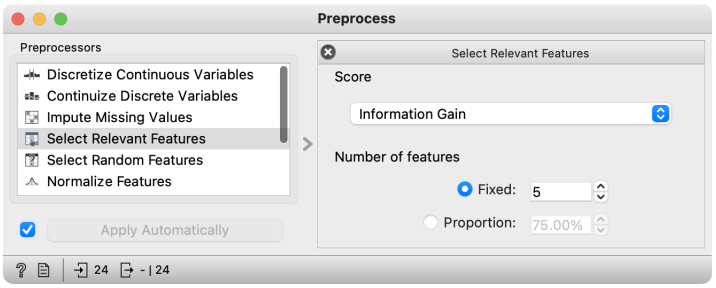
Cheating with Feature Subset Selection

We will borrow a gene expression data set from Gene Expression Omnibus for our example. There is a particular widget in the Orange bioinformatics add-on that we could use to fetch this and similar data sets. Instead, we will rely on the GEO data set gds360 available at `http://file.biolab.si/datasets/gds360.pkl`.

Consider a typical gene expression data set with samples in rows and genes expressions in columns. These data sets are usually fat; they include more genes than samples. Fat data sets are almost typical for systems biology. When we label the samples with phenotype, and our task is phenotype classification, many features (genes) will be irrelevant. Most often, only a few features correlate with class. So why not simply select a set of most informative features first and then do the whole analysis? At least cross-validation will then work much faster, as the model inference algorithms will deal with much smaller data tables. Cool. What a nice trick! Let's try it out in the following workflow.



The workflow above uses the data preprocessing widget, which we have configured to select the five most informative features.



Observe the classification accuracy obtained on the original data set and the data set with the five best-selected features. What is happening? What is there such difference in classification accuracy?

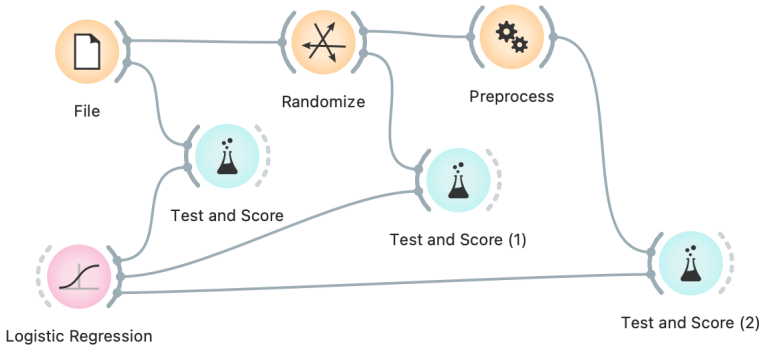
Cheating Even Works on Randomized Data

We can push the example from our previous lesson to the extreme. We will randomize the classification data. We will take the column with the class values and randomly permute it. We will use the Randomize widget to do this.

Later, we will classify this data set. We expect low classification accuracy on randomized data set. Then, we will select five features that are most associated with the class. Even though we randomly permuted the classes, there have to be some features that are weakly correlated with the class. Simply because we have tens of thousands of features, and we have only a few samples. There are enough features to associate with class simply by chance. Finally, we will score a random forest on a randomized data set with selected features.

Compare the scores reported by cross-validation on different data sets in this pipeline. Why is the accuracy in the final one relatively high? Would adding more "most informative features" improve or degrade the cross-validated performance on a randomized data set?

Instead of selecting the five most informative features, you can further reduce this number. Say, to two most informative features. What happens? Why does accuracy rise after this change?

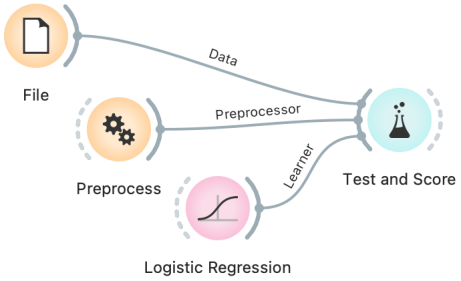


How to Correctly Perform Test and Score?

To put it simply: never, in any way, transform the data before cross-validation. Any transformation should happen within the cross-validation loop, first on the training set and, if required, on a test set. In a simple form: it's ok to transform the data, but we should change the data independently on the train, and the test set and the transformation on the test set should not use the information about the class value. Data imputation could be an example of such an operation, but it should be carried out separately for the train and test set and not consider classes.

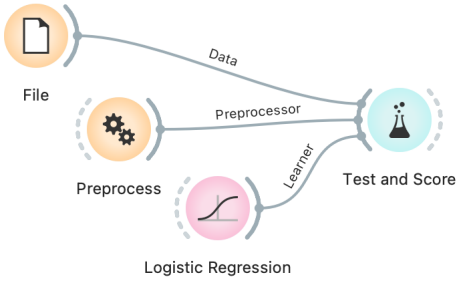
So how do we then correctly preprocess data in Orange? The idea of reducing the number of features before inferring a predictive model may still be appealing, now that we know we can use it on training data sets (leaving the test set alone). Following are two workflows that do this correctly.

The writing on the right looks straightforward. But actually one needs to be extremely careful not to succumb to overfitting when reporting results of cross-validation tests. The literature on systems biology is polluted with reporting on overly optimistic results, and high impact factors provide no guarantee that studies were carried out correctly (in fact, due to a lack of reviewers from the field of machine learning, mistakes likely stay overlooked).



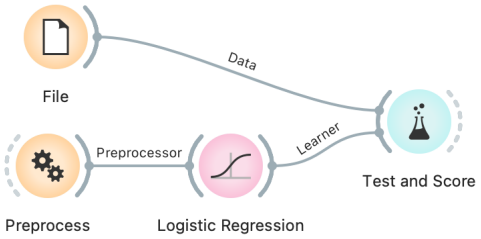
Simon et al. (2003) provides a great read on this topic. He found that many of the early papers in gene expression analysis reported high accuracy simply due to overfitting.

In this first workflow, we gave the *Test and Score* widget a preprocessor – we used feature selection in this example. The *Test and Score* uses it correctly only on the training sets. This type of workflow is preferred if we would like to test the effect of preprocessing on several different learning algorithms.



Alternatively, we can include a preprocessor in a learning method. The workflow now calls the preprocessor on the training data set just

before this learner performs inference of the predictive model.



The Preprocess widget does not necessary require a data set on its input. An alternative use of this widget is to output a method for data preprocessing, which we can then pass to either a learning method or to a widget for cross validation.

Can you extend this workflow to such an extent that you can test both a learner with preprocessing by feature subset selection and the same learner without this preprocessing? How does the number of selected features affect the cross-validated accuracies? Does the success of this particular combination of machine learning techniques depend on the input data set? Does it work better for some machine learning algorithms? Try its performance on k-nearest neighbors learner (warning: use small data sets, this classifier could be very slow).

This is not the first time we have used a widget that instead of a data passes forward a computation method. All the learners, like Random Forest, do so. A learner could get data on its input and pass a classifier to its output, or simple pass an instance of itself, that is, pass a learning algorithm to whichever widget could use it. For instance, to the Test and Score widget.

Somehow, in a shy way, we have also introduced a technique for feature selection and pointed to its possible utility for classification problems. Feature subset selection, or FSS in short, was and still is, to some extent, an essential topic in machine learning. Modern classification algorithms, though, perform it implicitly and can deal with many features without the help of external procedures for their advanced selection. Random forest is one such technique.

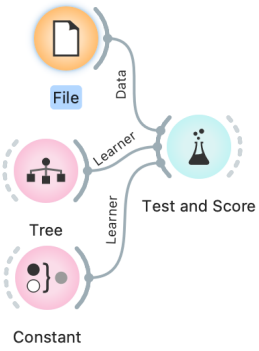
Classification Model Scoring

In multiple-choice exams, they grade you according to the number of correct answers. The same goes for classifiers: the more correct predictions they make, the better they are. Nothing could make more sense. Right?

Maybe not. Consider Dr. Smith. He is a specialist for specific diseases, and his diagnosis is correct in 98% of the cases. Would you consider visiting him if you have some symptoms related to his specialty?

Not necessarily. Dr. Smith's specialty is rare diseases. Two out of a hundred of his patients have it, and, being lazy, he always dismisses everybody as healthy. His predictions are worthless — although highly accurate. Classification accuracy is not an absolute measure, which we can judge out of context. At the very least, we have to compare it with the frequency of the majority class, which is, in the case of rare diseases, quite, hm, substantial.

For instance, on the GEO data set GDS 4182, the classification tree achieves 79% cross-validation accuracy, which may be reasonably good. Let us compare this with the Constant model, which implements Dr. Smith's strategy by always predicting the majority. It gets 83%. Classification trees are not so good after all, are they?

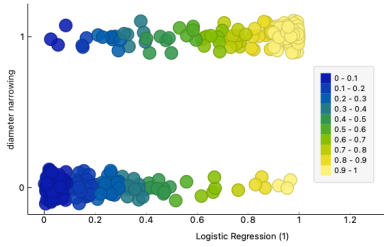


What do other columns in Test and Score widget represent? Keep reading!

Model	AUC	CA	F1	Precision	Recall
Tree	0.711	0.792	0.792	0.792	0.792
Constant	0.425	0.833	0.758	0.694	0.833

On the other hand, the accuracy of classification trees on GDS 3713 is about 71%, which seems rather good compared to the 50% achieved by predicting the majority.

Model	AUC	CA	F1	Precision	Recall
Tree	0.701	0.709	0.709	0.709	0.709
Constant	0.488	0.506	0.340	0.256	0.506



Classes versus probabilities as estimated by logistic regression. Can you replicate this image?

The problem with classification accuracy goes deeper, though. Classifiers usually make predictions based on the probabilities they compute. If a data instance belongs to class *healthy* with a probability of 80% and *disseased* with a probability of 20%, we classified it as *healthy*. Such classification makes sense, right?

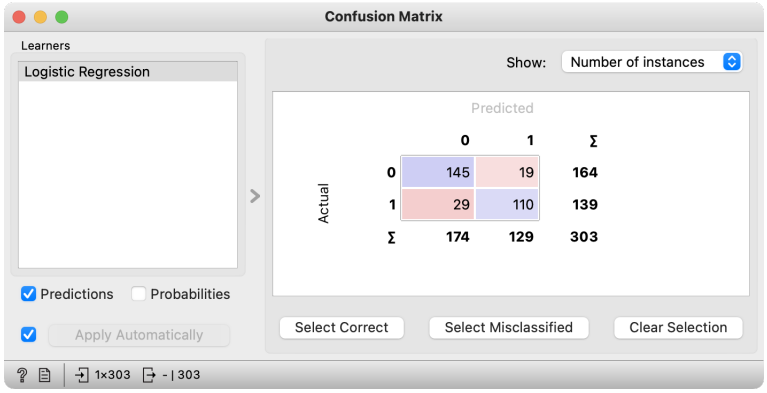
Maybe not, again. Say you fall down the stairs, and your leg hurts. You open Orange, enter some data into your favorite model and compute a 20% of having your leg broken. So you assume your leg is all right, and you take an aspirin. Or perhaps not?

What if the chance of a broken leg was 10%? 5%? 0.1%? Say we

decide that any leg with a 1% chance of being broken will be classified as broken. What will this do to our classification threshold? It is going to decrease badly — but we do not care. What do we do care about then? What kind of "accuracy" is essential?

Not all mistakes are equal. We can summarize them in the Confusion Matrix. Here is one for logistic regression on the heart disease data.

The numbers in the Confusion Matrix have names. We can classify a data instance as positive or negative; imagine this as being positive or negative when tested for some medical condition. This classification can be true or false. So there are four options, true positive (TP), false positive (FP), true negative (TN), and false negative (FN). Identify them in the table!



Logistic regression correctly classifies 145 healthy persons and 110 of the sick, the numbers on the diagonal. Classification accuracy is then 255 out of 303, which is about 84%.

Use the output from Confusion Matrix as a subset for Scatter plot to explore the data instances that were misclassified in a certain way.

Nineteen healthy people were unnecessarily scared. The opposite error is worse: the heart problems of 29 persons went undetected. We need to distinguish between these two kinds of mistakes.

We are interested in the probability that a person who has some problem will be correctly diagnosed. There were 139 such cases, and we correctly discovered 110. The proportion is $110/139 = 0.79$. This measure is called *sensitivity* or *recall* or *true positive rate* (TPR).

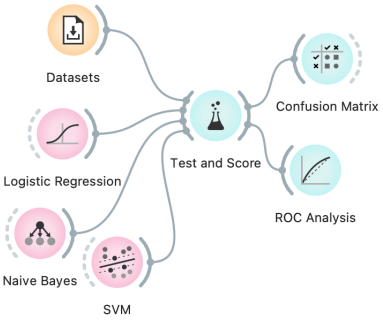
If you were interested only in sensitivity, though, here's Dr. Smith's associate partner — wanting to be on the safe side, she considers everybody ill, so she has a perfect sensitivity of 1.0.

To counterbalance the sensitivity, we compute the opposite: what is the proportion of correctly classified *negative* instances? 145 out of 164, that is, about 90%. This score is called *specificity* or *true negative* rate.

If you are interested in a complete list of statistics, see the Wikipedia page on Receiver operating characteristic, https://en.wikipedia.org/wiki/Receiver_operating_characteristic.

So, if the model classifies you as healthy, do you have a 90% chance of actually being OK? No, it's the other way around: 90% is the chance of being classified as OK if you are OK. (Think about it, it's not as complicated as it sounds). If you're interested in your chance of being OK if the classifier tells you so, you look for the *negative predictive value*. Then there's also *precision*, the probability of being positive if you are classified as such. And the *fall-out* and *negative likelihood ratio*. There is a list of other indistinguishable fancy names for various statistics, each useful for some purpose.

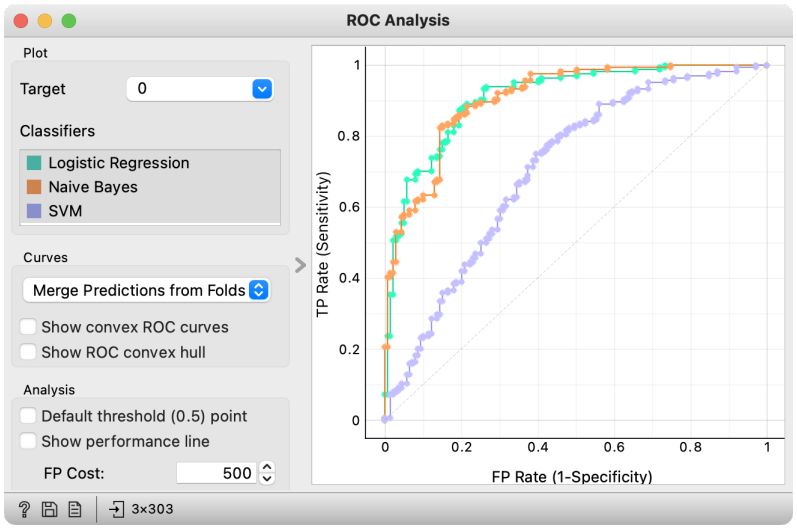
Choosing the Decision Threshold



The common property of scores from the previous lesson is that they depend on the threshold for classifying an instance as positive. We can balance between them by adjusting the threshold to find the required sensitivity at an acceptable specificity. We can even assign costs (monetary or not) to different kinds of mistakes and find the threshold with the minimal expected cost.

A valuable tool to investigate the effects of different thresholds is the Receiver-Operating Characteristic curve. Don't mind the (historical) meaning of the name and just call it the ROC curve.

Here are the curves for logistic regression, SVM with linear kernels, and naive Bayesian classifier on the same ROC plot.



The curves show how the sensitivity (y-axis) and specificity (x-axis, but from right to left) change with different thresholds.

There exists, for instance, a threshold for logistic regression (the green curve) that gives us 0.65 sensitivity at the specificity of 0.95 (the curve shows 1 - specificity). Or 0.9 sensitivity with a specificity of 0.8. Or a sensitivity of (almost) 1 with a specificity of somewhere around 0.3.

The optimal point would be at the top left. The diagonal represents the behavior of a random guessing classifier.

Which of the three classifiers is the best? It depends on the specificity and sensitivity we want; at some points, we prefer logistic regression and, at other points, the naive bayesian classifier. SVM doesn't cut it anywhere.

A popular score derived from the ROC curve is called an *area under curve*, or AUC for short. It measures, well, the area under the curve. ROC curve. If the curve goes straight up and then right, the area is

Sounds complicated? If it helps: perhaps you remember the term parametric curve from some of your math classes. ROC is a parametric curve where x and y (the sensitivity and 1 - specificity) are a function of the same parameter, the decision threshold.

1; we can not reach optimal AUC in practice. If the classifier guesses at random, the curve follows the diagonal, and AUC is 0.5. Anything below that is equivalent to guessing or bad luck.

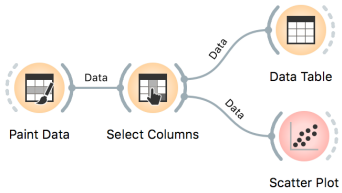
AUC has a kind of absolute scale. As a rule of thumb: 0.6 is bad, 0.7 is bearable, 0.8 is publishable, and anything above 0.95 is suspiciously good.

AUC also has an excellent probabilistic interpretation. Say that we are given two data instances, and we are told that one is positive and the other is negative. We use the classifier to estimate the probabilities of being positive for each instance and decide that the one with the highest probability is positive. It turns out that the probability that such a decision is correct equals the AUC of this classifier. Hence, AUC measures how well the classifier discriminates between the positive and negative instances.

From another perspective: if we use a classifier to rank data instances, then AUC of 1 signifies a perfect ranking, an AUC of 0.5 a random ranking, and an AUC of 0 an ideal reversed ranking.

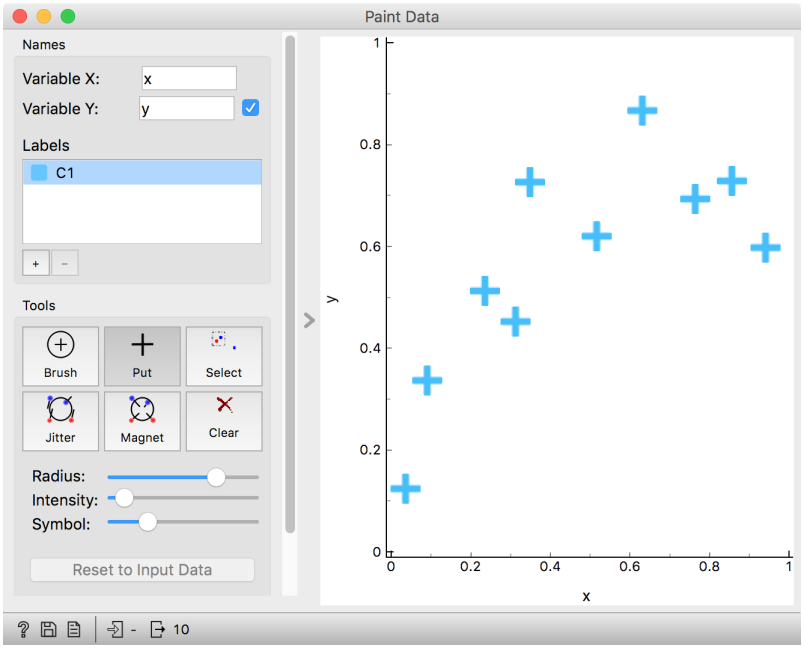
Linear Regression

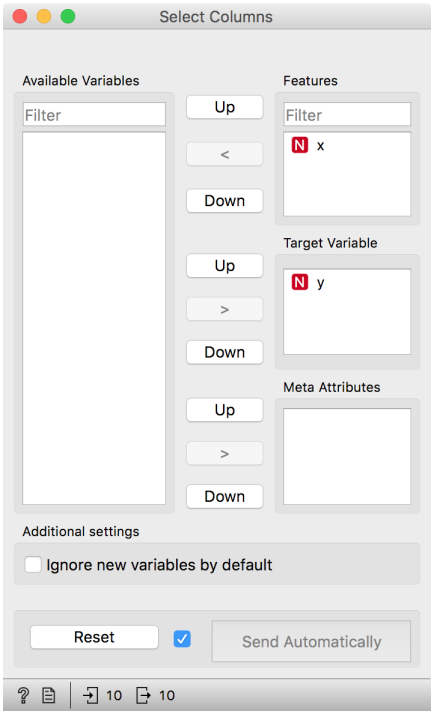
In the *Paint Data* widget, remove the C2 label from the list. If you have accidentally left it while painting, don't despair. The class variable will appear in the *Select Columns* widget, but you can "remove" it by dragging it into the Available Variables list.



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 use Put instead of the Brush tool.

We want to build a model that predicts the value of the target variable y from the feature x. Say that we would like our model to be linear, to mathematically express it as $h(x) = \theta_0 + \theta_1 x$. Oh, this is the equation of a line. So we would like to draw a line through our data points. The θ_0 is then an intercept, and θ_1 is a slope. But there are many different lines we could draw. Which one is the best? Which one is the one that fits our data the most? Are they the same?

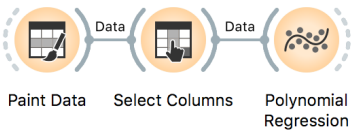


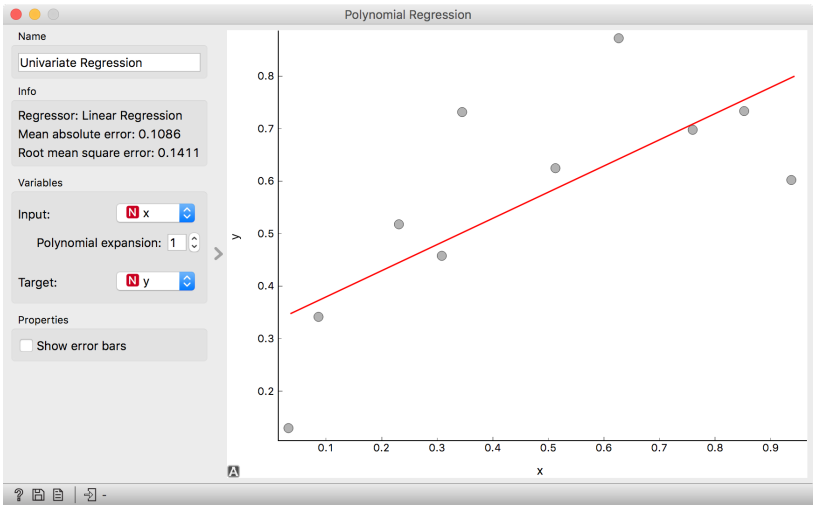


The question above requires us to define what a good fit is. Say, this could be the error the fitted model (the line) makes when it predicts the value of y for a given data point (value of x). The prediction is h(x), so the error is $h(x) - y$. We should treat the negative and positive

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! It 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*.

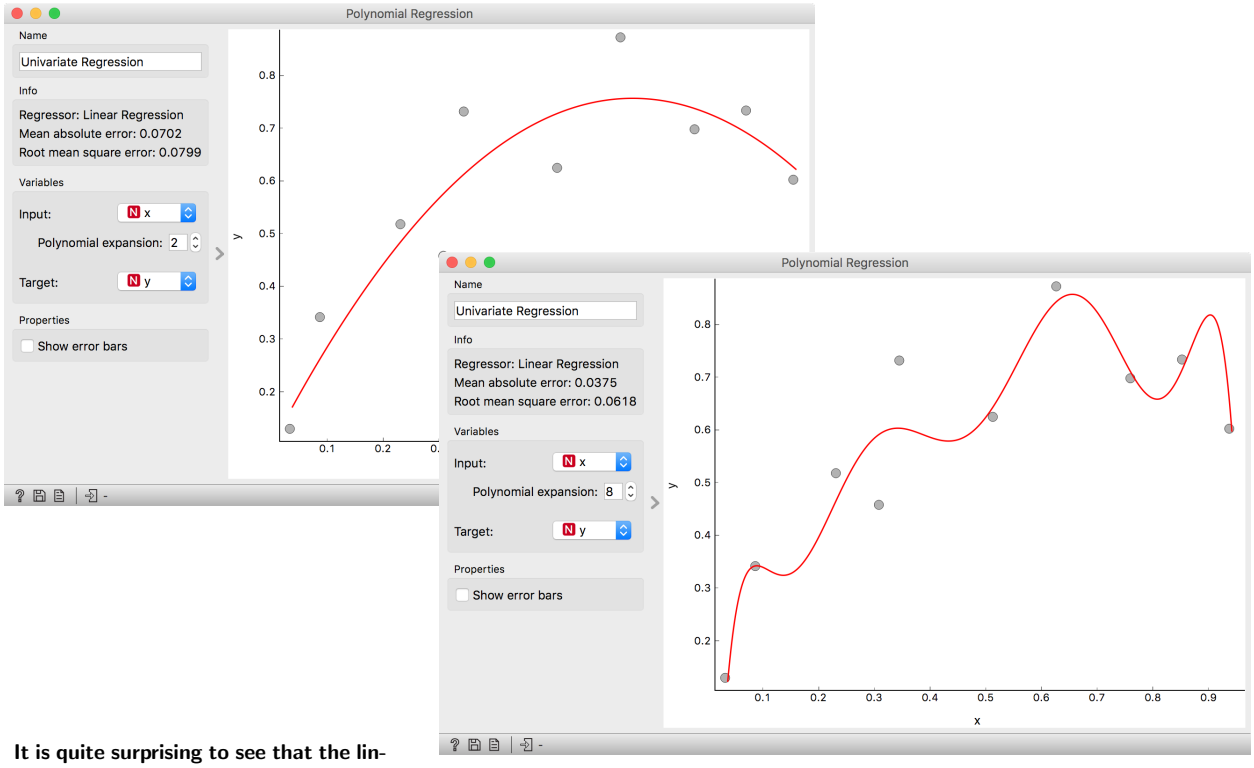
Do not worry about the strange name of the *Polynomial Regression*, we will get there in a moment.





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, ..., 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 $h(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$.

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

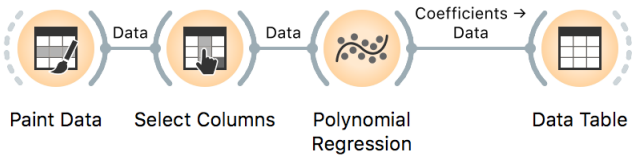


It is quite surprising to see that the linear regression model can fit non-linear (univariate) functions. It can fit the data with curves, such as those on the figures. How is this possible? Notice, though, that the model is a hyperplane (a flat surface) in the space of many features (columns) that are 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.

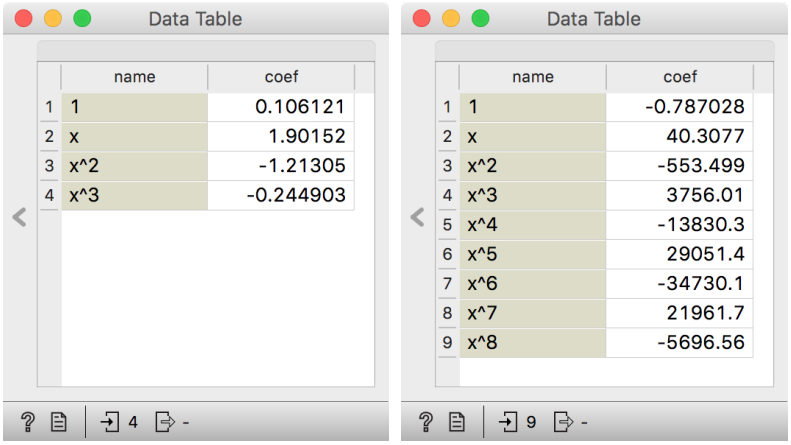
Overfitting is related to the complexity of the model. In polynomial regression, the parameters θ define the model. With the increased number of parameters, the model complexity increases. The simplest model has just one parameter (an intercept), ordinary linear regression has two (an intercept and a slope), and polynomial regression models have as many parameters as the polynomial degree. It is easier to overfit the data with a more complex model, as it can better adjust to the data. But is the overfitted model discovering the true data patterns? Which of the two models depicted in the figures above would you trust more?

Regularization

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



With smaller degree polynomials, values of θ stay small, but then as the degree goes up, the numbers get huge.

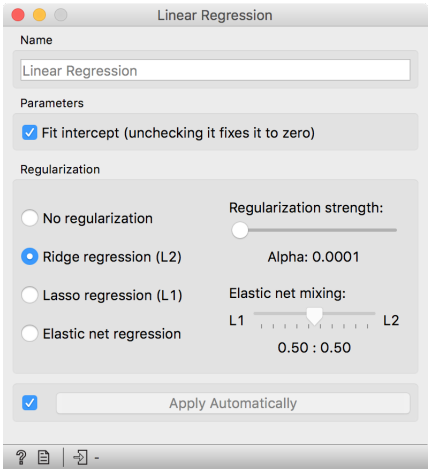


Which inference of linear model would overfit more, the one with high λ or with low λ? What should the value of λ be to cancel regularization? What if the value of λ is high, say 1000?

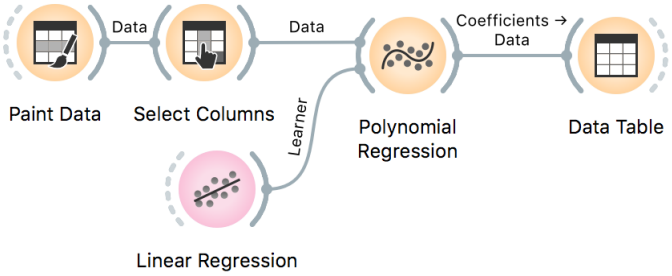
More complex models can fit the training data better. The fitted curve can wiggle sharply. The derivatives of such functions are high, so the coefficients θ need be. If only we could force the linear regression to infer models with a small value of coefficients. Oh, but we can. Remember, we have started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

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

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



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 θ. Lasso regression minimizes the sum of the absolute value of coefficients. Although the difference may seem negligible, the consequences are that lasso regression may result in a large proportion of coefficients θ being zero, in this way performing feature subset selection.

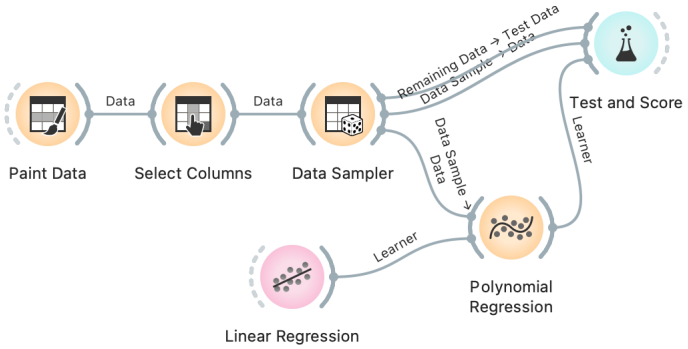


Now for the test. Increase the degree of polynomial to the max. Use Ridge Regression. Does the inferred model overfit the data? How does the degree of overfitting depend on regularization strength?

Regularization and Accuracy on a Test Set

Overfitting hurts. Overfit models fit the training data well but can perform miserably on new data. Let us observe this effect in regression. We will use hand-painted data set, split it into the training (50%) and test (50%) data set, polynomially expand the training data set to enable overfitting and build a model. We will test the model on the (seen) training data and the (unseen) held-out data.

Paint about 20 to 30 data instances. Use the attribute y as the target variable in Select Columns. Split the data 50:50 in Data Sampler. Cycle between test on train or test data in Test and Score. Use ridge regression to build a linear regression model.



Now we can vary the regularization strength in *Linear Regression* and observe the accuracy in *Test and Score*. For accuracy scoring, we will use RMSE, root mean squared error, which is computed by observing the error for each data point, squaring it, averaging this across all the data instances, and taking a square root.

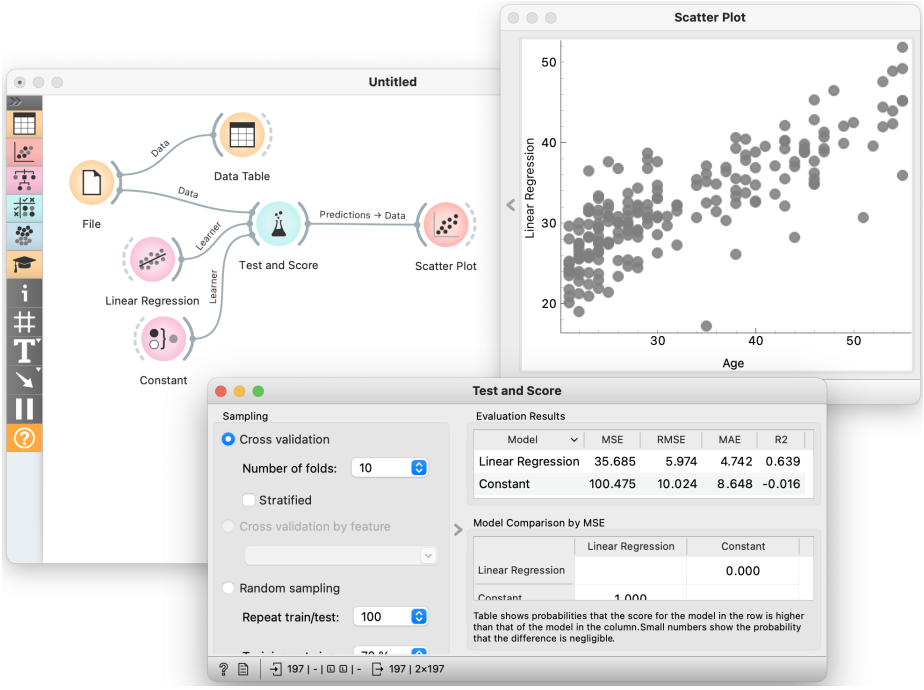
The core of this lesson is to compare the error on the training and test set while varying the level of regularization. Remember that regularization controls overfitting. The more we regularize, the less tightly we fit the model to the training data. So for the training set, we expect the error to drop with less regularization and more overfitting. The error on the training data increases with more regularization and less fitting. We expect no surprises here. But how does this play out on the test set? Which sides minimizes the test-set error? Or is the optimal level of regularization somewhere in between? How do we estimate this level of regularization from the training data alone?

Orange is currently not equipped with the fitting of meta parameters, like the degree of regularization, and we need to find their optimal values manually. At this stage, it suffices to say that we must infer meta parameters from the training data set without touching the test data. If the training data set is sufficiently large, we can split it into a set for training the model and a data set for validation. Again, Orange does not support such optimization yet, but it will sometime in the future. :)

Regularization

Download the methylation data set from `http://file.biolab.si/datasets/methylation.pkl.gz`. **Predictions of age from methylation profile were investigated by Horvath (2013) Genome Biology 14:R115.**

Enough painting. Now for the real data. We will use a data set that includes human tissues from subjects of different ages. The tissues were profiled by measurements of DNA methylation, a mechanism for cells to regulate gene expression. Methylation of DNA is scarce when we are young and gets more abundant as we age. We have prepared a data set where the degree of methylation was expressed per gene. Let us test if we can predict the age from the methylation profile - and if we can do this better than by just predicting the average age of subjects in the training set.

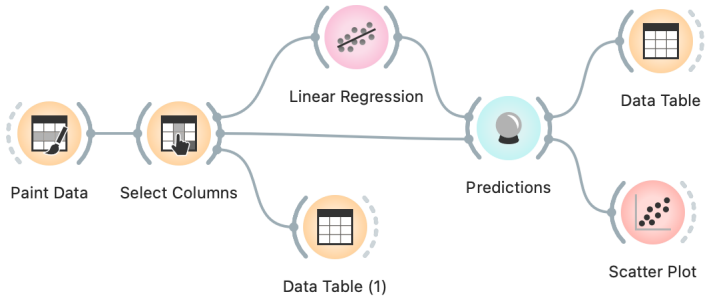


Using other learners, like random forests, takes a while on this data set. But you may try to sample the features, obtain a smaller data set, and try various regression learners.

This workflow looks familiar and is similar to those for classification problems—-the *Test and Score* widget reports on statistics we have not seen before. MAE, for one, is the mean average error. As for classification, we have used cross-validation. Mean average error was computed only on the test data instances and averaged across ten cross-validation runs. The results indicate that our modeling technique misses the age by about five years, which is a much better result than predicted by the mean age in the training set.

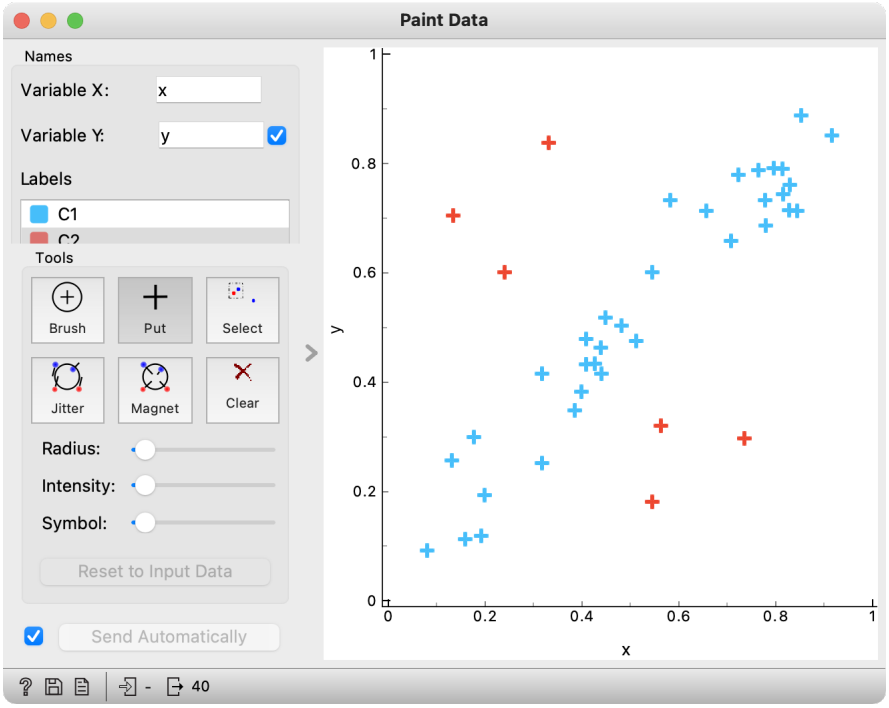
Evaluating Regression

The last lessons quickly introduced scoring for regression and essential measures such as RMSE and MAE. In classification, the confusion matrix was an excellent addition to finding misclassified data instances. But the confusion matrix could only be applied to discrete classes. Before Orange gets some similar for regression, one way to find misclassified data instances is through scatter plot!



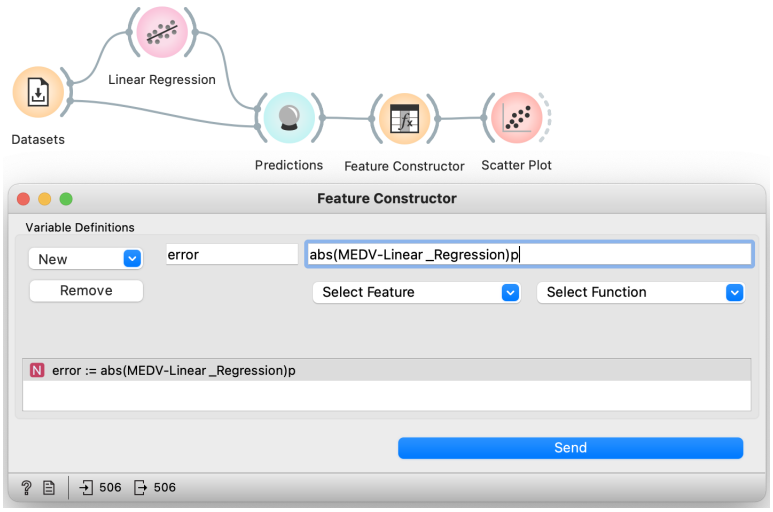
This workflow visualizes the predictions that we have constructed on the training data. How would you change the widget to use a separate test set? Hint: The Sample widget can help.

We can play around with this workflow by painting the data such that the regression would perform well on the blue data point and fail on the red outliers. In the scatter plot, we can check if the predicted and true class difference was what we had expected.



A similar workflow would work for any data set—for instance, the

housing data set (from Orange distribution). Say, just like above, we would like to plot the relationship between true and predicted continuous class, but would like to add information on the absolute error the predictor makes. Where is the error coming from? We need a new column. The Feature Constructor widget (albeit a bit geekish) comes to the rescue.

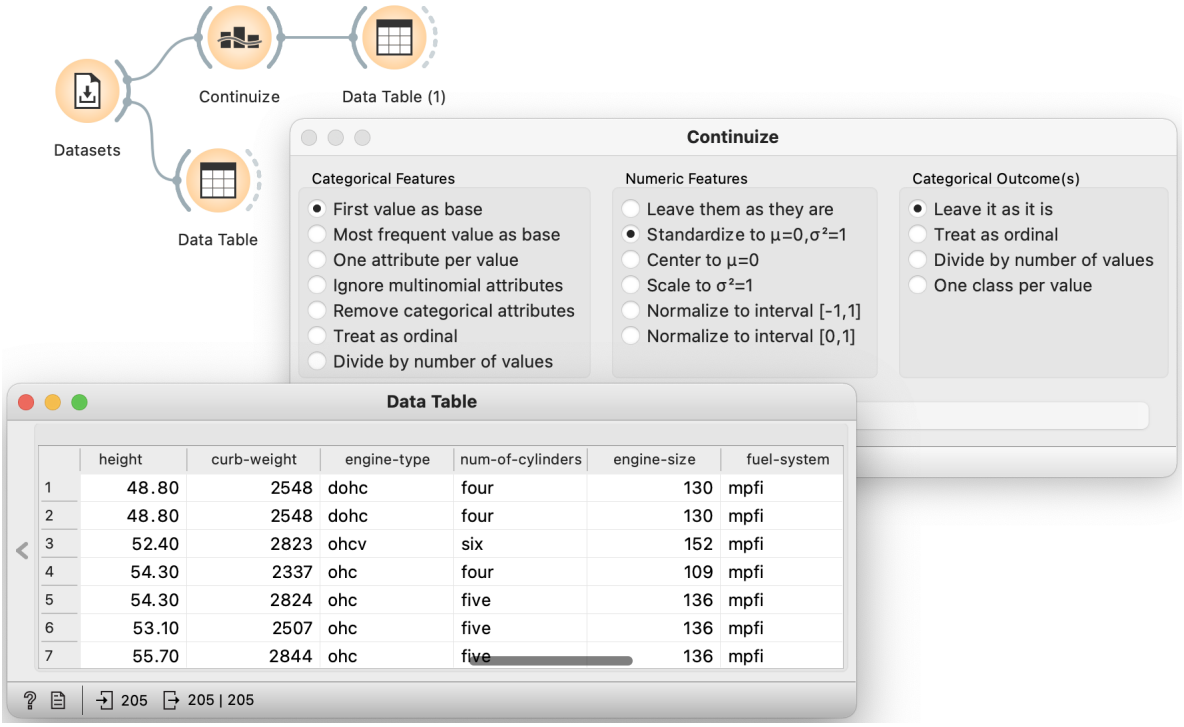


In the *Scatter Plot*, we can now select the data where the predictor erred substantially and explore the results further.

We could, in principle, also mine the errors to see if we can identify data instances for which this was high. But then, if this is so, we could have improved predictions at such regions. Like, construct predictors that predict the error. Creating such predictors looks weird. Could we then also build a predictor that predicts the error of the predictor that predicts the error? Strangely enough, such ideas have recently led to something called Gradient Boosted Trees, which are nowadays among the best regressors. Check them out using the Gradient Boosting widget.

Feature Scoring and Selection

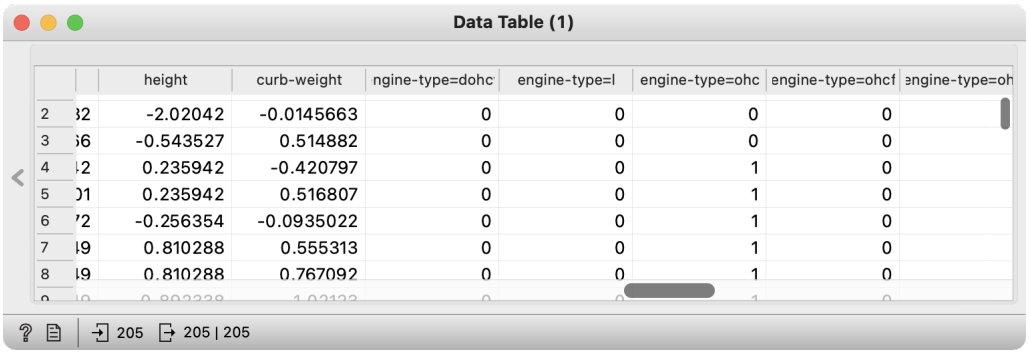
Linear regression infers a model that estimates the class, a real-valued feature, as a sum of products of input features and their weights. Consider the data on prices of imported cars in 1985. Inspecting this data set in a *Data Table* shows that some features, like fuel-system, engine-type, and many others, are discrete. Linear regression only works with numbers. In Orange, linear regression will automatically convert all discrete values to numbers, often using several features to represent a single discrete feature. We also do this conversion manually by using the *Continuize* widget.

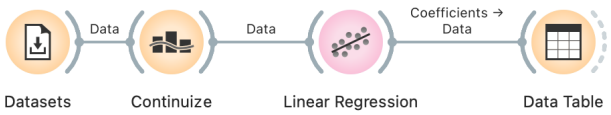


Before continuing, you should check what *Continuize* does and how it converts the nominal features into real-valued features. The table below should provide sufficient illustration.

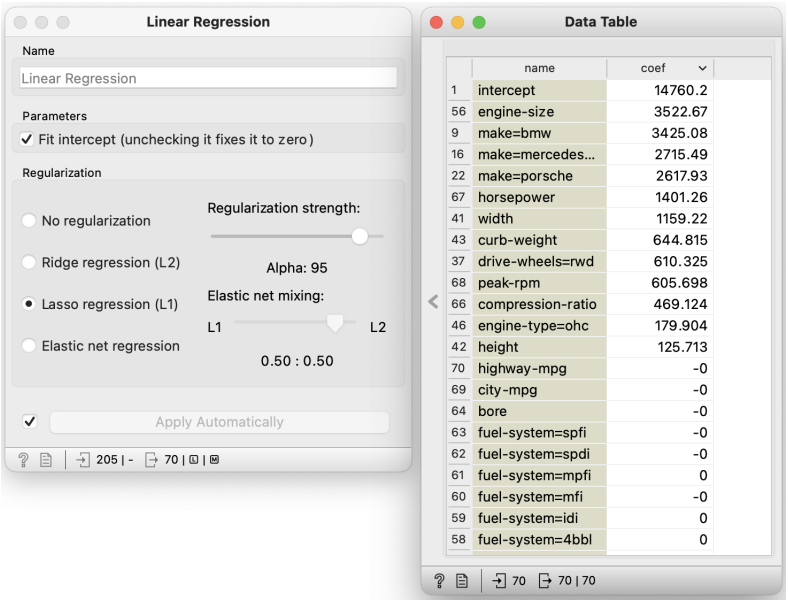
Now to the core of this lesson. Our workflow reads the data continuizes it such that we also normalize all the features to bring them to equal scale. We then load the data into the *Linear Regression* widget and check out the feature coefficients in the *Data Table*.

In *Linear Regression*, we will use L1 regularization. Compared to L2 regularization, which aims to minimize the sum of squared weights, L1 regularization is rougher: it minimizes the sum of absolute values of the weights. The result of this "roughness" is that many of the





features will get zero weights. But this feature elimination may also be exactly what we want. We want to select only the most important features and see how the model that uses only a smaller subset of features behaves. Also, this smaller set of features is ranked. Engine size is a huge factor in the pricing of our cars, and so is the make, where Porsche, Mercedes, and BMW cost more than other cars (ok, no news here).

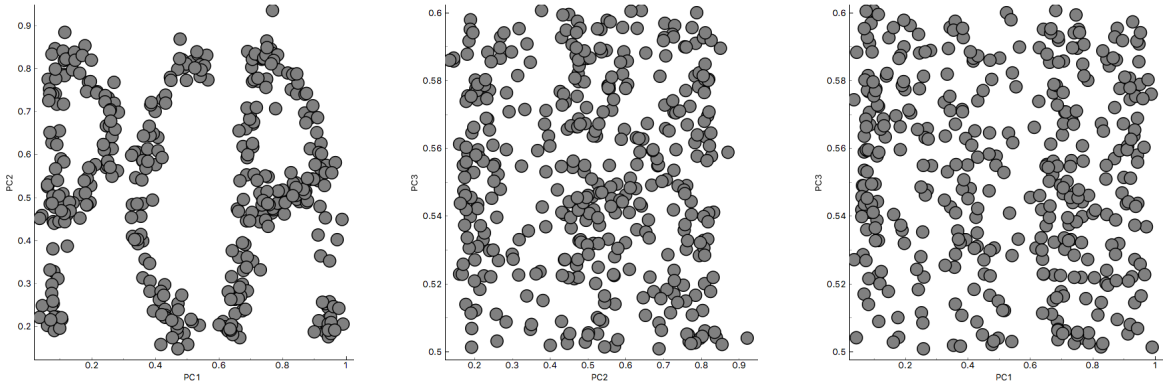


We care about features with substantial weights, regardless of their sign. Therefore, we should change the workflow to compute and show the data features by their absolute weight. Could you change the workflow accordingly? Hint: use the Feature Construction widget.

We should notice that the number of features with non-zero weights varies with regularization strength. Stronger regularization would result in fewer features with non-zero weights.

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?

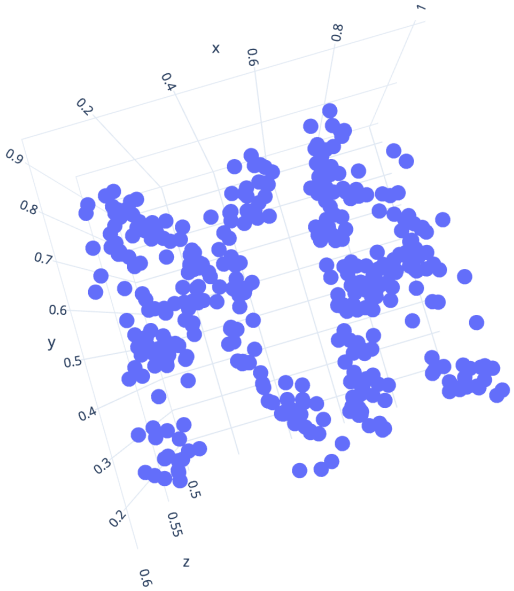


Yes, the first scatter plot looks very useful: it tells us that x and y are highly correlated and that we have three clusters of somewhat irregular shape. But remember: this data is three dimensional. What is we saw it from another, perhaps better perspective?

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.

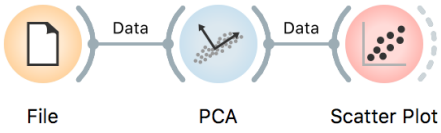
It turns out the flies are actually also spread in the third direction. Thus you need three numbers after all.

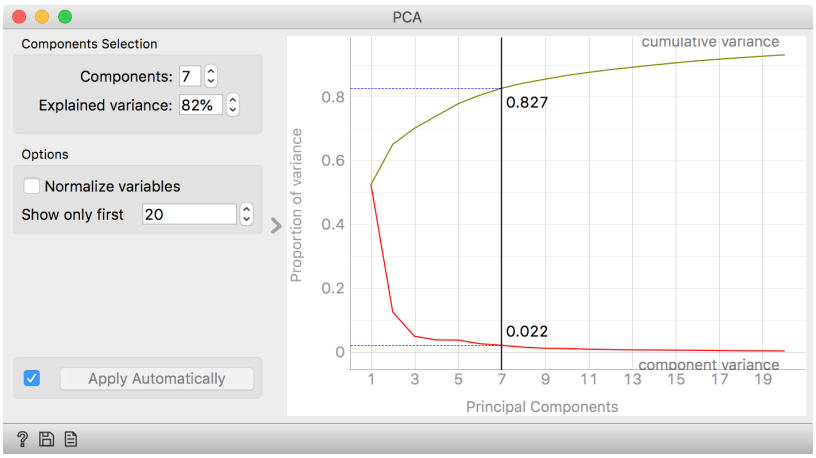
Or do you? It all depends on how spread they are in the second and in the third direction. If the spread along the second is relatively small in comparison with the first, you are fine with a single dimension. If not, you need two, but perhaps still not three.

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.

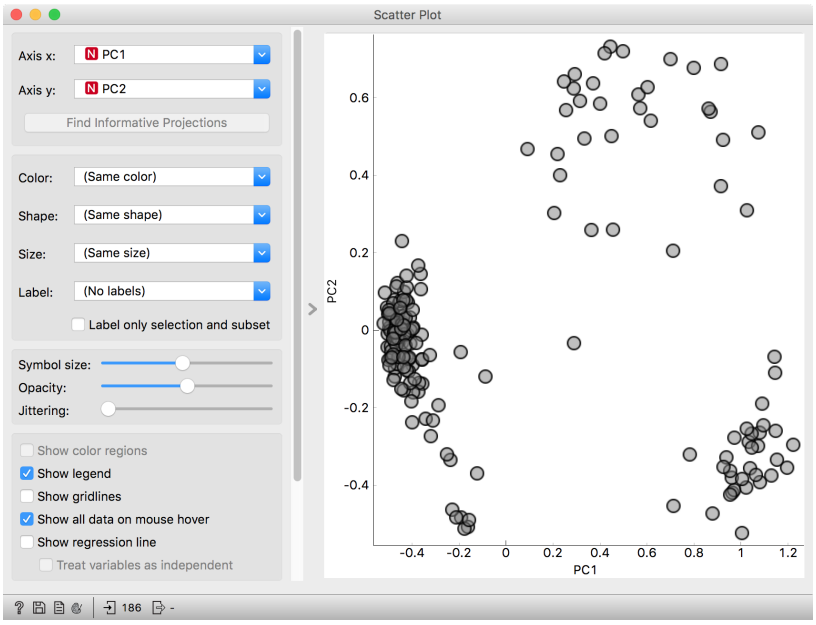
A technique that does this is called Principle Components Analysis, or *PCA*. The corresponding widget is simple: it receives the data and outputs the transformed data.

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.



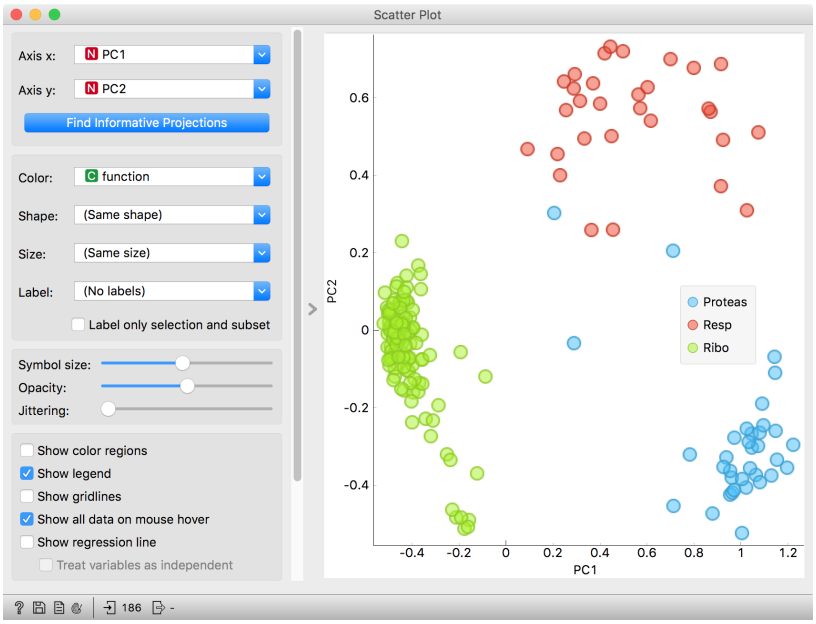


The PCA on the left shows the scree diagram for brown-selected data. Set like this, the widget replaces the 80 features with just seven - and still keeping 82.7% of information. (Note: disable "Normalize data" checkbox to get the same picture.) Let us see a scatter plot for the first two components.

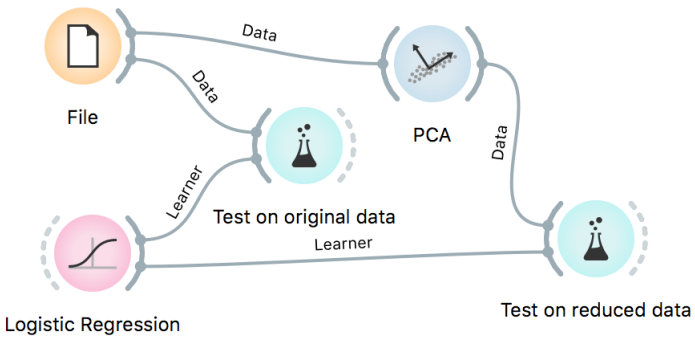


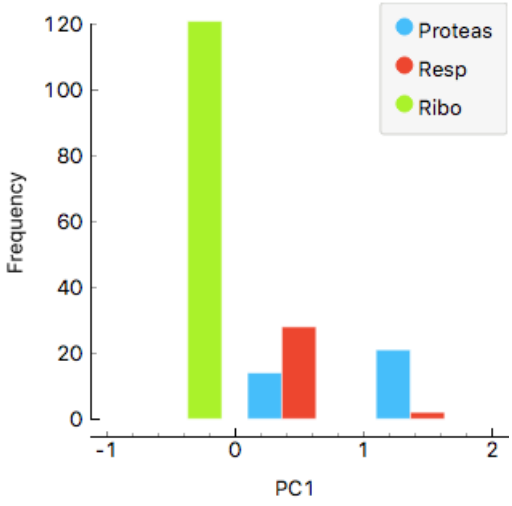
The axes, PC1 and PC2, do not correspond to particular features in the original data, but to their linear combination. What we are looking at is a projection onto the plane, defined by the first two components. When you consider only two components, you can imagine that PCA puts a hyperplane into multidimensional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

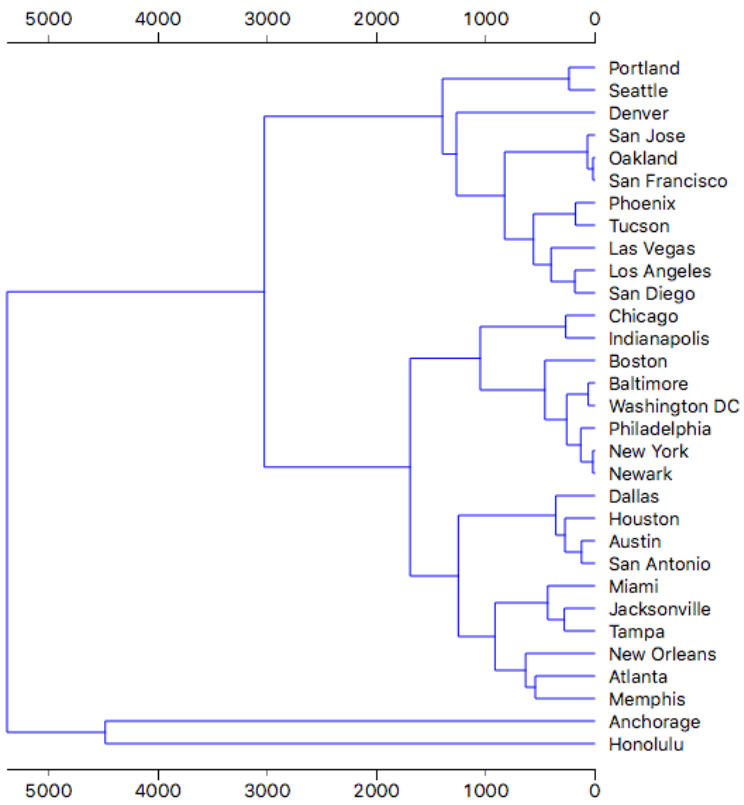
Mapping the Data

Imagine a foreign visitor to the US who knows nothing about the US geography. He doesn't even have a map; the only data he has is a list of distances between the cities. Oh, yes, and he attended the Introduction to Data Mining.

If we know distances between the cities, we can cluster them.

For this example we retrieved the data from http://www.mapcrow.info/united_states.html, removed the city names from the first line and replaced it with "31 labelled".

The file is available at http://file.biolab.si/files/us-cities.dst.zip. To load it, unzip the file and use the *File Distance* widget.



How much sense does it make? Austin and San Antonio are closer to each other than to Houston; the tree is then joined by Dallas. On the other hand, New Orleans is much closer to Houston than to Miami. And, well, good luck hitchhiking from Anchorage to Honolulu.

As for Anchorage and Honolulu, they are leftovers; when there were only three clusters left (Honolulu, Anchorage and the big cluster with everything else), Honolulu and Anchorage were closer to each other than to the rest. But not close — the corresponding lines in the dendrogram are really long.

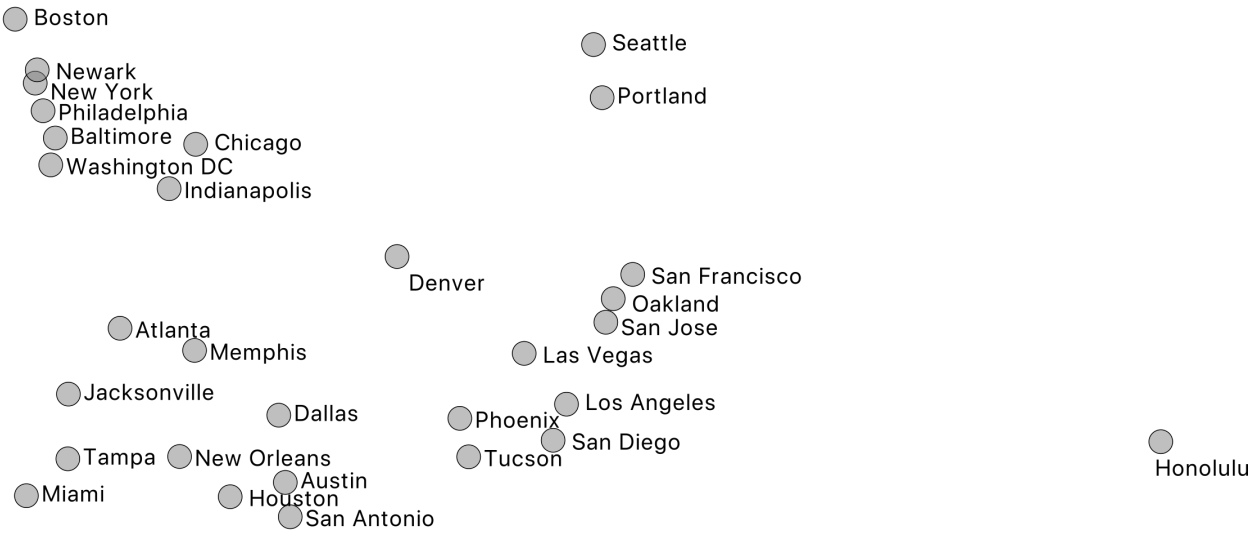
The real problem is New Orleans and San Antonio: New Orleans is close to Atlanta and Memphis, Miami is close to Jacksonville and

Tampa. And these two clusters are suddenly more similar to each other than to some distant cities in Texas.

In general, two points from different clusters may be more similar to each other than to some points from their corresponding clusters.

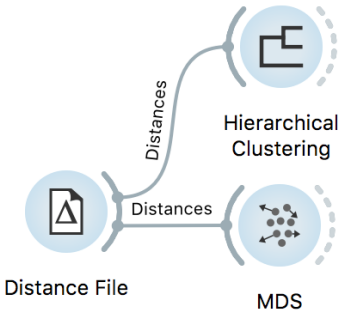
To get a better impression about the physical layout of cities, people have invented a better tool: a map! Can we reconstruct a map from a matrix of distances? Sure. Take any pair of cities and put them on a paper with the distance corresponding to some scale. Add the third city and put it at the corresponding distance from the two. Continue until done. Excluding, for the sake of scale, Anchorage, we get the following map.

We can't run k-means clustering on this data, since we only have distances, and k-means runs on real (tabular) data. Yet, k-means would have the same problem as hierarchical clustering.

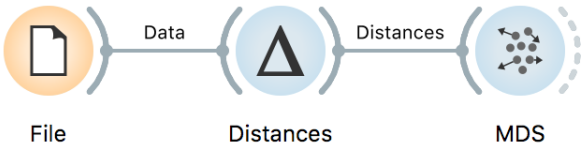


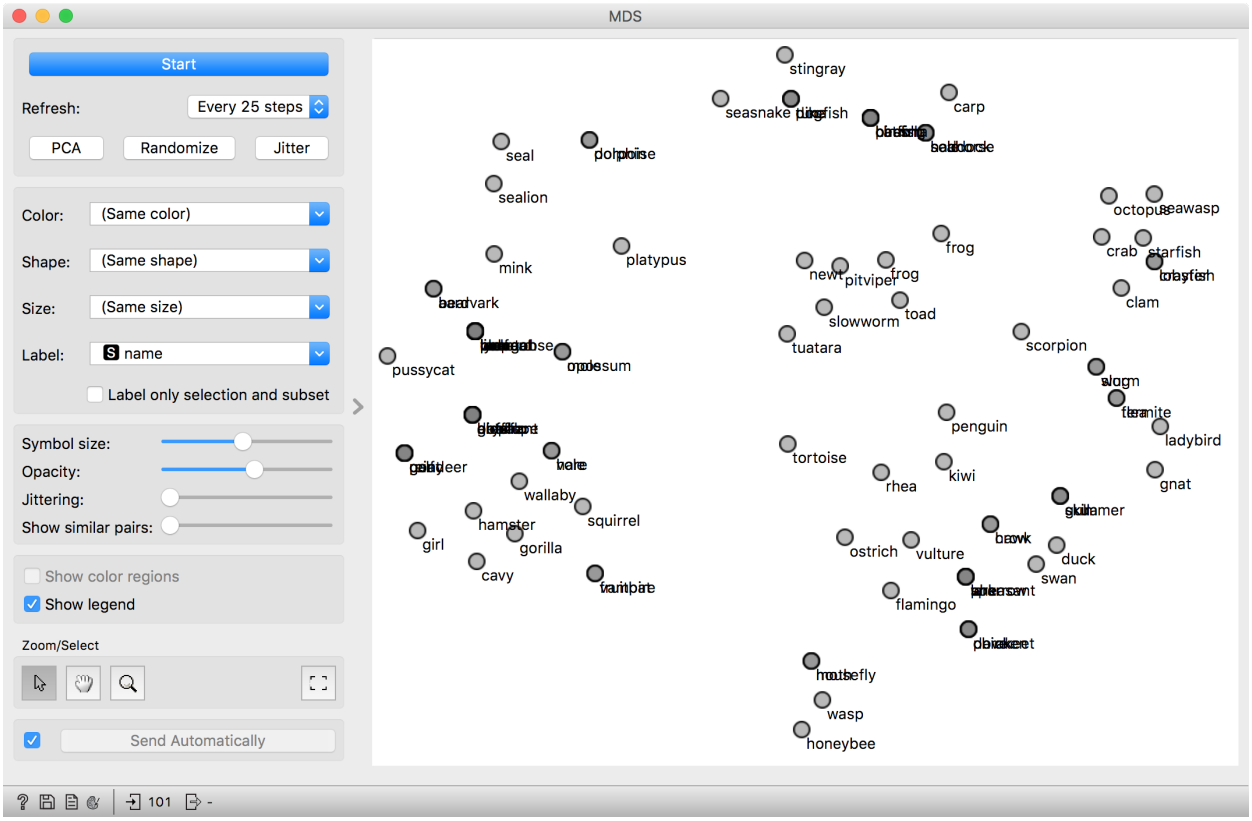
We have not constructed this map manually, of course. We used a widget called *MDS*, which stands for Multidimensional scaling.

It is actually a rather exact map of the US from the Australian perspective. You cannot get the orientation from a map of distances, but now we have a good impression about the relations between cities. It is certainly much better than clustering.



Remember the clustering of animals? Can we draw a map of animals? Does the map make any sense? Are similar animals together? Color the points by the types of animals and you should see.





The map of the US was accurate: one can put the points in a plane so that the distances correspond to actual distances between cities. For most data, this is usually impossible. What we get is a projection (a non-linear projection, if you care about mathematical finesses) of the data. You lose something, but you get a picture.

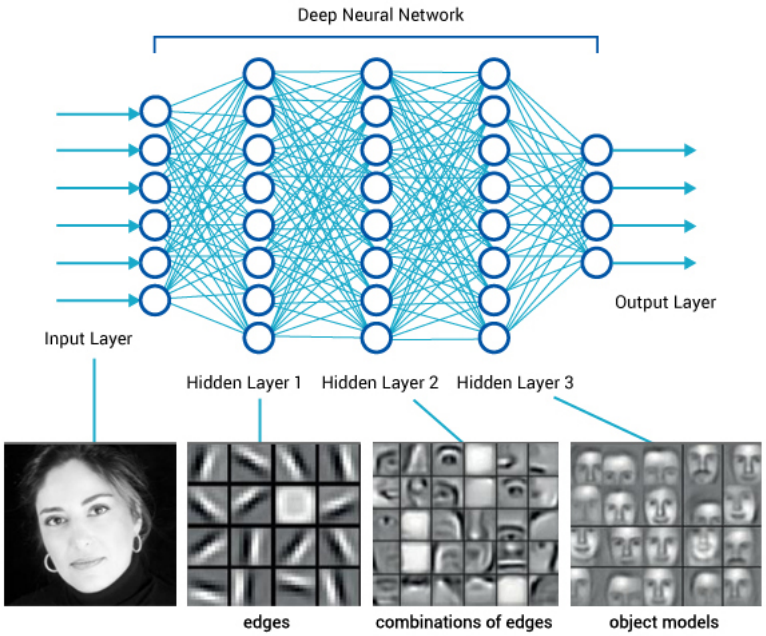
The MDS algorithm does not always find the optimal map. You may want to restart the MDS from random positions. Use the slider "Show similar pairs" to see whether the points that are placed together (or apart) actually belong together. In the above case, the honeybee belongs closer to the wasp, but could not fly there as in the process of optimization it bumped into the hostile region of flamingos and swans.

Image Embedding

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.

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

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.

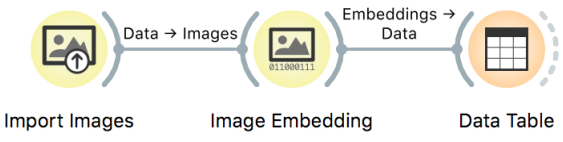
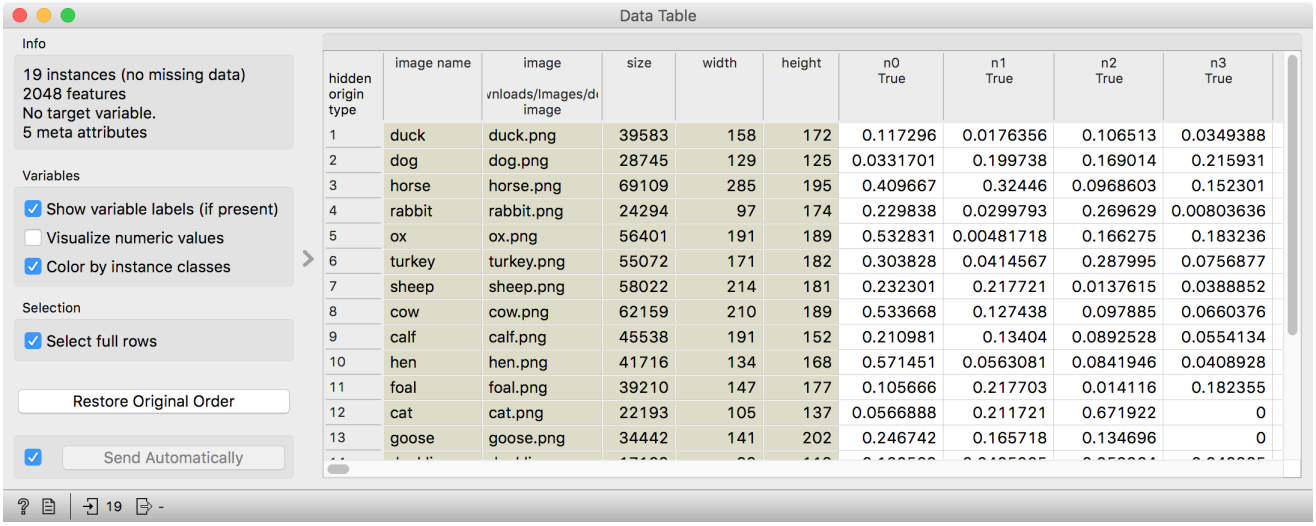


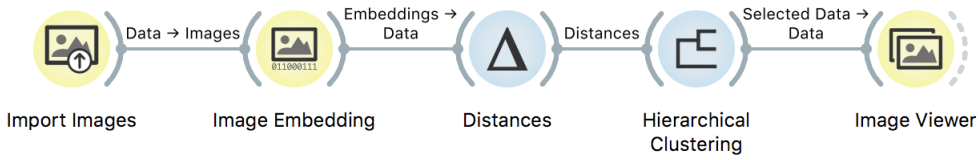
Image embedding describes the images with a set of 2048 features appended to the table with meta features of images.

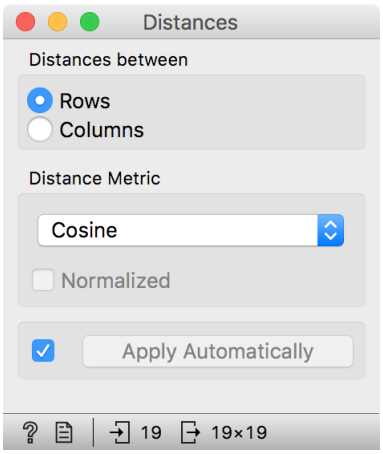


hidden origin type	image name	image /nloads/Images/d image	size	width	height	n0 True	n1 True	n2 True	n3 True
1	duck	duck.png	39583	158	172	0.117296	0.0176356	0.106513	0.0349388
2	dog	dog.png	28745	129	125	0.0331701	0.199738	0.169014	0.215931
3	horse	horse.png	69109	285	195	0.409667	0.32446	0.0968603	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

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

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.





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).

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