BIOLAB AND COLLABORATORS

USING QUASAR

BIOLAB

PUBLISHED BY BIOLAB

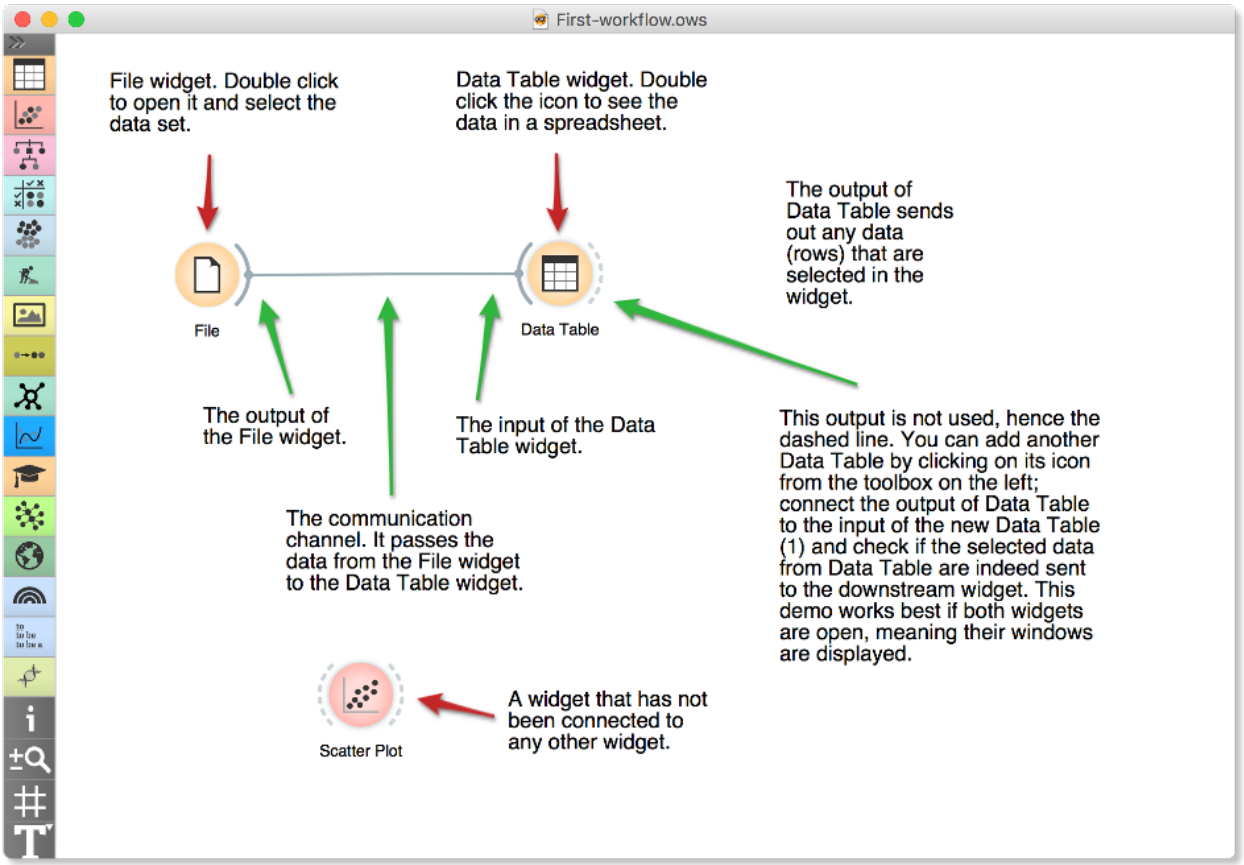
TUFTE-LATEX.GOOGLECODE.COM

First printing, November 2022

Contents

Workflows in Quasar

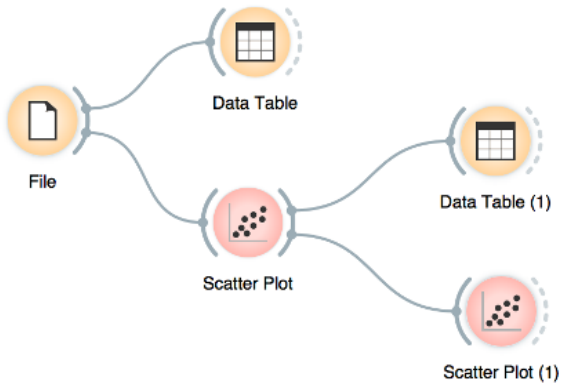
Quasar workflows consist of components that read, process, and visualize data. We refer to these components as "widgets" We place the widgets on a drawing board called the "canvas" to design a workflow. Widgets communicate by sending information along their communication channel. Output from one widget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

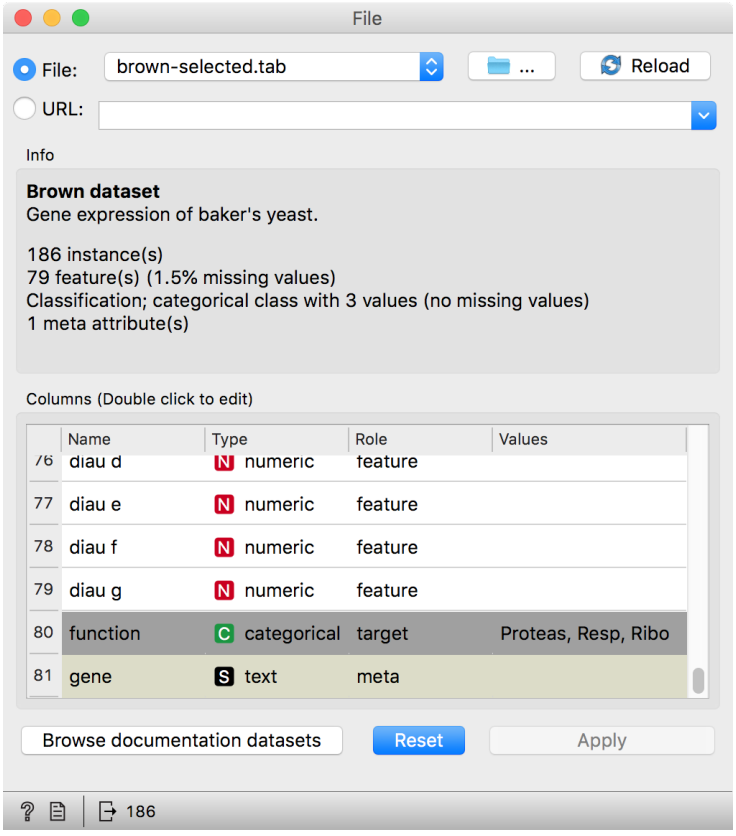
We construct workflows by dragging widgets onto the canvas and connecting them by drawing a line from the transmitting widget to the receiving widget. The widget's outputs are on the right, and the inputs on the left. In the workflow above, the *File* widget sends data to the *Data Table* widget.

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. Quasar comes with several preloaded data sets. From these ("Browse documentation data sets..."), choose *brown-selected.tab*, a yeast gene expression data set.



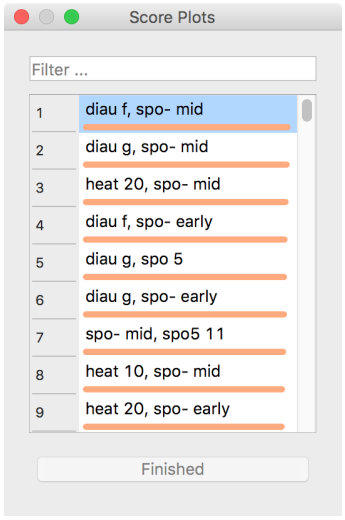
Quasar workflows often start with a File widget. The brown-selected data set comprises 186 rows (genes) and 81 columns. Out of the 81 columns, 79 contain gene expressions of baker's yeast under various conditions, one column (marked as a "meta attribute") provides gene names, and one column contains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

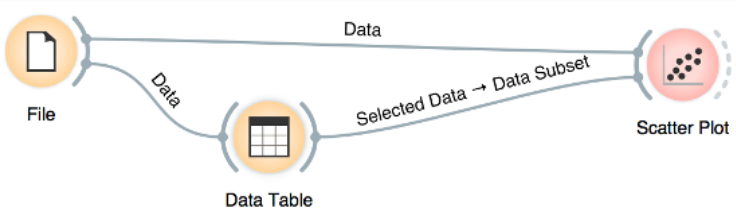
3. Use a combination of two *Scatter Plot* widgets, where the second scatter plot shows a detail from a smaller region selected in the first scatter plot.

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?



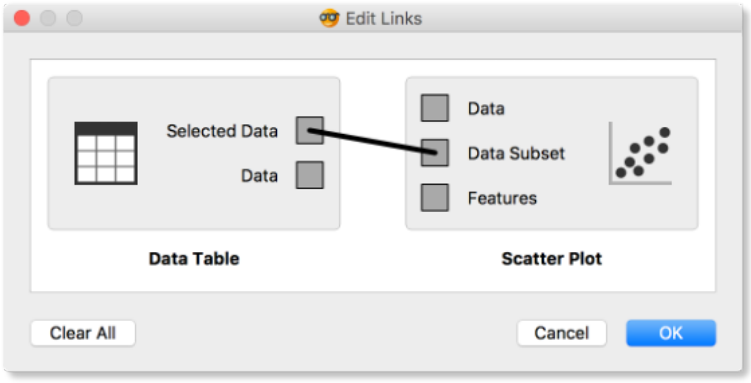
Scatter Plot and Ranking

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.



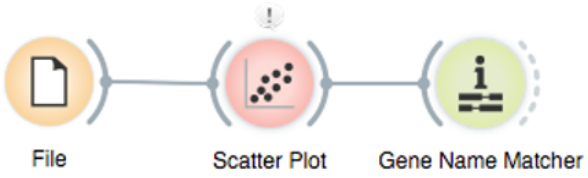
In this workflow, we have switched on the option "Show channel names between widgets" in File/Preferences.

How does Quasar 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 Quasar widgets.

Quasar 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.

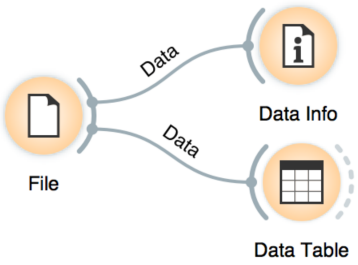


Basic data exploration

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 Quasar better, and also learn about several interesting visualizations.

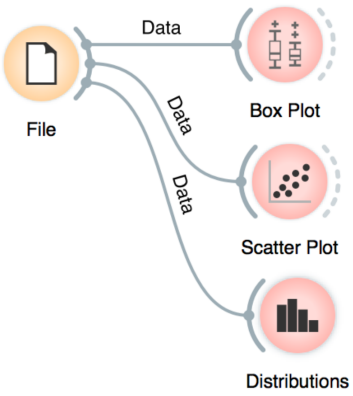
We will start with an empty canvas; to clean it from our previous lesson, use either File/New or select all the widgets and remove them (use the backspace/delete key).

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.

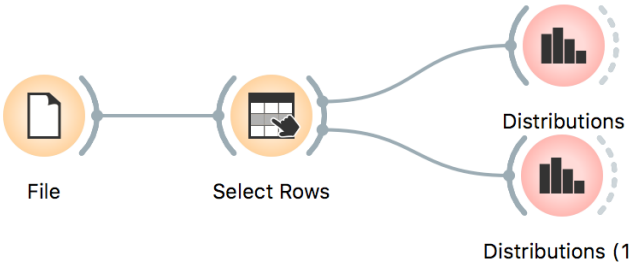
Let us check whether common visualizations tell us anything interesting. (Hint: look for gender differences. These are always interesting and occasionally even real.)



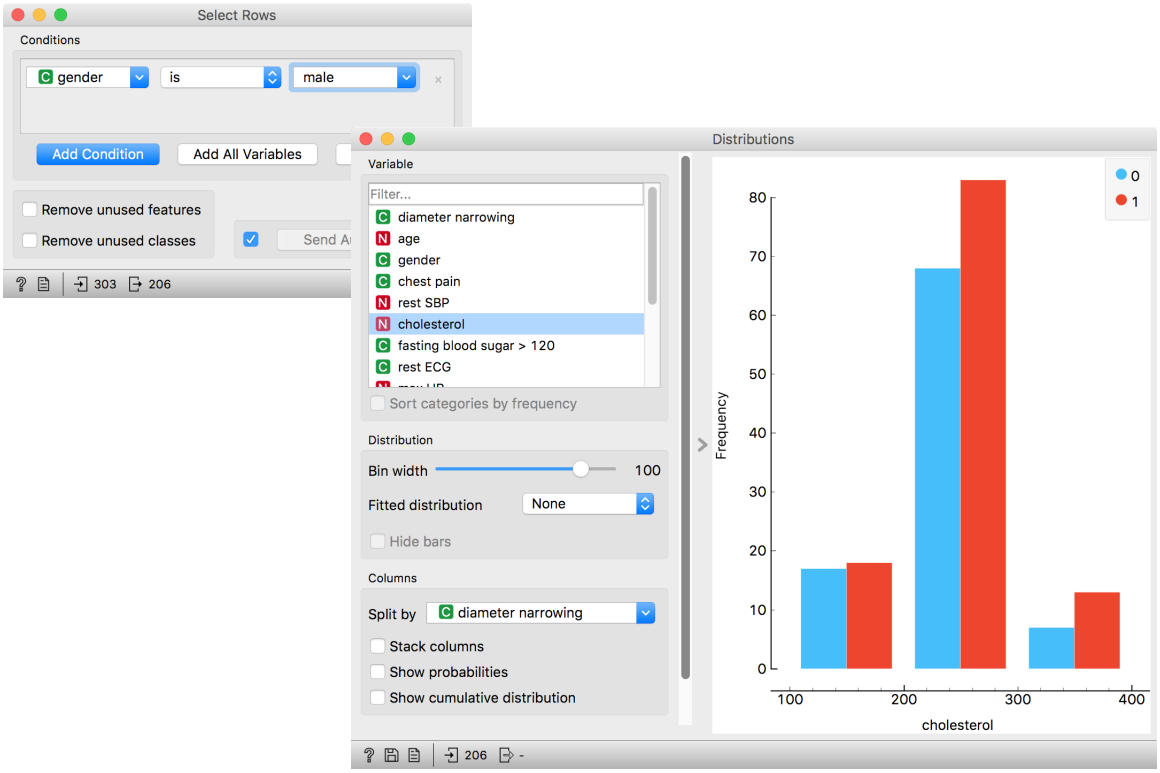
Quick check with common statistics and visualization widgets.

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.



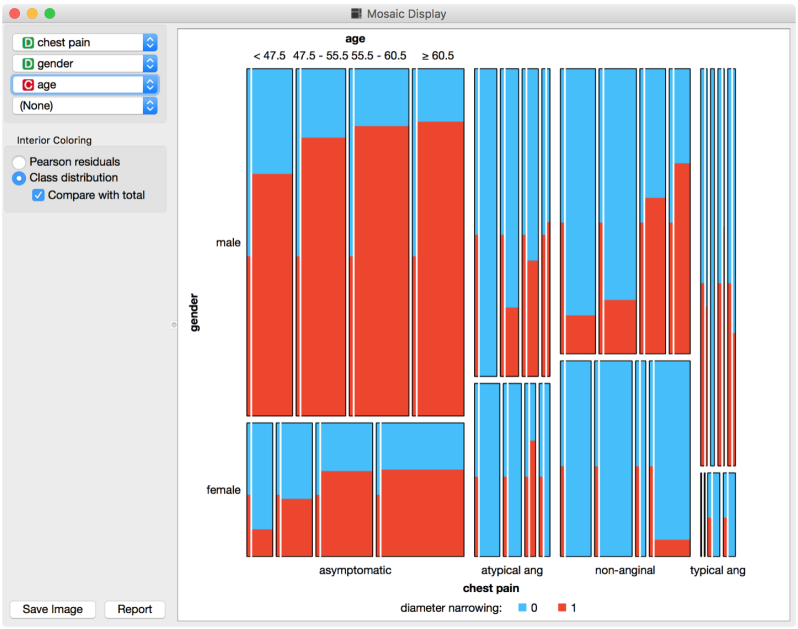
In the *Select Rows* widget, we select the female patients. You can also add other conditions. The selection of data instances provides a powerful combination with visualization of data distribution. Try having at least two widgets open simultaneously and explore the data.



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

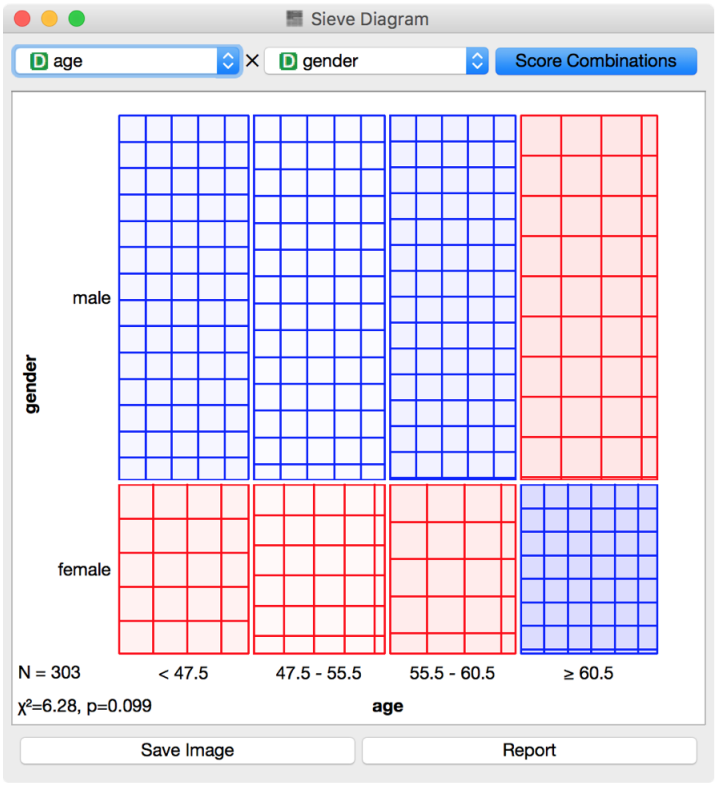
The mosaic display shows a rectangle split into columns with widths reflecting the prevalence of different types of chest pain. Each column is then further split vertically according to gender distributions within the column. The resulting rectangles are split again horizontally according to age group 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.

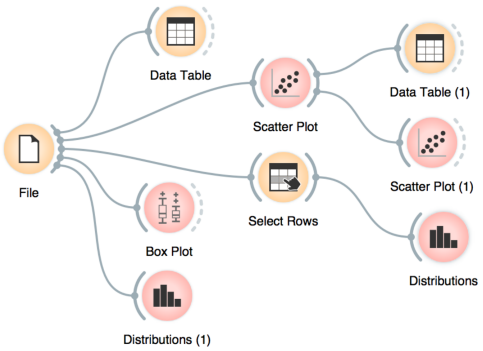


See the Score Combinations button? Try to guess what it does? And how does it score the combinations? Hint: there are some Greek letters at the bottom of the widget.

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.

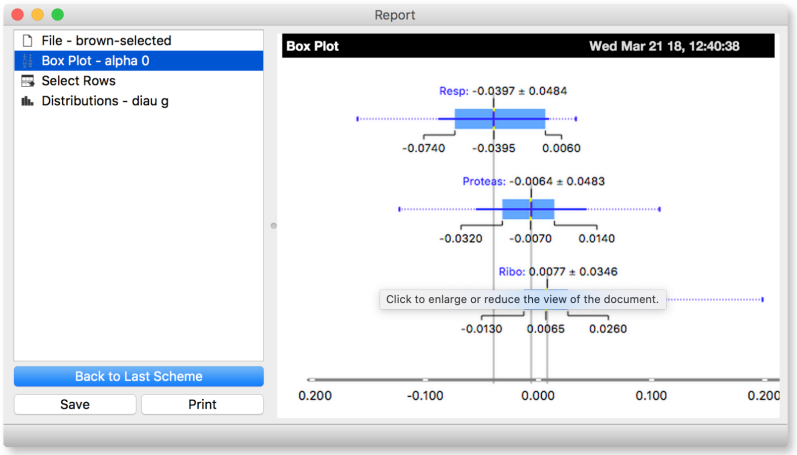


You can save this workflow using the File/Save menu and share it with your colleagues. Just don't forget to put the data files in the same directory as the file with the workflow.

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.

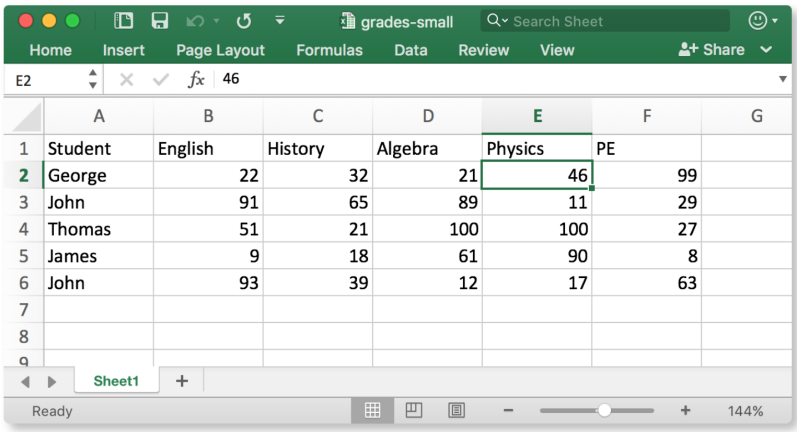
The report window and the additional text input box (bootom).



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

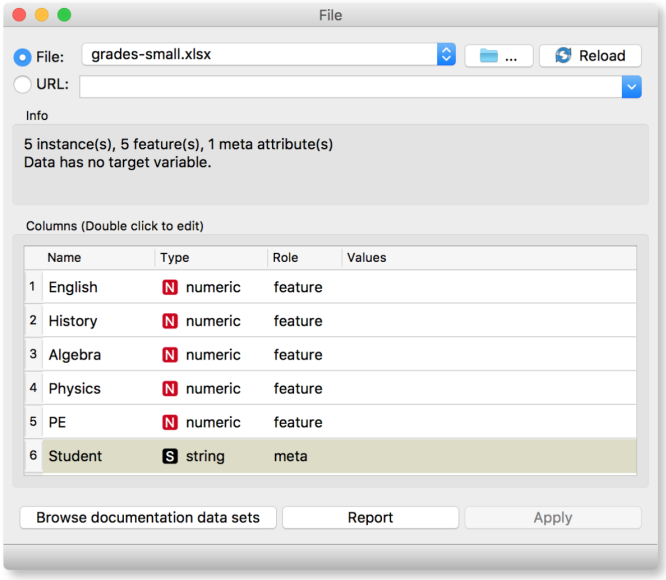
The data sets we have worked with in the previous lesson come with the Quasar installation. Quasar can read data from many file formats which include tab and comma separated and Excel files. To see how this works, let's prepare a data set (with school subjects and grades) in Excel and save it on a local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

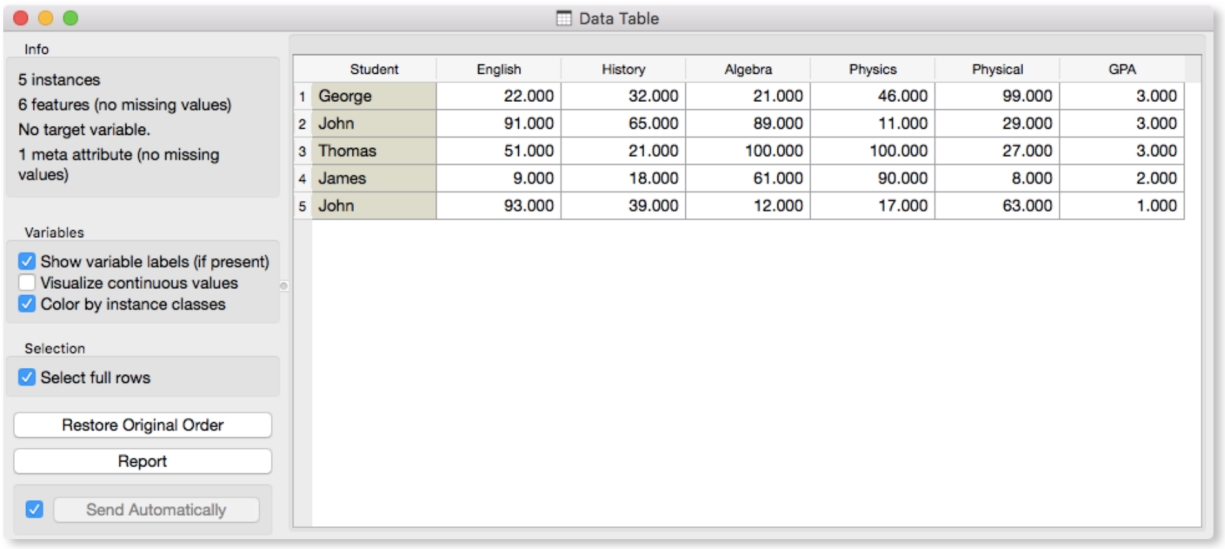
Looks good! Quasar has correctly guessed that student names are character strings and that this column in the data set is special, meant

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!

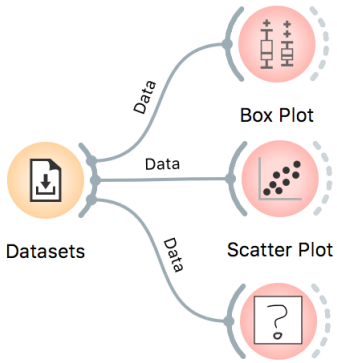
Instead of using Excel, we could also use Google Sheets, a free online spreadsheet alternative. Then, instead of finding the file on the local disk, we would enter its URL address to the File widget URL entry box.

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

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

Assignment: Data Inspection

Understanding the data is crucial for any data science task. And the best way to achieve this is with visualizations. With the *Datasets* widget load *Employee Attrition* data set, which describes employees of a company with 32 features, such as age, job role, years at company, and so on. We also know, whether an employee resigned from the company (Attrition = Yes) or not (Attrition = No).



Workflow for the assignment.

Inspect the data to understand it better:

1. Use *Scatter Plot* and try *Find Informative Projections*. How does the top combination look like? Is it useful? Why (not)?
2. Use *Box Plot* and try *Order by relevance to subgroups*. Which attribute best splits the data by *Attrition*? Explain it.
3. Which widget would you use to observe relationship between two discrete variables in relation to Attrition? Use *Find informative projections* in that widget and explore the top result.

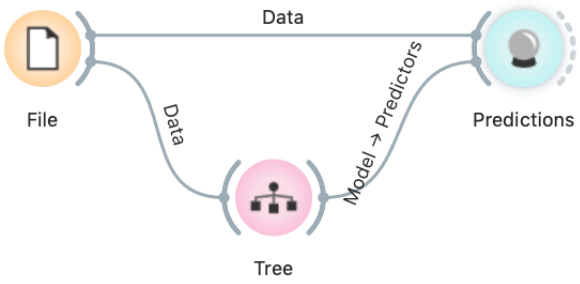
Classification

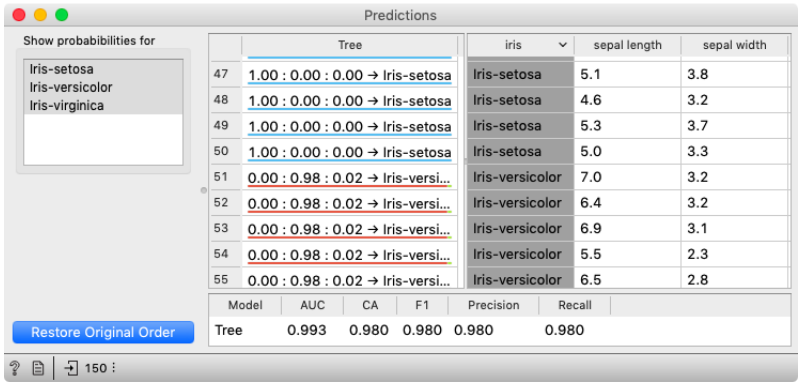
We call the variable we wish to predict a target variable, or an outcome or, in traditional machine learning terminology, a class. Hence we talk about classification, classifiers, classification trees...

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

Let us create one now.

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





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?

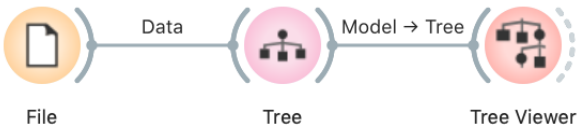
So many questions to answer!

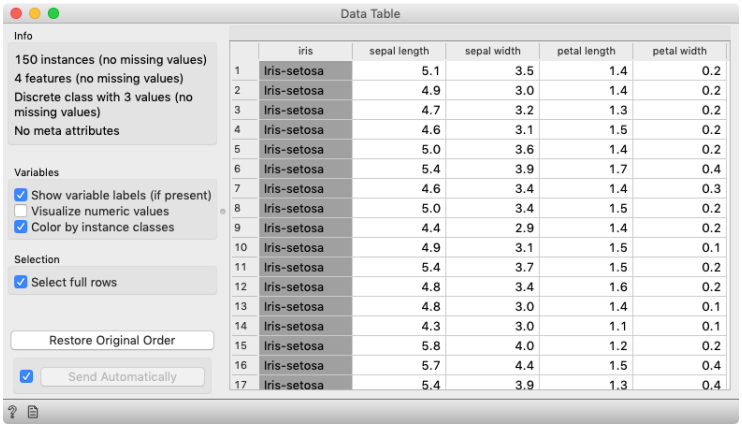
Classification Trees

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.

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

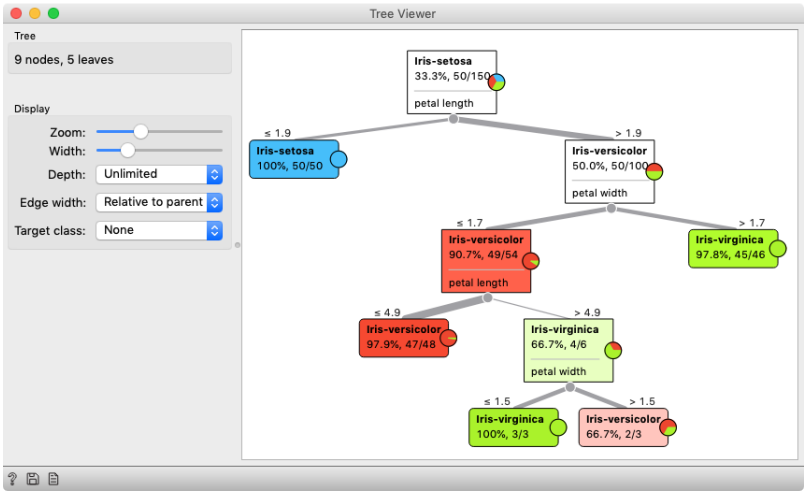
Let us load *iris* data set, build a tree (widget *Tree*) and visualize it in a *Tree Viewer*.





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

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



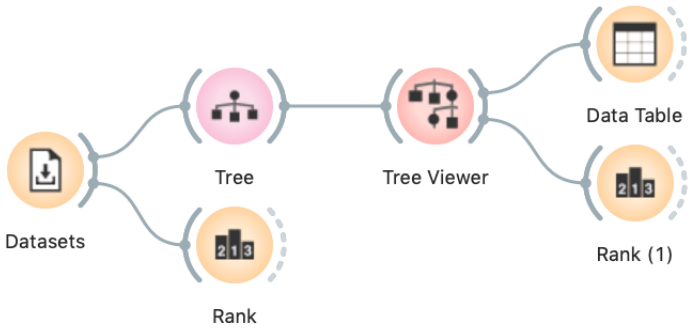
Trees place the most useful feature at the root. What would be the most useful feature? The feature that splits the data into two purest possible subsets. It then splits both subsets further, again by their most useful features, and keeps doing so until it reaches subsets in which all data belongs to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

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.

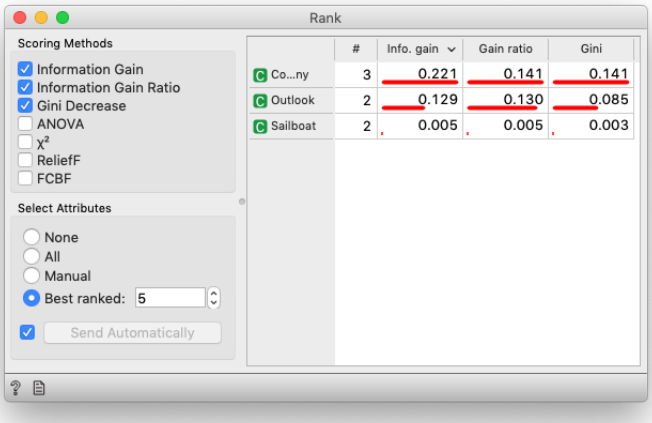
The *Rank* widget can be used on its own to show the best predicting features. Say, to figure out which genes are best predictors of the phenotype in some gene expression data set.

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

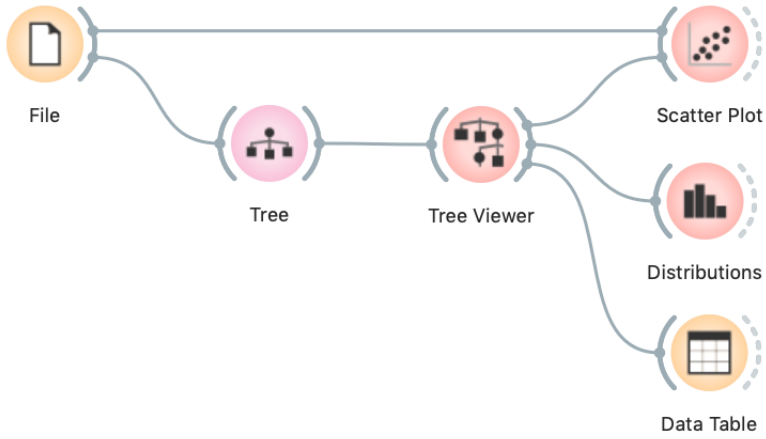


Besides the information gain, *Rank* displays several other measures (including Gain Ratio and Gini), which are often quite in agreement and were invented to better handle discrete features with many different values.

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

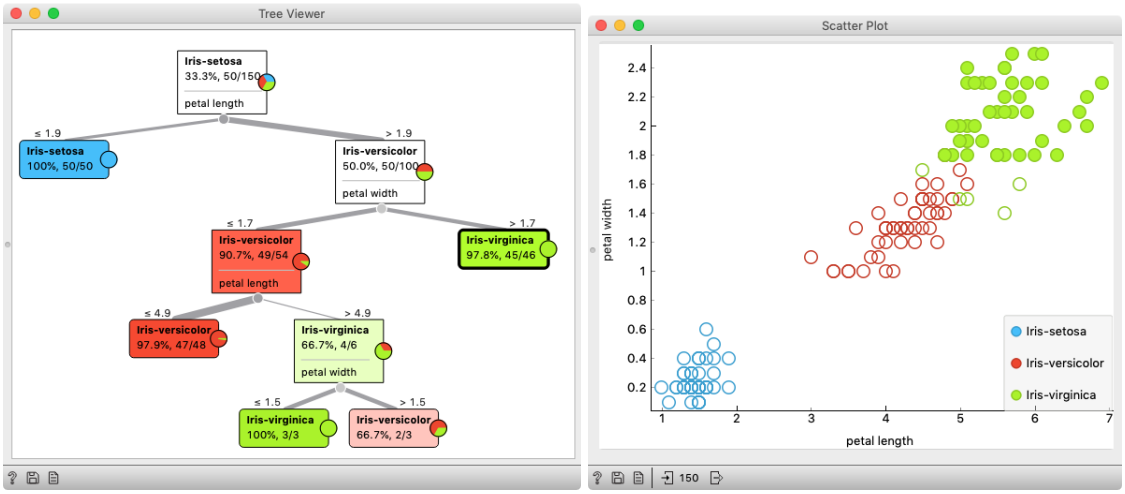


Here is an interesting combination of a *Tree Viewer* and a *Scatter Plot*. This time, use the *Iris* data set. In the *Scatter Plot*, we first find the best visualization of this data set, that is, the one that best separates the instances from different classes. Then we connect the *Tree Viewer* to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

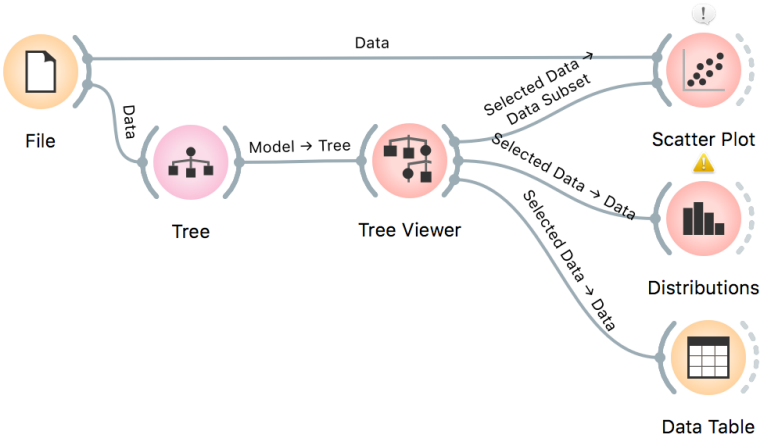


In the *Tree Viewer* we selected the rightmost node. All data instances coming to the selected node are highlighted in *Scatter Plot*.

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.

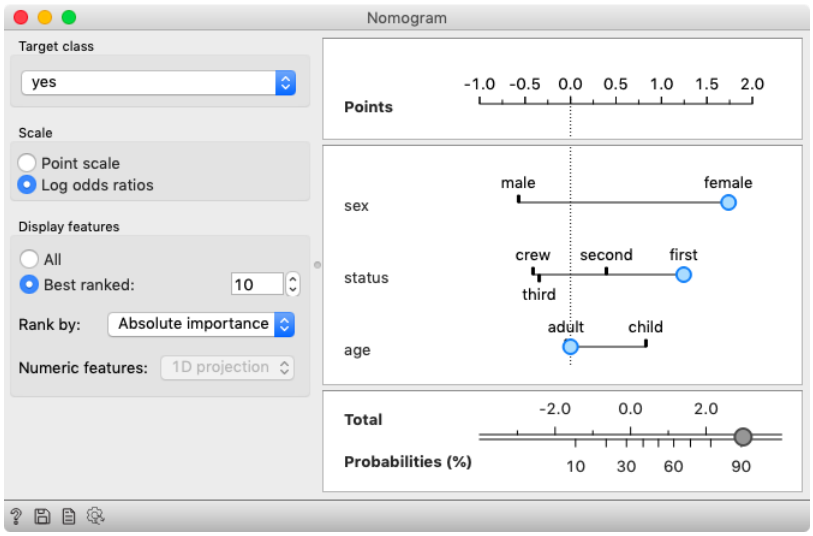
Naive Bayes

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.

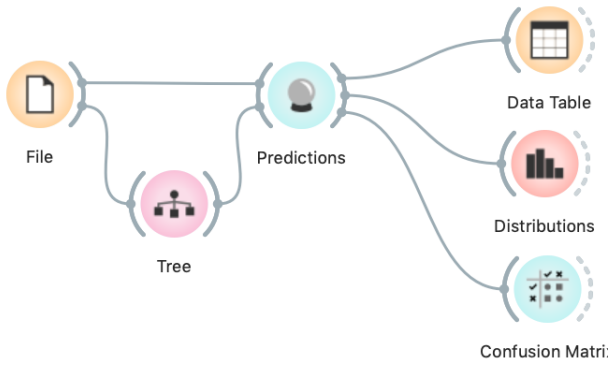


According to the probability theory individual contributions should be multiplied. Nomograms get around this by working in a log-space: a sum in the log-space is equivalent to multiplication in the original space. Therefore nomograms sum contributions (in the log-space) of all feature values and then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

Classification Accuracy

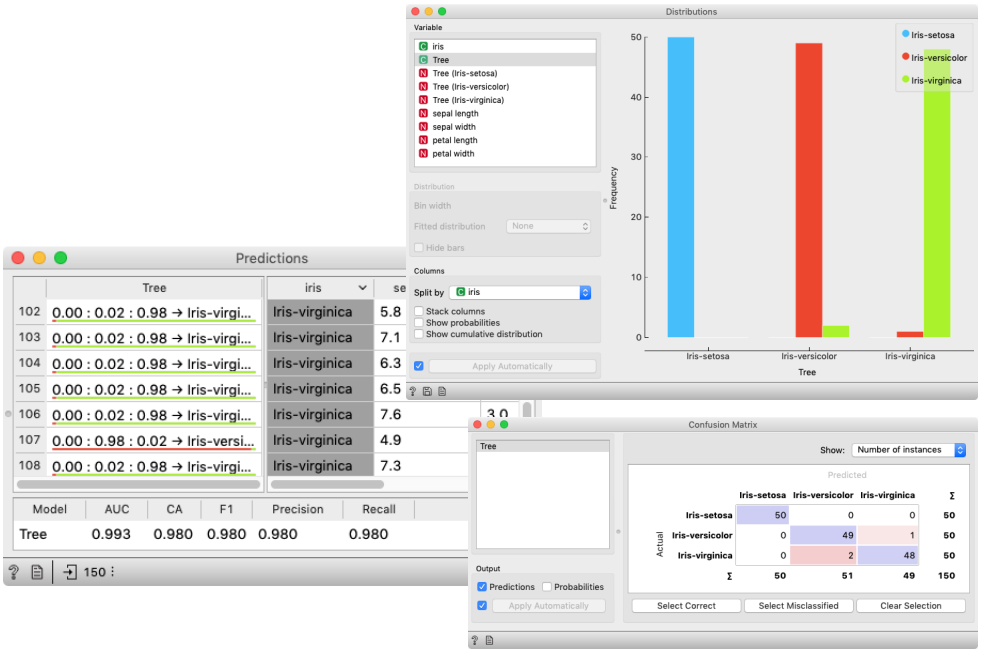
$$\text{accuracy} = \frac{\#\{\text{correct}\}}{\#\{\text{all}\}}$$

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



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.

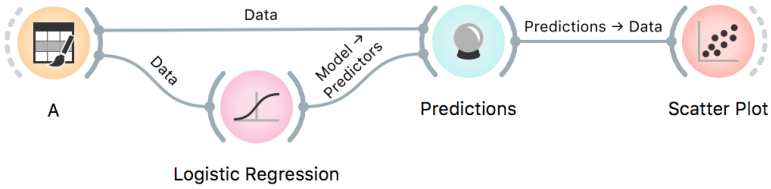
For precise statistics of correctly and incorrectly classified examples open the *Confusion Matrix* widget.



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

Assignment: Decision Boundaries

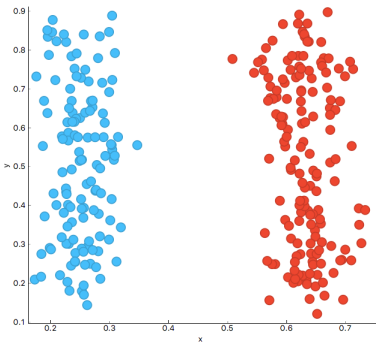
Classifiers come in all shapes and sizes. What we mean by that is that each has its own way of learning from the data, its own strengths and weaknesses. Knowing how classifiers work is crucial for selecting the right algorithm for your task.

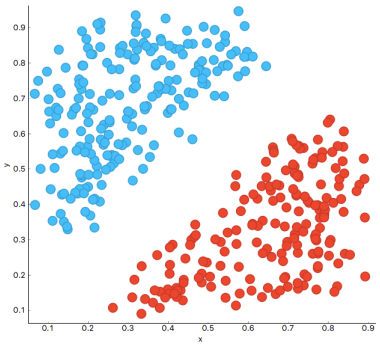


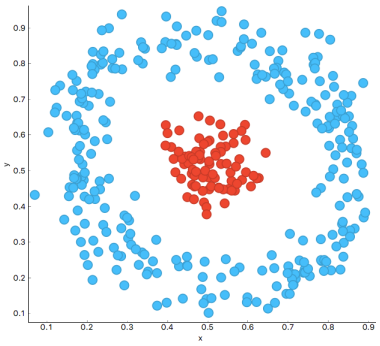
Try *Tree*, *Logistic Regression*, *SVM*, *Random Forest*, and *kNN* classifiers:

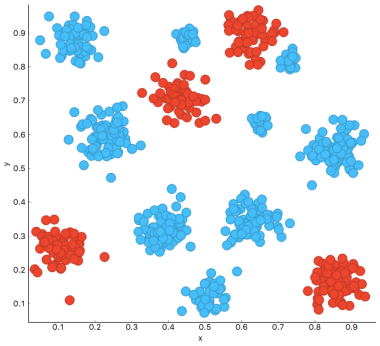
1. Which classifiers work well with data set A? Which with B, C, and D?
2. Which classifier is struggling the most? Which one the least? Why?
3. Look at the Tree with *Tree Viewer* for data set C. What do you notice?
4. In the above example, you can separate classes with a single stroke of a pen. Now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

You can try painting the data yourself or download it from here.





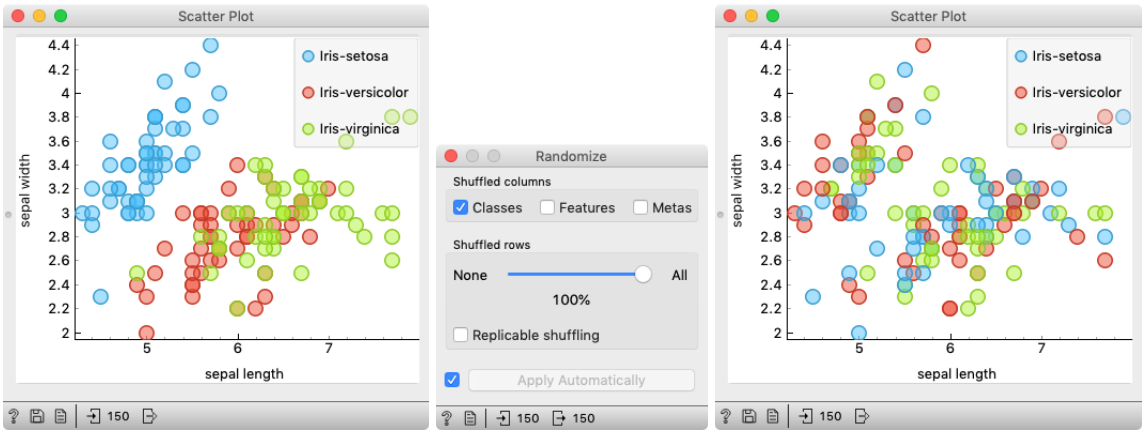




How to Cheat

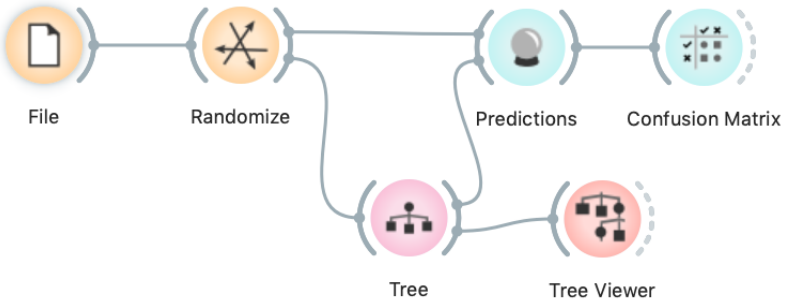
This lesson has a strange title and it is not obvious why it was chosen. Maybe you, the reader, should tell us what this lesson has to do with cheating.

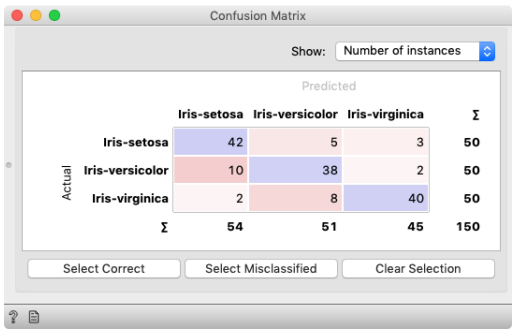
At this stage, the classification tree looks very good. There's only one data point where it makes a mistake. Can we mess up the data set so bad that the trees will ultimately fail? Like, remove any existing correlation between features and the class? We can! There's the *Randomize* widget with class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

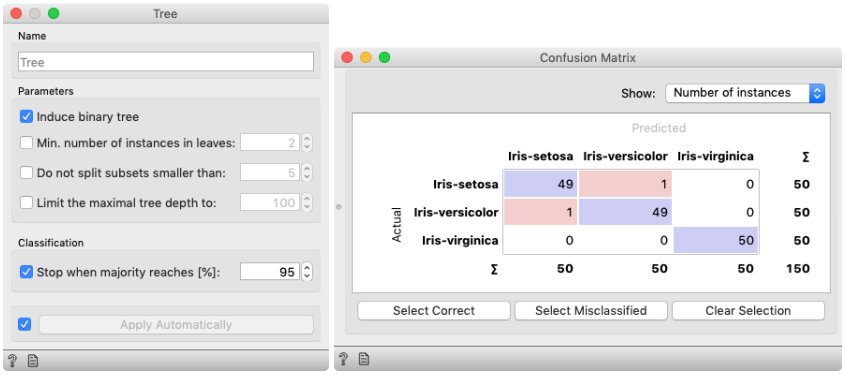




And the result? Here is a screenshot of the *Confusion Matrix*.

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.



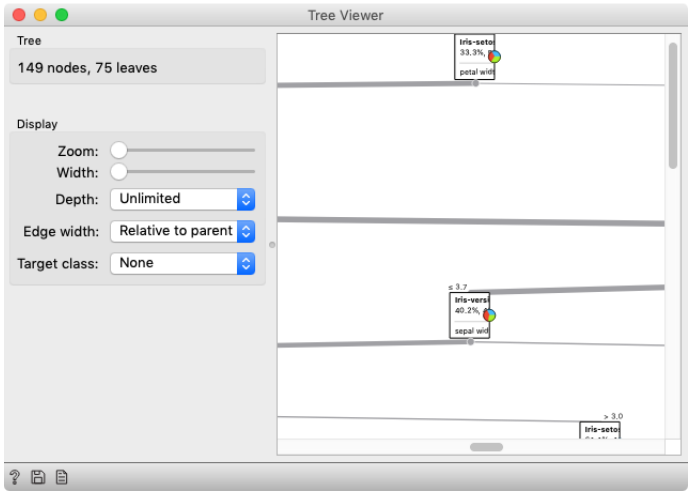
After we disable 2–4 check box in the *Tree* widget, our classifier starts behaving almost perfectly.

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

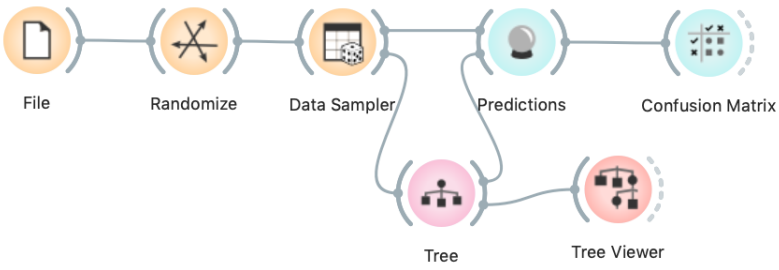
To find the answer to this riddle, open the *Tree Viewer* and check out the tree. How many nodes does it have? Are there many data instances in the leaf nodes?

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.



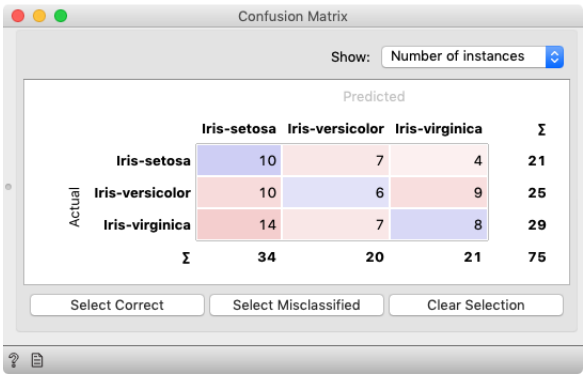
Connect the *Data Sampler* widget carefully. The *Data Sampler* splits the data to a sample and out-of-sample (so called remaining data). The sample was given to the *Tree* widget, while the remaining data was handed to the *Predictions* widget. Set the *Data Sampler* so that the size of these two data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

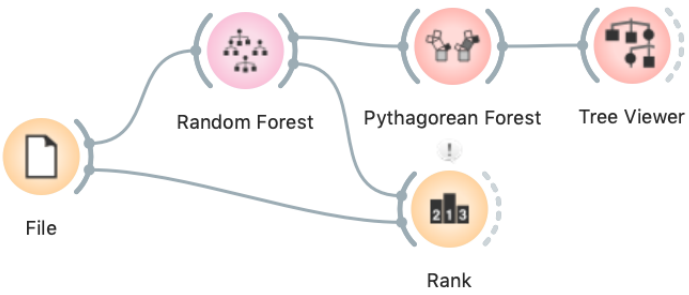


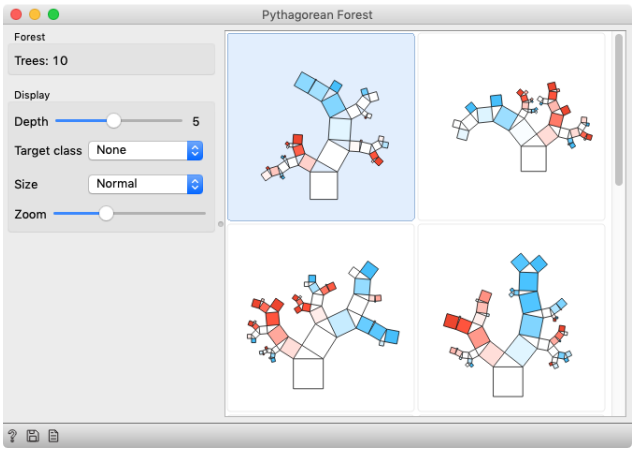
We have just learned that we need to train the classifiers on the training set and then test it on a separate test set to really measure performance of a classification technique. With this test, we can distinguish between those classifiers that just memorize the training data and those that actually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

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.

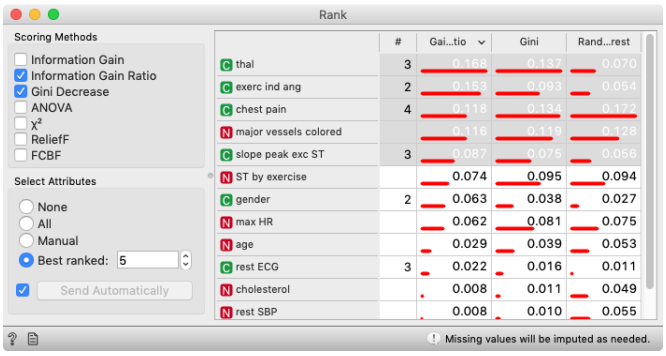




The *Pythagorean Forest* widget shows us how random the trees are. If we select a tree, we can observe it in a *Tree Viewer*.

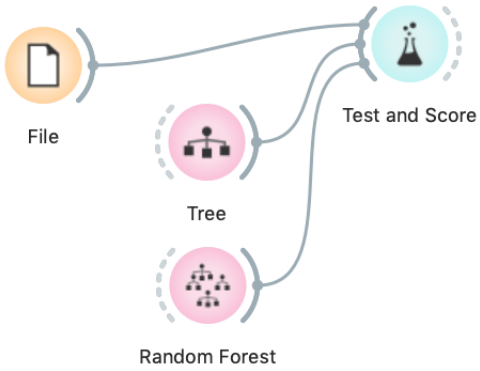
There are two sources of randomness: (1) training data is sampled with replacement, and (2) the best feature for a split is chosen among a subset of randomly chosen features.

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.

Cross-Validation



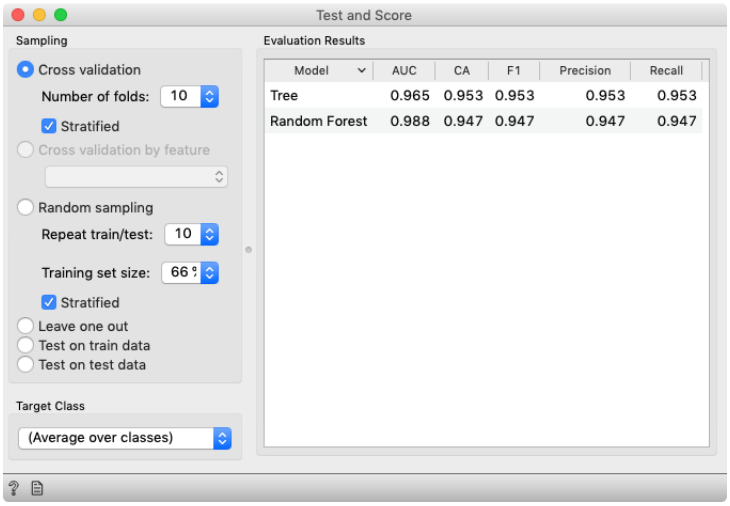
Estimating the accuracy may depend on a particular split of the data set. To increase robustness, we can repeat the measurement several times, each time choosing a different subset of the data for training. One such method is cross-validation. It is available in Orange in the *Test and Score* widget.

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.



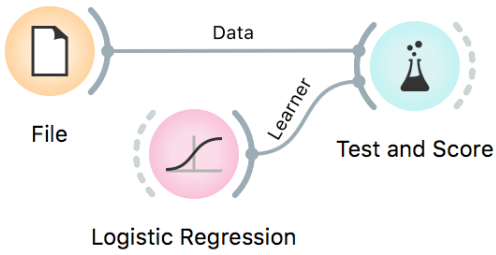
In the *Test and Score* widget, the second column, CA, stands for classification accuracy, and this is what we really care for for now.

Assignment: Overfitting

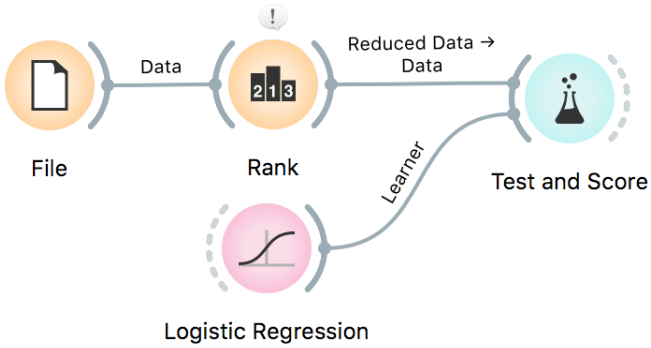
Overfitting is something we try to avoid at all times. But overfitting comes in many shapes and sizes. For this exercise we will use a *blood-loneliness* data set with the File widget. This data set relates gene expressions in blood with a measure of loneliness obtained from a sample of elderly persons. Let's try to model loneliness with logistic regression and see how well the model performs.

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

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

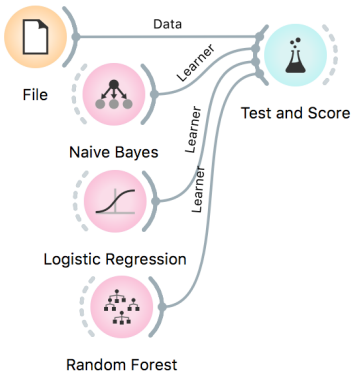


1. Train the Logistic Regression model on the data and observe its performance. What is the result?
2. We have many features in our data. What if we select only the most relevant ones, the genes that actually matter? Use Rank to select the top 20 best performing features.



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

A Few More Classifiers

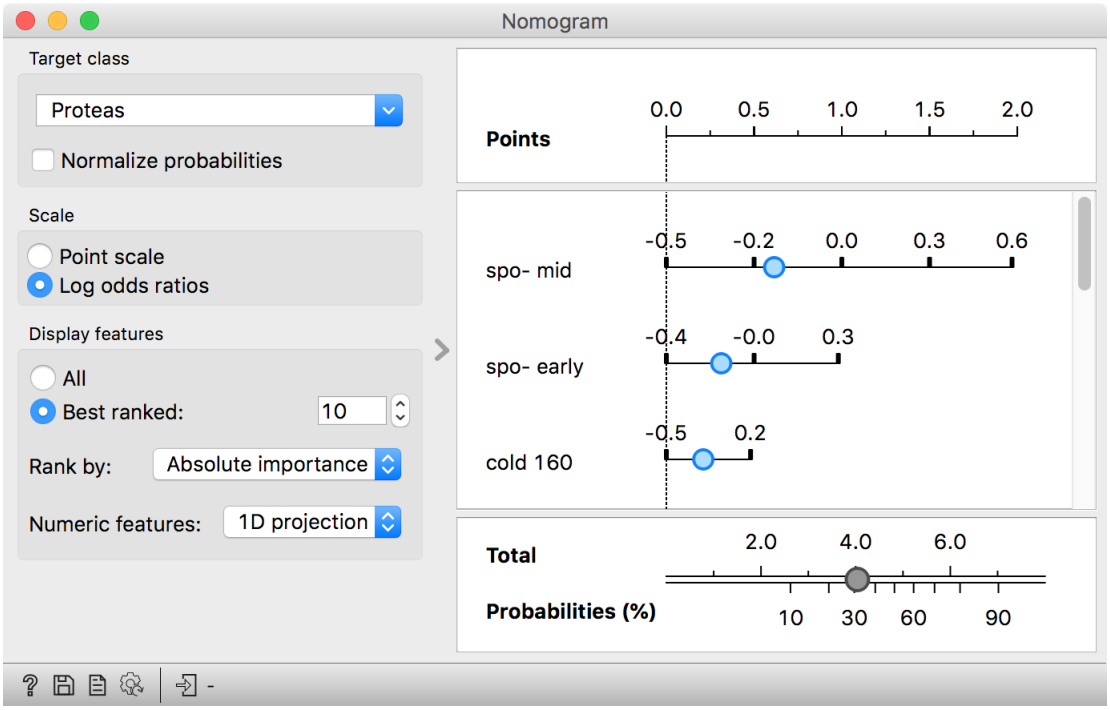


We have ended the previous lesson with cross-validation and classification trees. There are many other, much more accurate classifiers. A particularly interesting one is Random Forest, which averages across predictions of hundreds of classification trees. It uses two tricks to construct different classification trees. First, it infers each tree from a sample of the training data set (with replacement). Second, instead of choosing the most informative feature for each split, it randomly selects from a subset of most informative features. In this way, it randomizes the tree inference process. Think of each tree shedding light on the data from a different perspective. Just like in the wisdom of the crowd, an ensemble of trees (called a forest) usually performs better than a single tree.

Another popular classifier is logistic regression. In this model, each variable has its weight or importance. Logistic regression computes weights of each variable during the training phase. For prediction, it simply multiplies the weight of the variable with its value, computes the total sum and log transforms it into probability.

We can use Nomogram to observe the importance of variables in a model and their weights. The variables in the plot are ranked and the length of the line corresponds to their importance in the model.

Logistic regression is a classification, not a regression method. It is called regression because it is methodologically similar to linear regression.



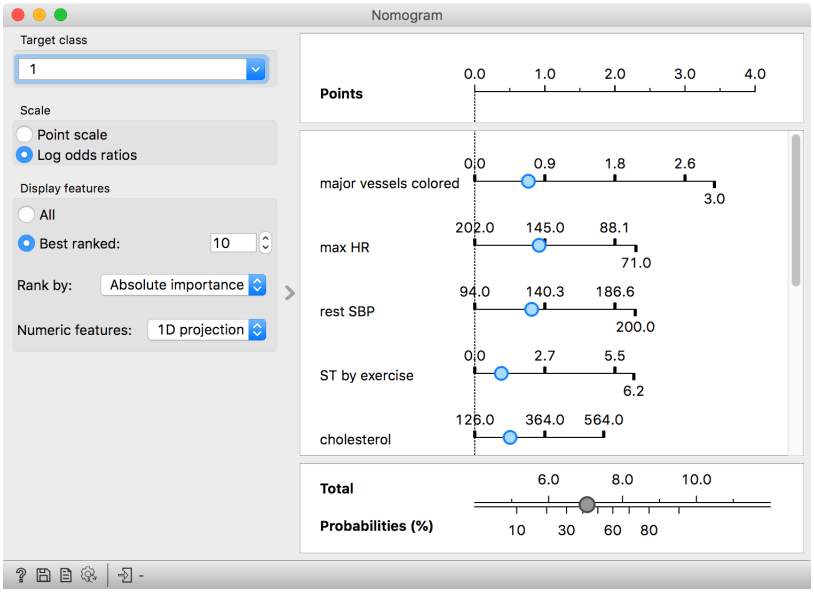
Logistic Regression

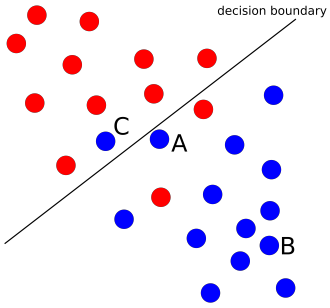
Logistic regression is one of the best-known classifiers. The model returns the probability of a class variable, based on input features. First, it computes probabilities with a one-versus-all approach, meaning that for a multiclass problem, it will take one target value and treat all the rest as "other",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

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

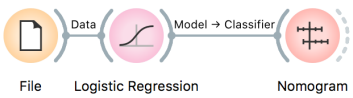
Logistic regression tries to find such a plane that all points from one class are as far away from the boundary (in the correct direction) as possible.

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





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



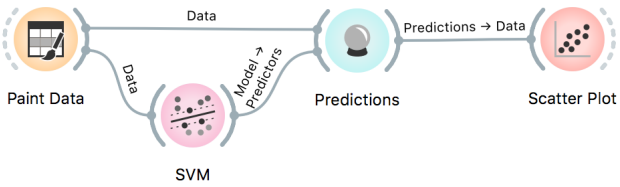
Another characteristic of logistic regression is that it observes all variables at once and takes the correlation into account. If some variables are correlated, their importance will be spread among them.

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

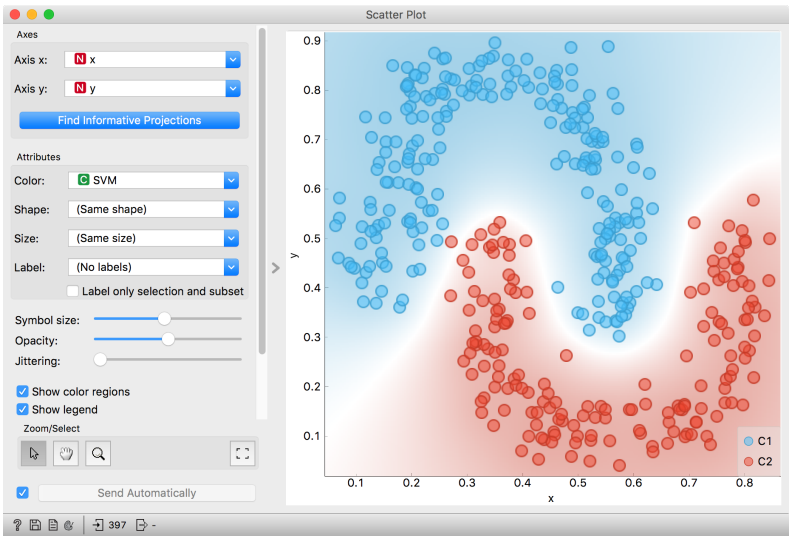
Support Vector Machines

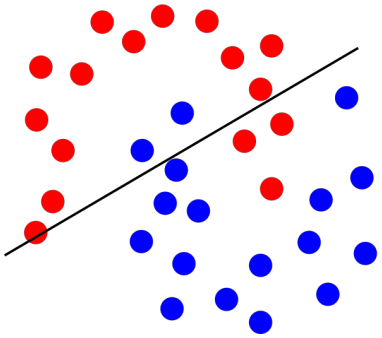
Support vector machines (SVM) are another example of linear classifiers, similar to logistic or linear regression. However, SVM can overcome splitting the data by a plane by using the so-called *kernel trick*. This means the hyperplane (decision boundary) can be transformed to a higher-dimensional space,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

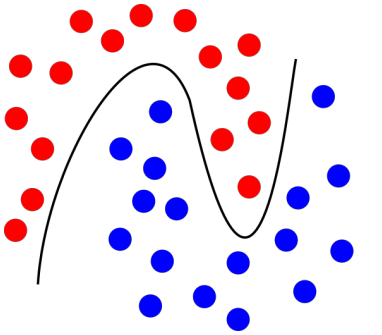


Abstract talking aside, SVM with different kernels can split the data not by ordinary hyperplanes, but with more complex curves. The complexity of the curve is decided by the kernel type and by the arguments given to the algorithm, like the degree and coefficients, and the penalty for misclassifications.



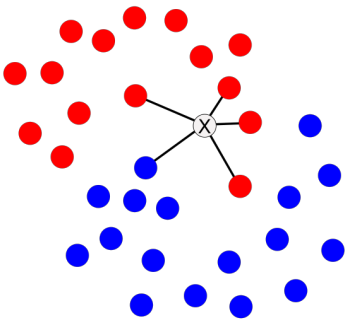


Decision boundary of a linear regression classifier.



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

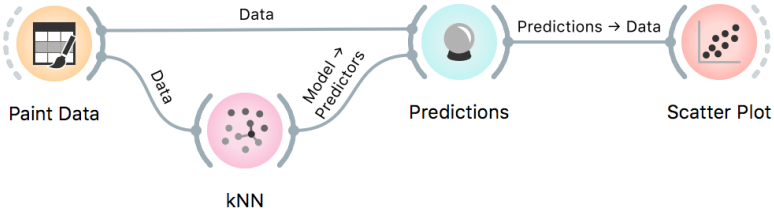
k-Nearest Neighbors



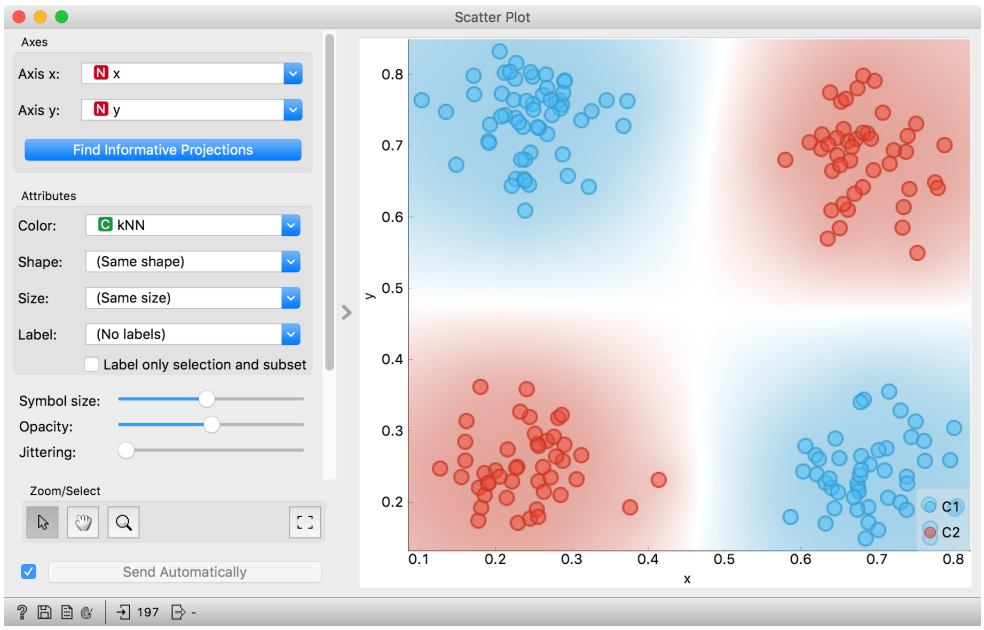
kNN classifier looks at k nearest neighbors, say 5, of instance X. 4 neighbors belong to the red class and 1 to the blue class. X will thus be classified as red with 80% probability.

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*.



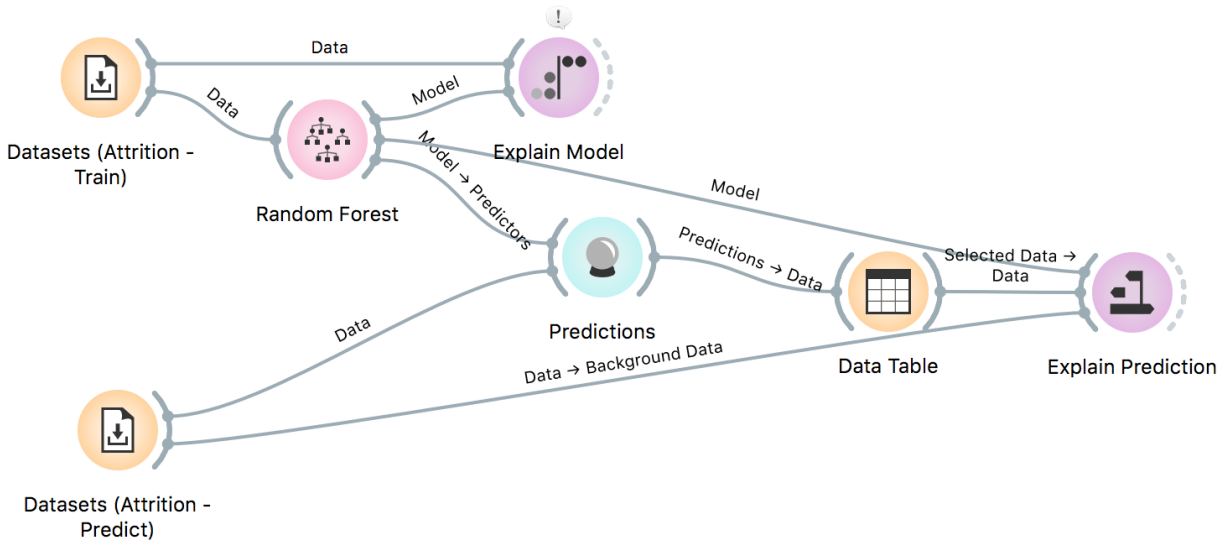
The advantage of kNN algorithm is that it can successfully model the data, where classes are not linearly separably. It can also be re-trained quickly, because new data instances effect model only locally. However, the first training is can be slow for large data sets, as the model has to estimate k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



Assignment: Model Explanation

Understanding the model is essential for decision-making. We will be using *Attrition - Train* data from the *Datasets* widget. We already know the data and its properties. But for reaching any kind of decisions regarding employee attrition, it is important not only to evaluate the models, but to understand them - what they do, which features are important, and in what way.

For this assignment, we will be using Explain add-on, which you can install in Options – Add-ons.

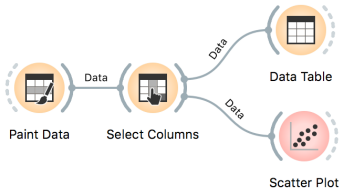


Use *Random Forest* to train a model, then answer the following questions:

1. Looking at *Explain Model*, which are the top three features for the random forest model?
2. Change the target class to *Yes*. What happens?
3. Two of the top three features are the same. Why?
4. In *Data Table*, select the employee for whom you wish to explain the prediction. Say, we go with John. Will John likely stay with the company or resign? Look at *Predictions* for an answer.
5. In *Explain Predictions*, explain *why* John is leaving or staying.

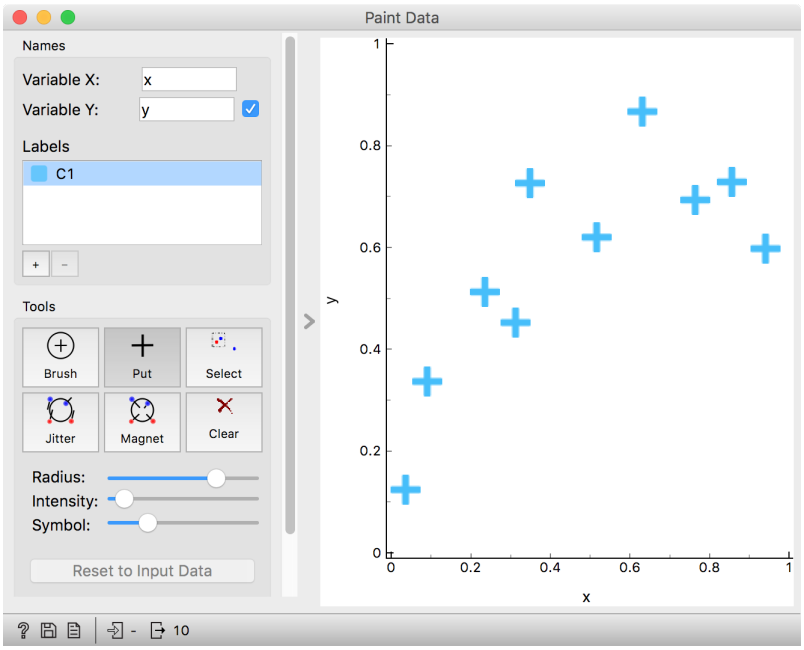
Linear Regression

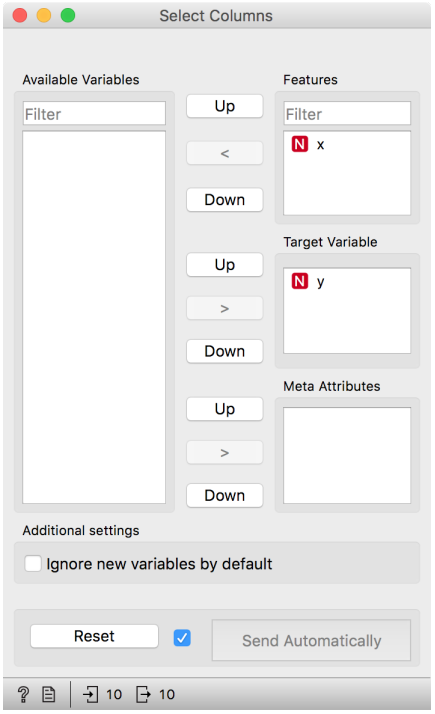
In the *Paint Data* widget, remove the C2 label from the list. If you have accidentally left it while painting, don't despair. The class variable will appear in the *Select Columns* widget, but you can "remove" it by dragging it into the Available Variables list.



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?

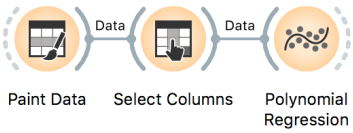


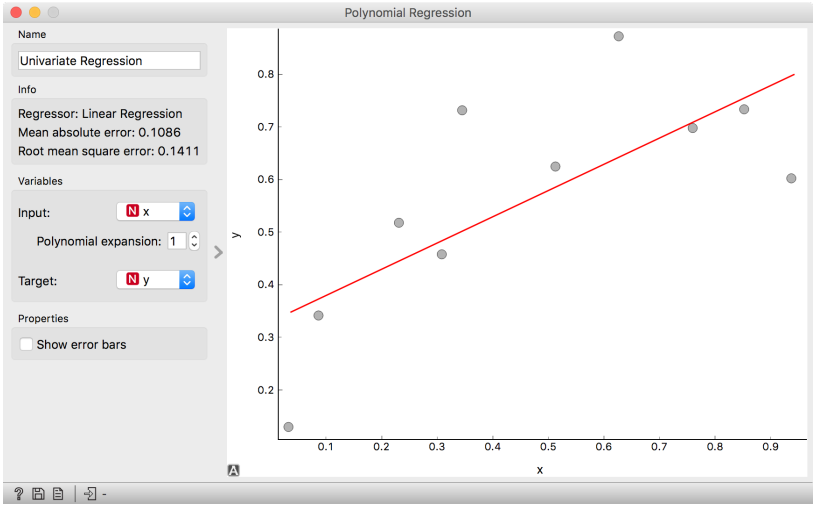


The question above requires us to define what a good fit is. Say, this could be the error the fitted model (the line) makes when it predicts the value of y for a given data point (value of x). The prediction is h(x), so the error is $h(x) - y$. We should treat the negative and

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*.

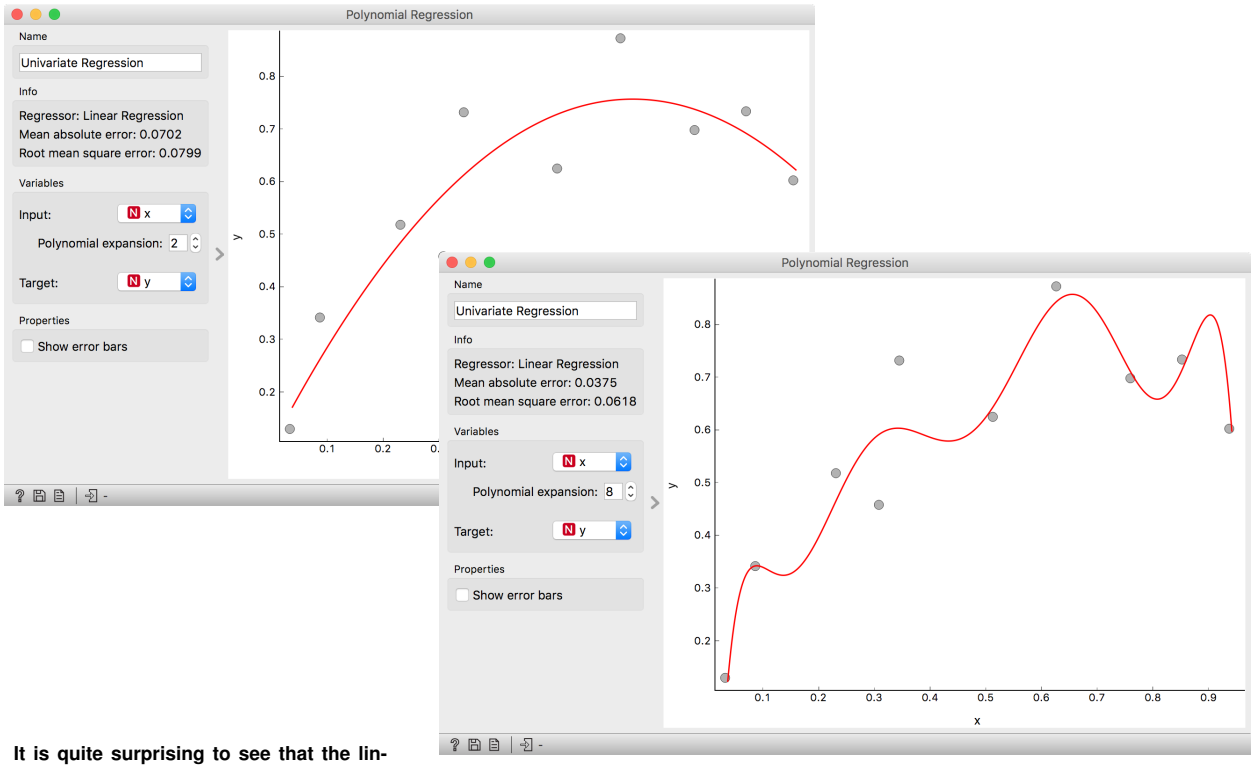
Do not worry about the strange name of the *Polynomial Regression*, we will get there in a moment.





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.

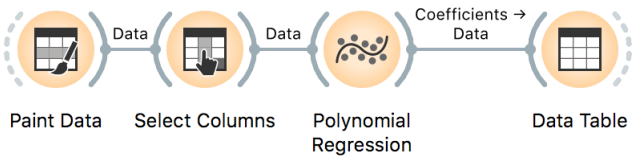


It is quite surprising to see that the linear regression model can fit non-linear (univariate) functions. It can fit the data with curves, such as those on the figures. How is this possible? Notice, though, that the model is a hyperplane (a flat surface) in the space of many features (columns) that are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

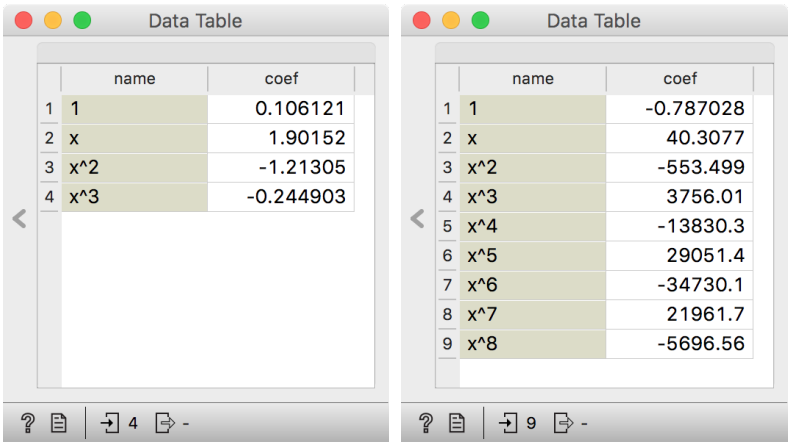
Overfitting is related to the complexity of the model. In polynomial regression, the parameters θ define the model. With the increased number of parameters, the model complexity increases. The simplest model has just one parameter (an intercept), ordinary linear regression has two (an intercept and a 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.

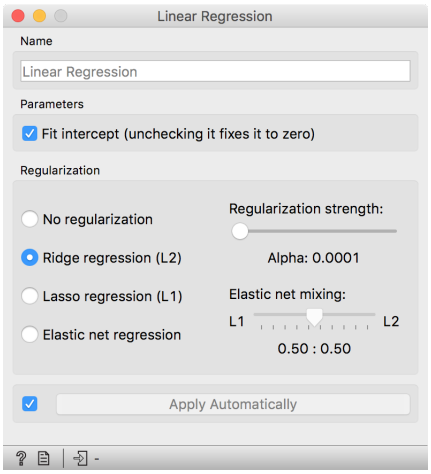


More complex models can fit the training data better. The fitted curve can wiggle sharply. The derivatives of such functions are high, so the coefficients θ need be. If only we could force the linear regression to infer models with a small value of coefficients. Oh, but we can. Remember, we have started 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.

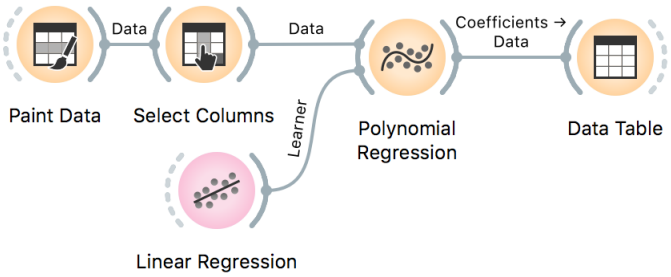
Here we go: we just reinvented regularization, which helps machine learning models not overfit the training data. To observe the effects of regularization, we can give *Polynomial Regression* to our linear regression learner, which supports these settings.

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?

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.

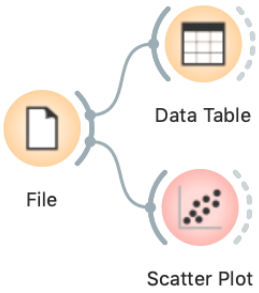


Now for the test. Increase the degree of polynomial to the max. Use Ridge Regression. Does the inferred model overfit the data? How does the degree of overfitting depend on regularization strength?

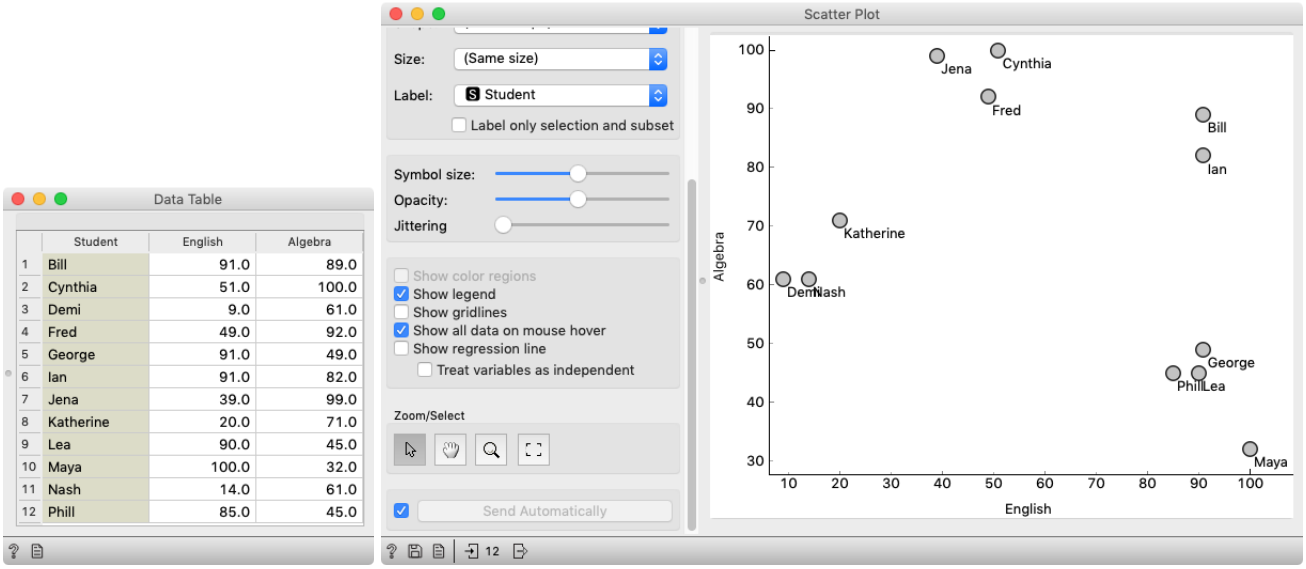
Hierarchical Clustering

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.

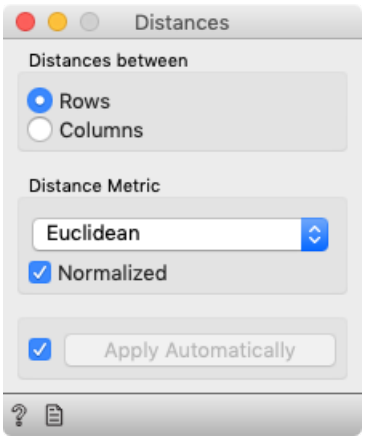


We will introduce clustering with a simple data set on students and their grades in English and Algebra. Load the data set from `http://file.biolab.si/text/grades.tab`.

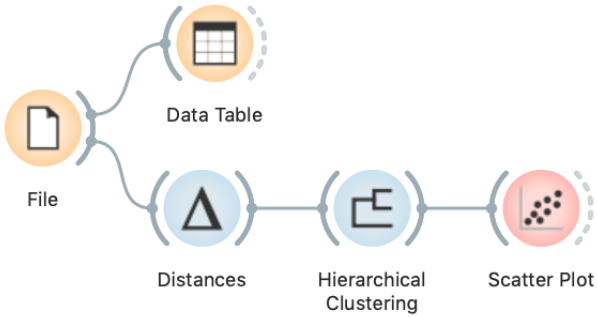


Next, we need to define a clustering algorithm. Say that we start with each data instance being its cluster, and then, at each step, we join the closest clusters. We estimate the distance between the clusters with the average distance between all their pairs of data points. This algorithm is called hierarchical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

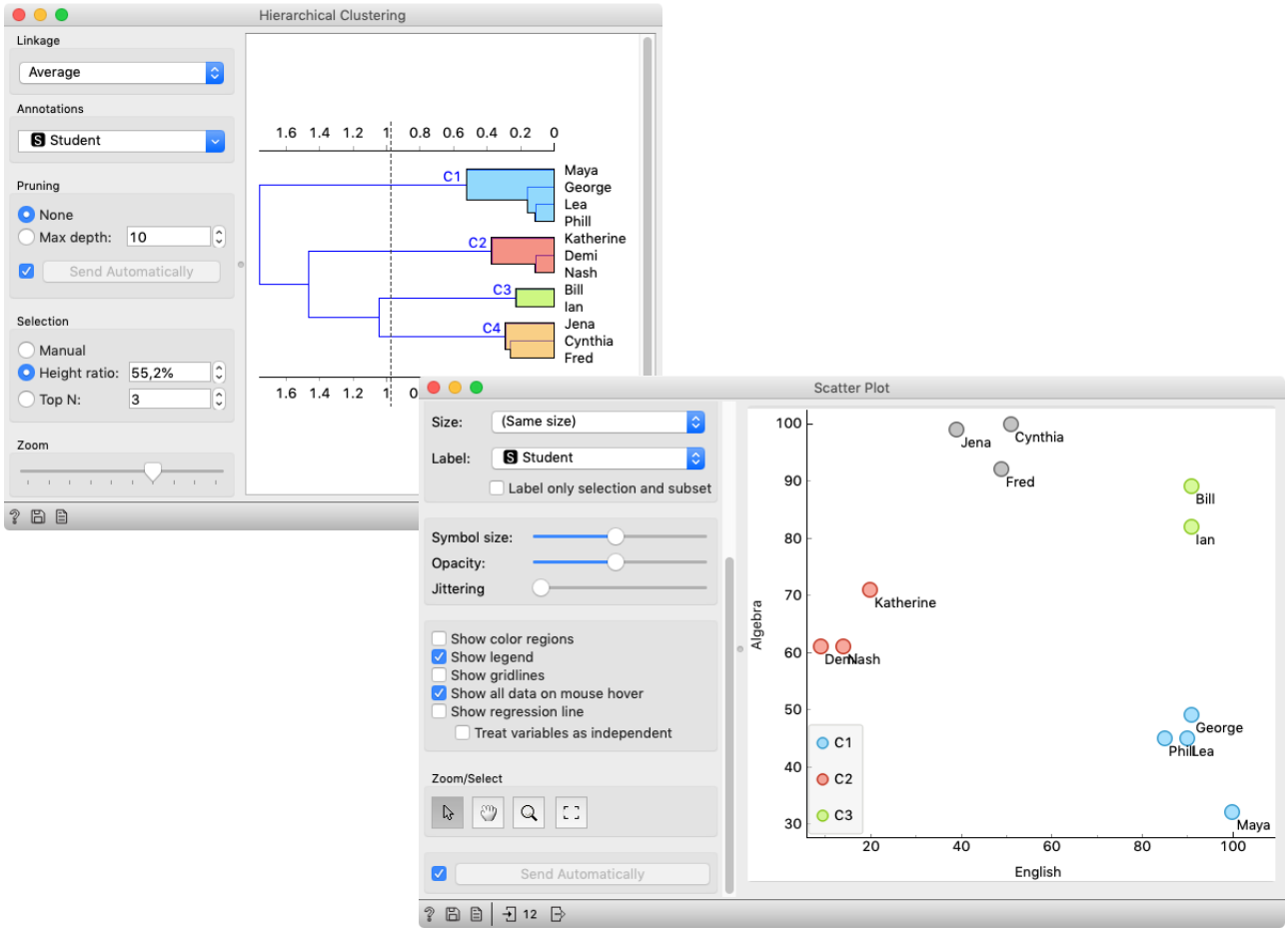
There are different ways to measure the similarity between clusters. The estimate we have described is called average linkage. We could also estimate the distance through the two closest points in each group (single linkage) or through the two points that are furthest away (complete linkage).



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:

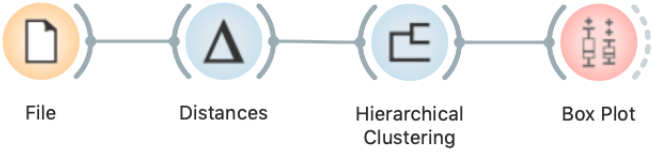


Animal Kingdom

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.

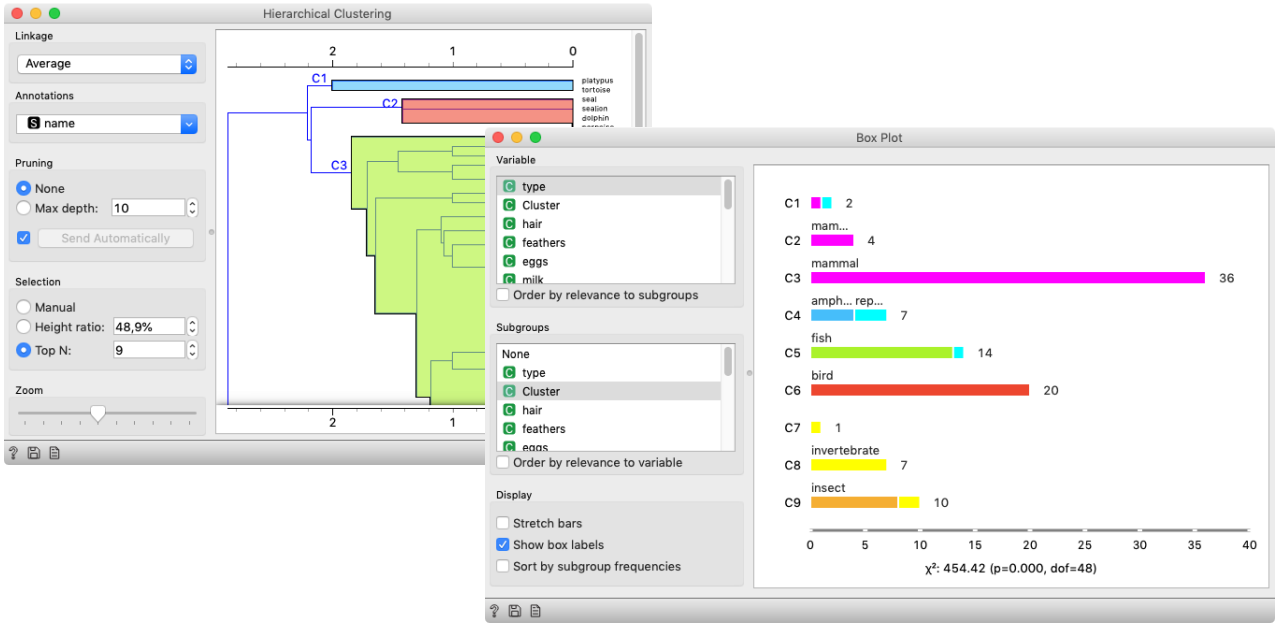
Funny stuff was we never understood the order in which the cards were laid out in the album. We later learned about taxonomy, but being more inclined to engineering we never mastered learning it in our biology classes. Luckily, there's data mining and the idea that taxonomy simply stems from measuring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

k-Means Clustering

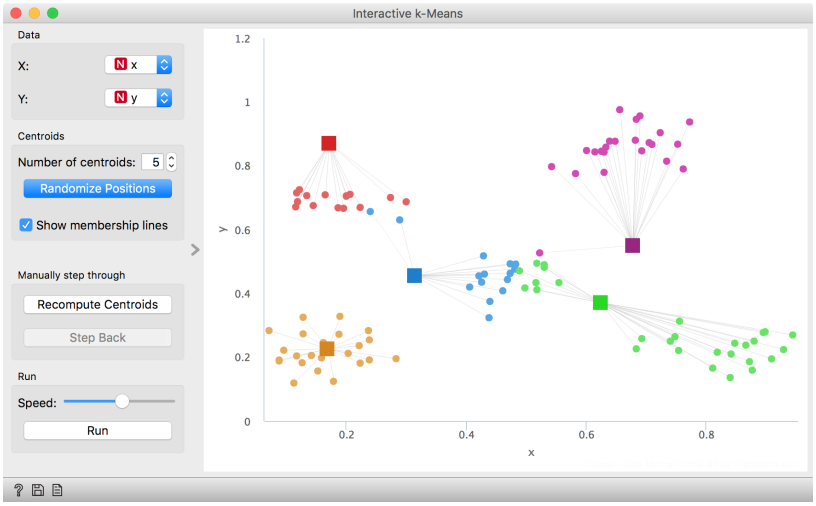
Hierarchical clustering is not suitable for larger data sets due to the prohibitive size of the distance matrix: with 30 thousand objects, the distance matrix already has almost one billion elements. An alternative approach that avoids using the distance matrix is k-means clustering.

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.

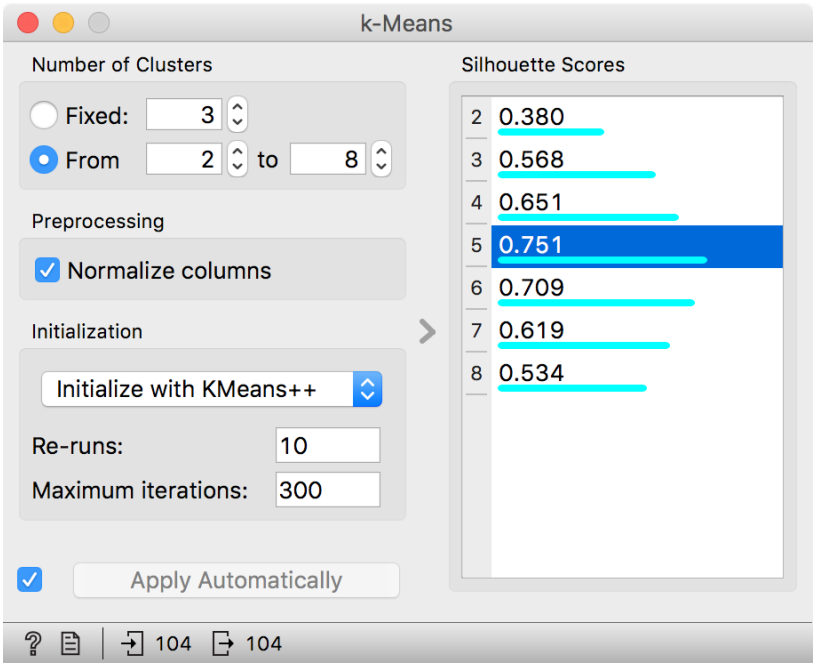
How do we set the initial number of clusters? That's simple: we choose the number that gives the optimal clustering.

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.

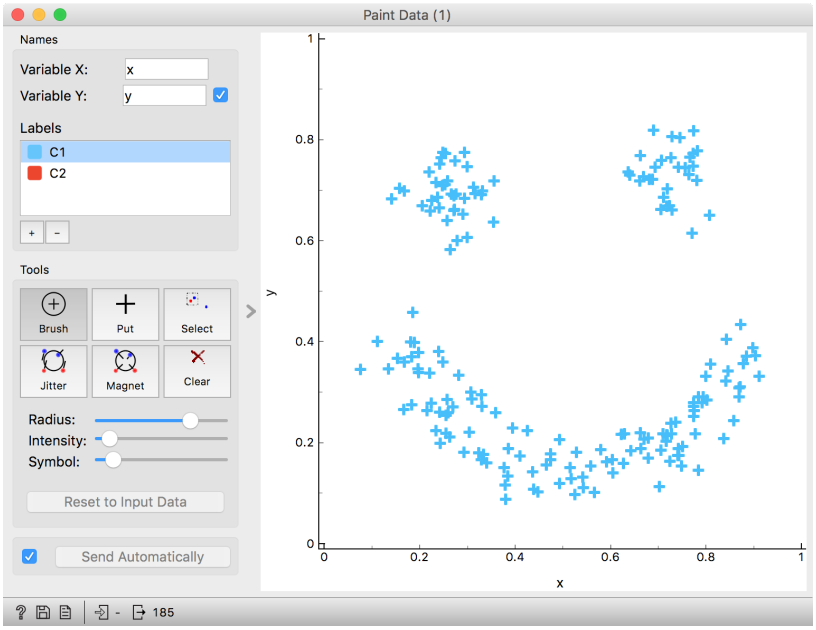
For this, we abandon our educational toy and connect Paint Data to the widget k-Means. We tell it to find the optimal number of clusters between 2 and 8, as scored by the Silhouette.



Works like a charm.

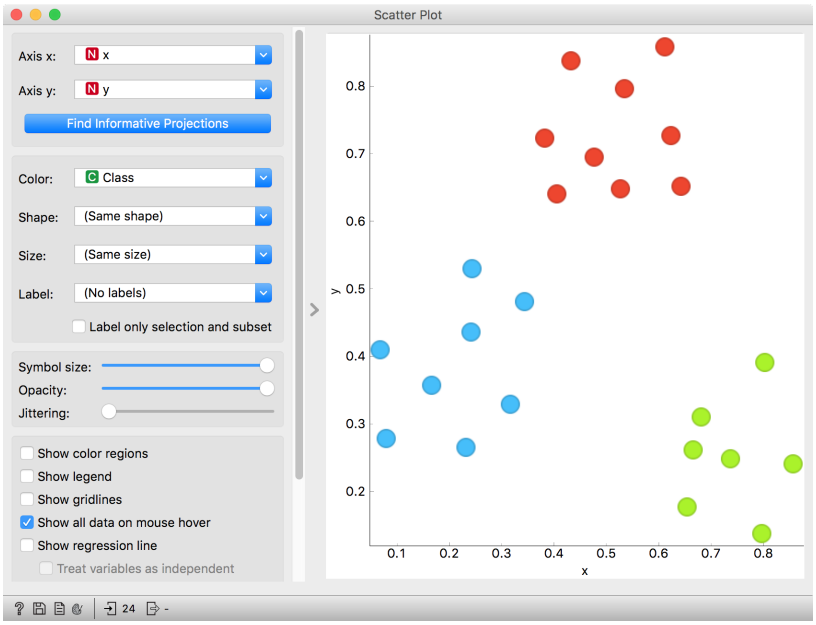
Except that it often doesn't. First, the result of k-means clustering depends on the initial selection of centers. With unfortunate selection, it may get stuck in a local optimum. We solve this by re-running the clustering multiple times from random positions and using the best result. Second, the silhouette 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?

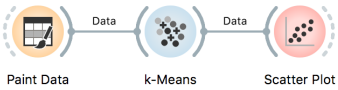


Silhouettes

Consider a two-feature data set which we have painted in the *Paint Data* widget. We send it to the k-means clustering, tell it to find three clusters, and display the clustering in the scatter plot.



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

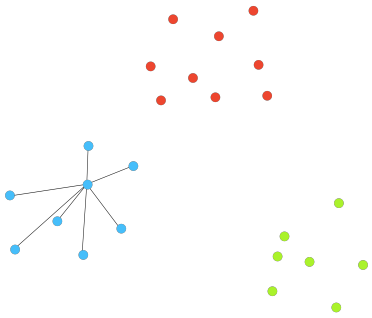


The data points in the green cluster are well separated from those in the other two. Not so for the blue and red points, where several points are on the border between the clusters. We would like to quantify the degree of how well a data point belongs to the cluster to which it is assigned.

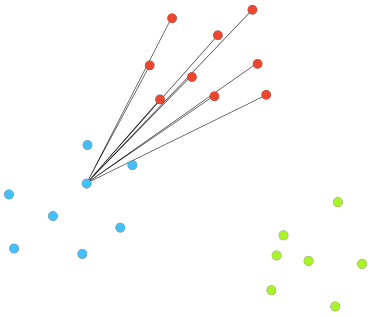
We will invent a scoring measure for this and we will call it a silhouette (because this is how it's called). Our goal: a silhouette of 1 (one) will mean that the data instance is well rooted in the cluster, while the score of 0 (zero) will be assigned to data instances on the border between two clusters.

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

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



Average distance A.



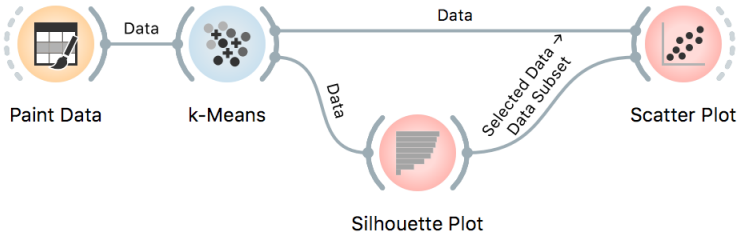
Average distance B.

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

The larger the B, the better.

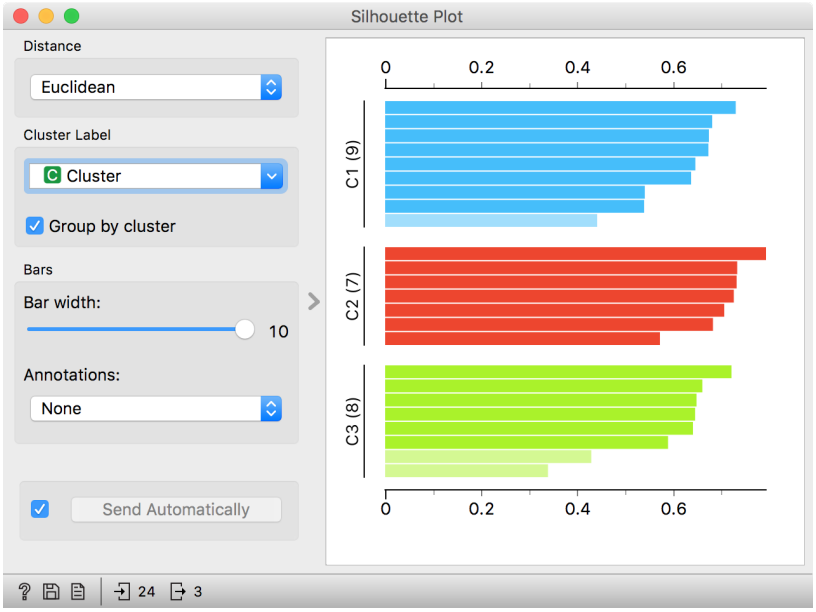
The point is well rooted within its own cluster if the distance to the points from the neighboring cluster (B) is much larger than the distance to the points from its own cluster (A), hence we compute B-A. We normalize it by dividing it with the larger of these two numbers, S = (B -A) / maxA, B. Voilá,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



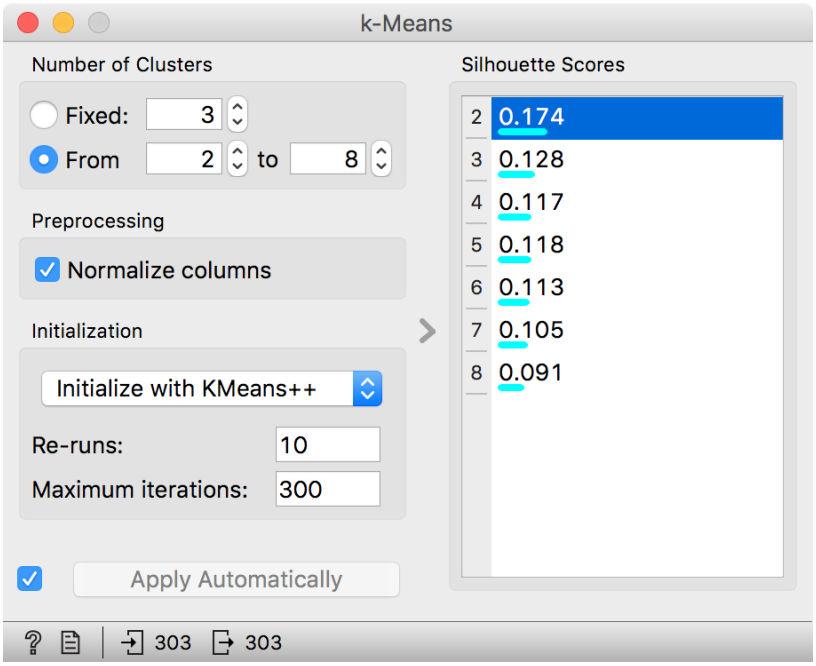
This of course looks great for data sets with two features, where the scatter plot reveals all the information. In higher-dimensional data, the scatter plot shows just two features at a time, so two points that seem close in the scatter plot may be actually far apart when all features - perhaps thousands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

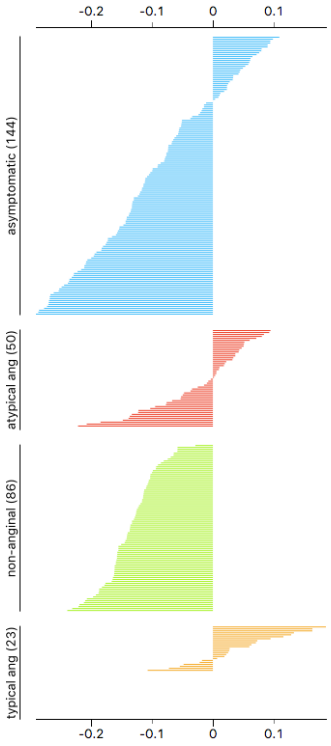
We selected three data instances with the worst silhouette scores. Can you guess where they lie in the scatter plot?



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.

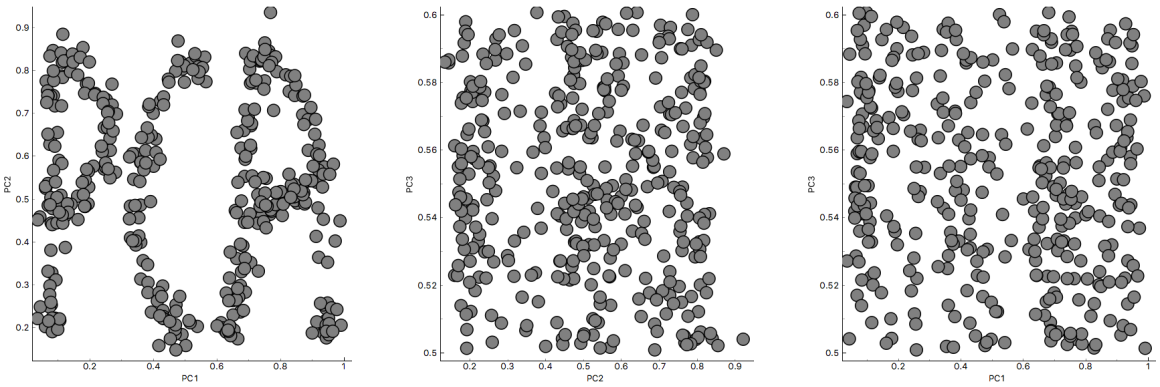
We don't have to group the instances by the class. For instance, the silhouette on the left would suggest that the patients from the heart disease data with typical anginal pain are similar to each other (with respect to the distance/similarity computed from all features), while those with other types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





Principal Component Analysis

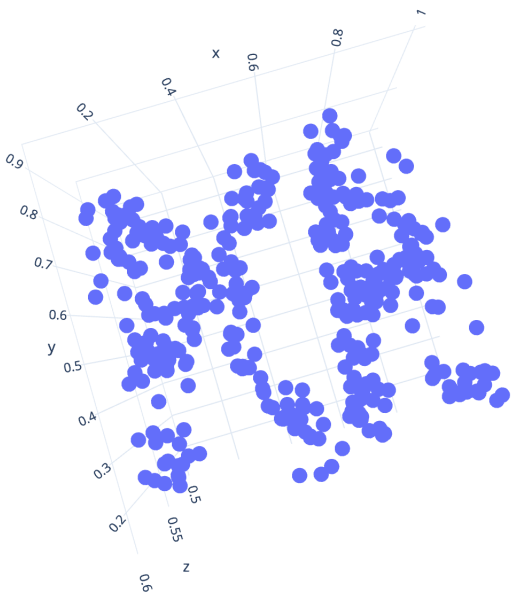
Which of the following three scatter plots (showing x vs. y, x vs. z and y vs. z) for the same three-dimensional data gives us the best picture about the actual layout of the data in space?



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.

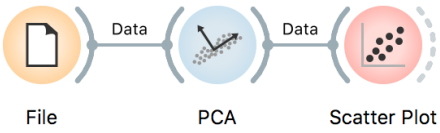
After some careful observation, you notice the flies are a bit spread in one other direction, so they do not fly along a line but along a band. Therefore, we need two numbers, one along the first and one along the — you guessed it — second principal component.

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

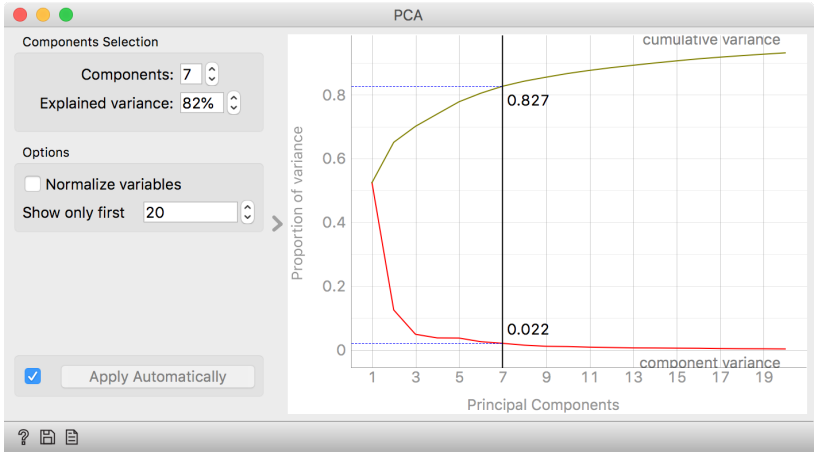
Or do you? It all depends on how spread they are in the second and in the third direction. If the spread along the second is relatively small in comparison with the first, you are fine with a single dimension. If not, you need two, but perhaps still not three.

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.

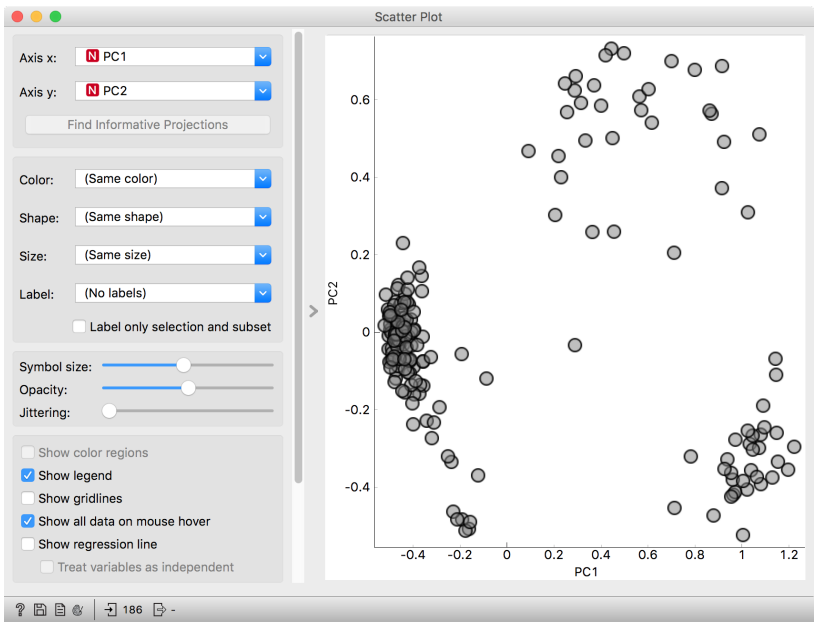
A technique that does this is called Principle Components Analysis, or *PCA*. The corresponding widget is simple: it receives the data and outputs the transformed data.



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.

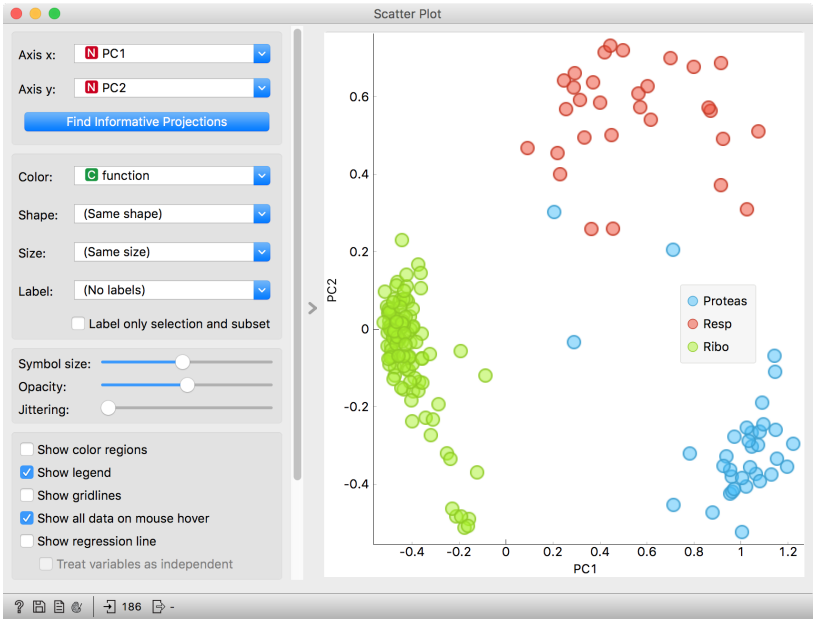


The PCA on the left shows the scree diagram for brown-selected data. Set like this, the widget replaces the 80 features with just seven - and still keeping 82.7% of information. (Note: disable "Normalize data" checkbox to get the same picture.) Let us see a scatter plot for the first two components.

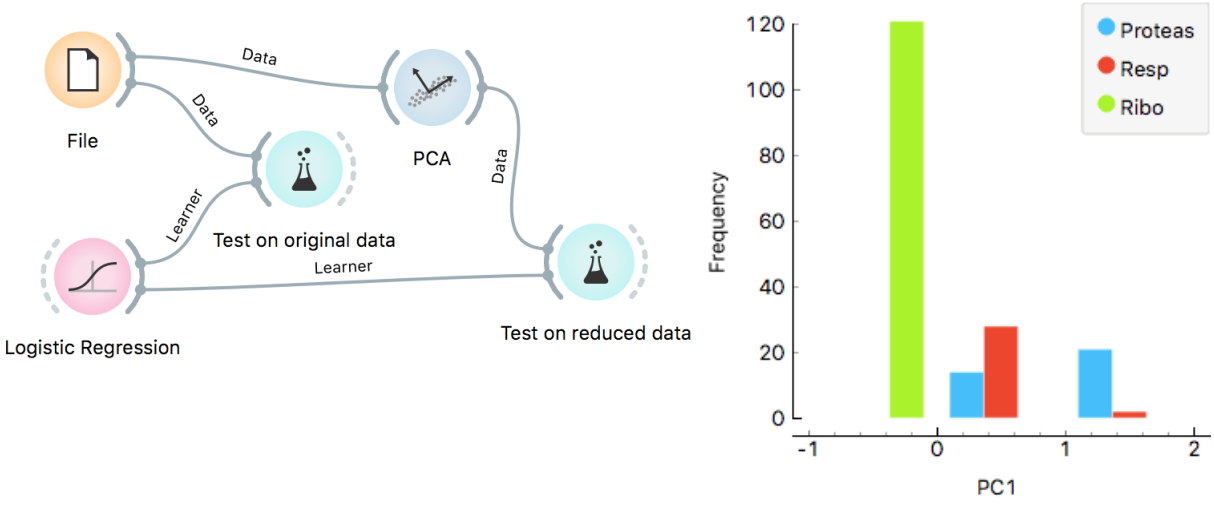


The axes, PC1 and PC2, do not correspond to particular features in the original data, but to their linear combination. What we are looking at is a projection onto the plane, defined by the first two components. When you consider only two components, you can imagine that PCA puts a hyperplane into multidimensional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

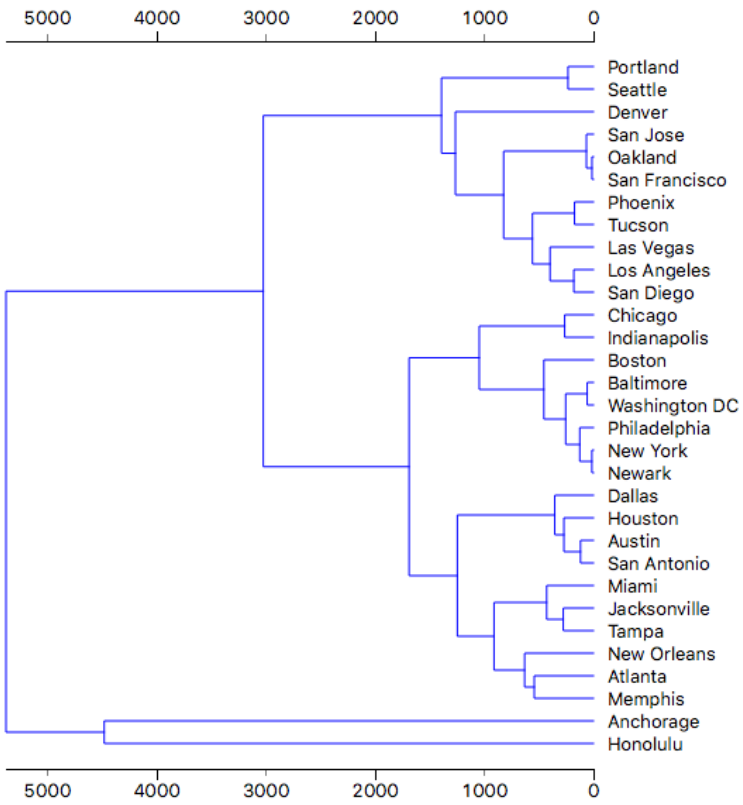
On the original data, logistic regression gets 98% AUC and classification accuracy. If we select just a single component in PCA, we already get a 93%, and if we take two, we get the same result as on the original data.

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.

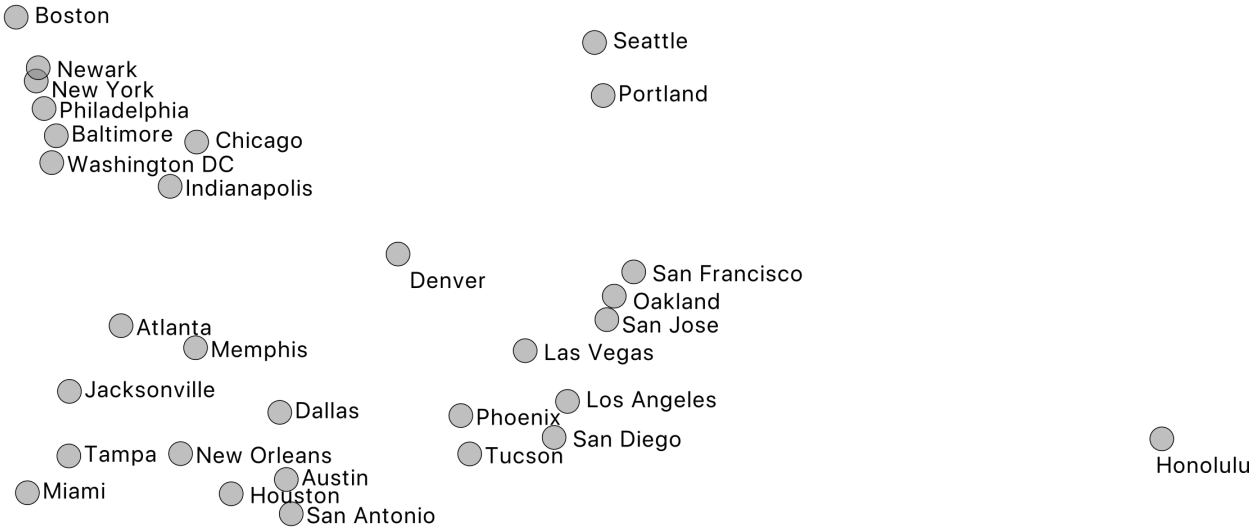
The real problem is New Orleans and San Antonio: New Orleans is close to Atlanta and Memphis, Miami is close to Jacksonville and

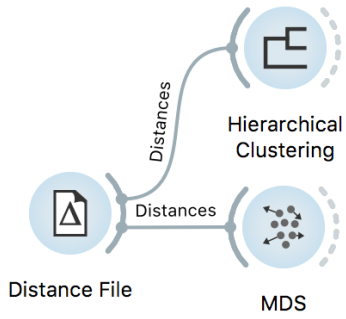
We can't run k-means clustering on this data, since we only have distances, and k-means runs on real (tabular) data. Yet, k-means would have the same problem as hierarchical clustering.

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.

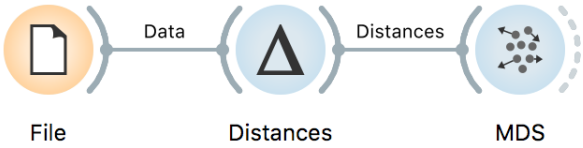


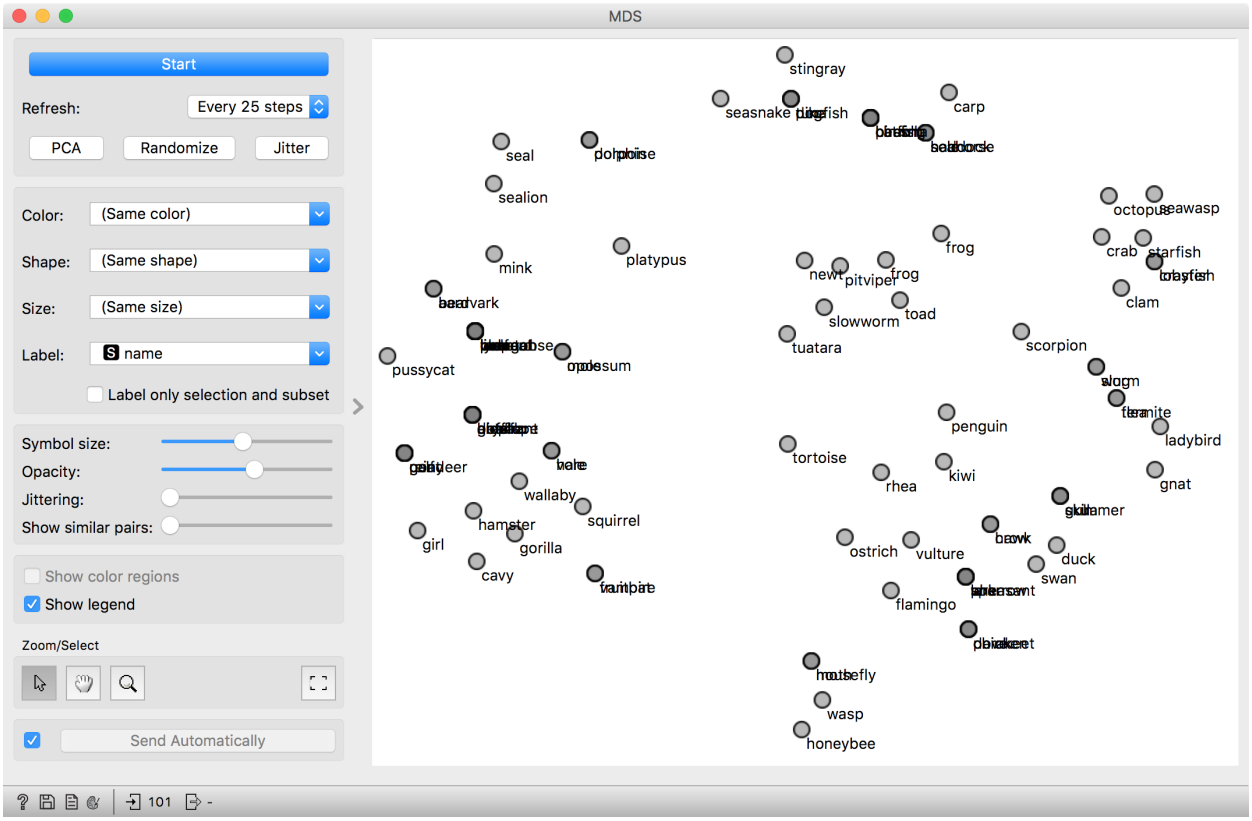


We have not constructed this map manually, of course. We used a widget called *MDS*, which stands for Multidimensional scaling.

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.





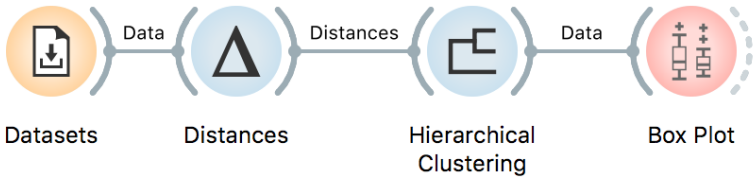
The map of the US was accurate: one can put the points in a plane so that the distances correspond to actual distances between cities. For most data, this is usually impossible. What we get is a projection (a non-linear projection, if you care about mathematical finesses) of the data. You lose something,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

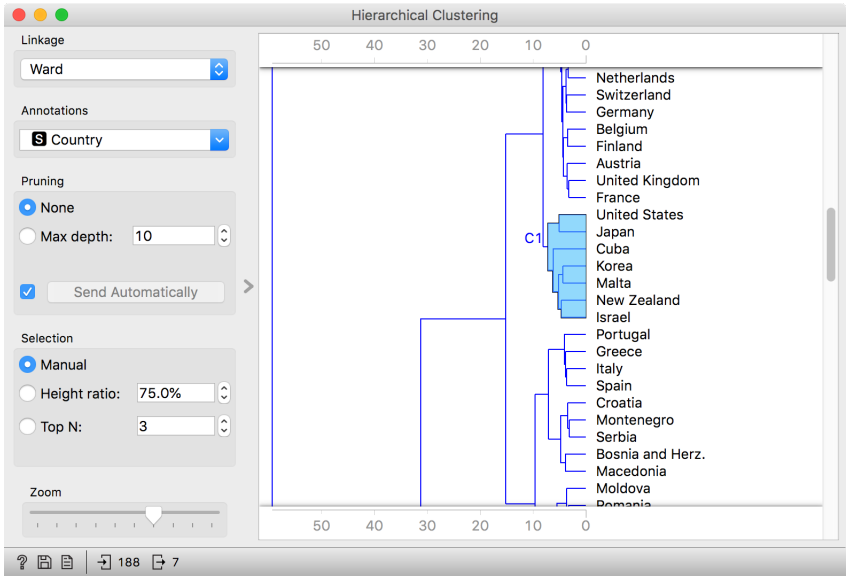
Assignment: Clustering

Clustering helps to discover groups of data, for example similar countries based on certain socio-economic features. For this task, we will use *HDI* data set from the *Datasets* widget. The data reports the Human Development Index for the year 2016 for 188 countries. While HDI has its limitations, the data offers an interesting exercise in clustering.

Workflow for the assignment.

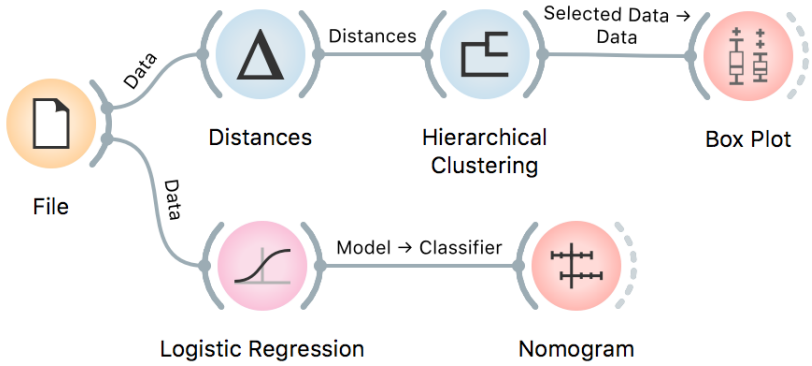


1. Try Euclidean and cosine distance with Ward linkage. Which one works better? Why?
2. How many groups did you discover? What number would make sense?
3. Explain the final clusters. What defines each of them?
4. Use Euclidean distance and Ward linkage. Can you explain why is Cuba clustered together with South Korea and the United Stated? Use *Box Plot* and the Data output from Hierarchical Clustering to answer this question.



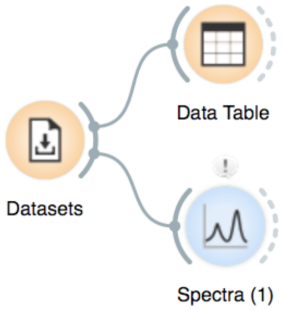
Assignment: Clustering vs. Classification

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



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

1. Explore the groups with *Box Plot*. What is the characteristic of cluster 1 (C1)? What is the characteristic of cluster 2 (C2)?
2. Which are the top three attributes distinguishing between clusters? Write them down.
3. In *Nomogram*, which are the three most important attributes for the model?
4. Are the attributes from clustering the same as those from logistic regression? Why (not)?
5. *Major vessels colored* is the most important attribute for logistic regression. How well does it split between the clusters?

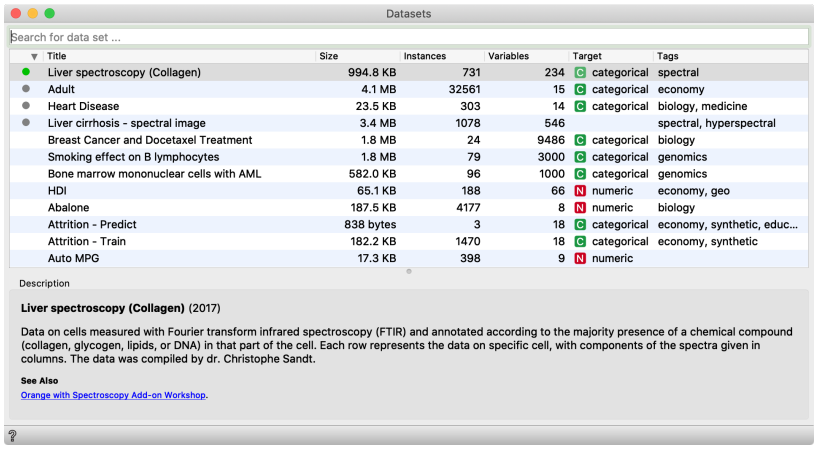


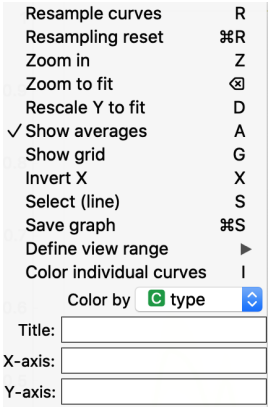
Your first spectroscopy workflow!

Spectral data

Let's make the small workflow shown on the right and open the "Liver spectroscopy" data set from Quasar's *Datasets* widget. In a *Data Table*, each row represents a spectrum. For the liver dataset, all columns, except the class column, describe absorbance at a specific wavenumber. Their column names must be numbers, otherwise Quasar's spectral tools will just enumerate them, starting from 0.

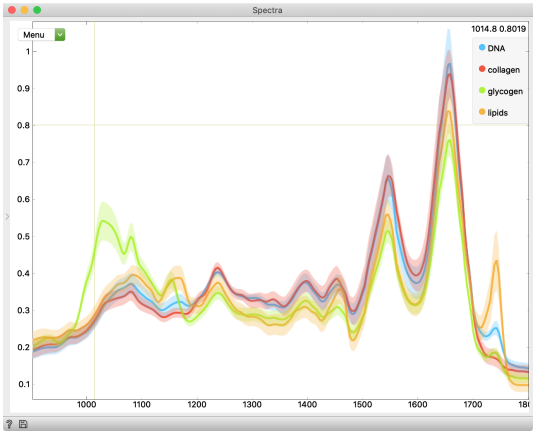
The *Datasets* widget provides data for training and testing purposes. The files are stored on a server and to use it, you need a working internet connection, but after you accessed them, they are stored on your computer for off-line use.



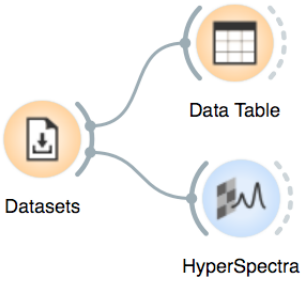


The *Spectra* widget and it's options. Try to use keyboard shortcuts on the right for frequent actions.

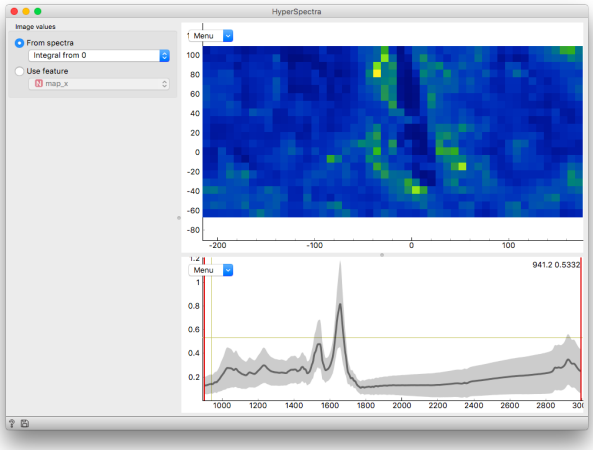
Connect the data to a *Spectra* widget from the **Spectroscopy** toolbox. To see the graph below, choose the feature for coloring in the top-left Menu (or click on the graph and press "c").



Working with hyperspectral data

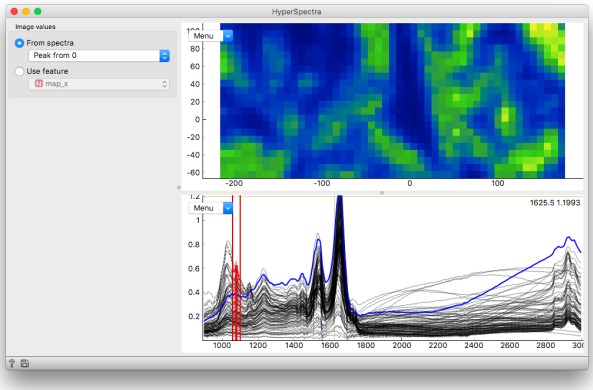


We can also visualize hyperspectral data sets. In the *Datasets* widget you will find "Liver cirrhosis" data. Connecting to a *Data Table*, you will see that each spectrum contains information (in meta variables) about image positions (map_x and map_y). Quasar can recognize the image positioning features from the file automatically. Otherwise, you could set them manually in the image Menu under the *Axis x* and *Axis y* options.



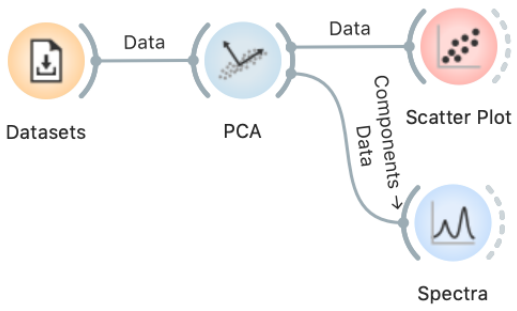
The *HyperSpectra* widget has two main parts, it can show image (top) and a spectra (bottom). Explore the options on both plots in their Menus and the left panel, where you can change the visualization parameters.

By default, the image is the 2D representation of the whole integral of each spectrum. To change it, move the red lines on the spectrum plot. With the dropdown menu on the left panel you can select other representations.



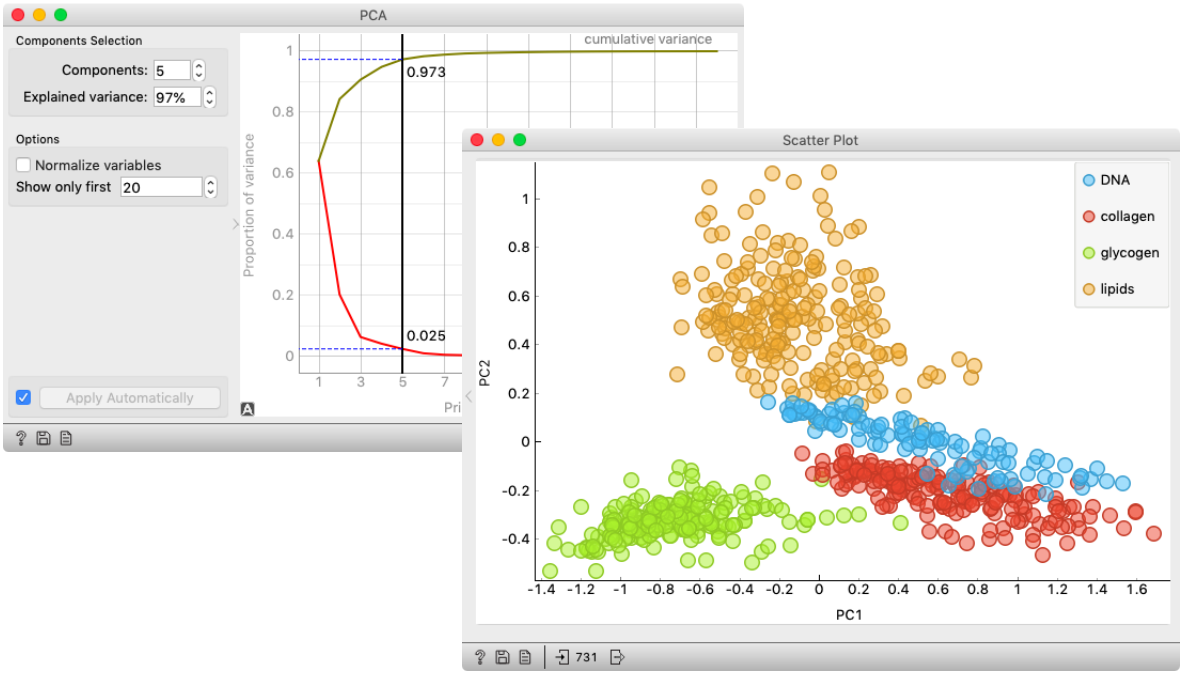
To view the plotted integrals, set the spectra display to show individual spectra and click a spectrum. Integrals for the selected spectrum are shaded.

PCA on spectral data



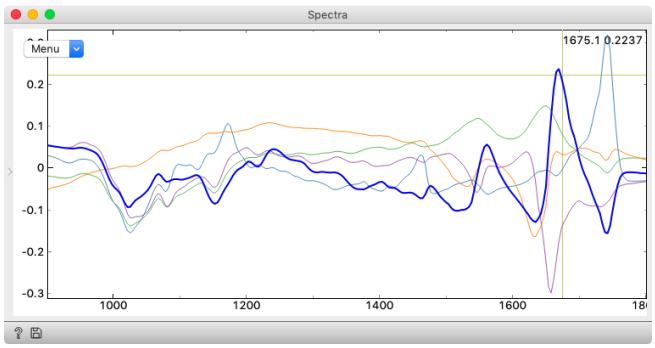
In this lesson we will explore the capabilities of Quasar for principal component analysis (PCA) on spectroscopy data. As usual, we will use the Liver Spectroscopy dataset. Connect *Datasets* to the *PCA* widget, choose the first 5 principal components and then connect *PCA*'s default output, "Data", into the *Scatter Plot*.

We see that the first two principal components separate majority compounds in that part of the tissue well.



We chose not to normalize variables in *PCA*. Why?

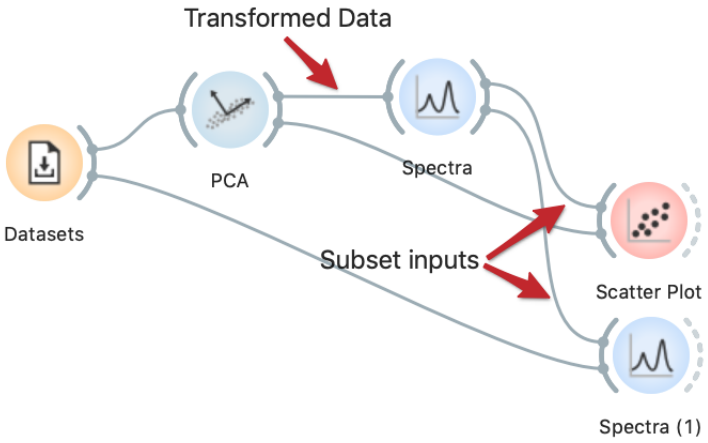
To see what different principal components represent, connect *PCA's* "Components" output (be careful, *PCA* has 4 outputs) into *Spectra*. Wondering which principal component is highlighted in the following screenshot? Wait for the tooltip...



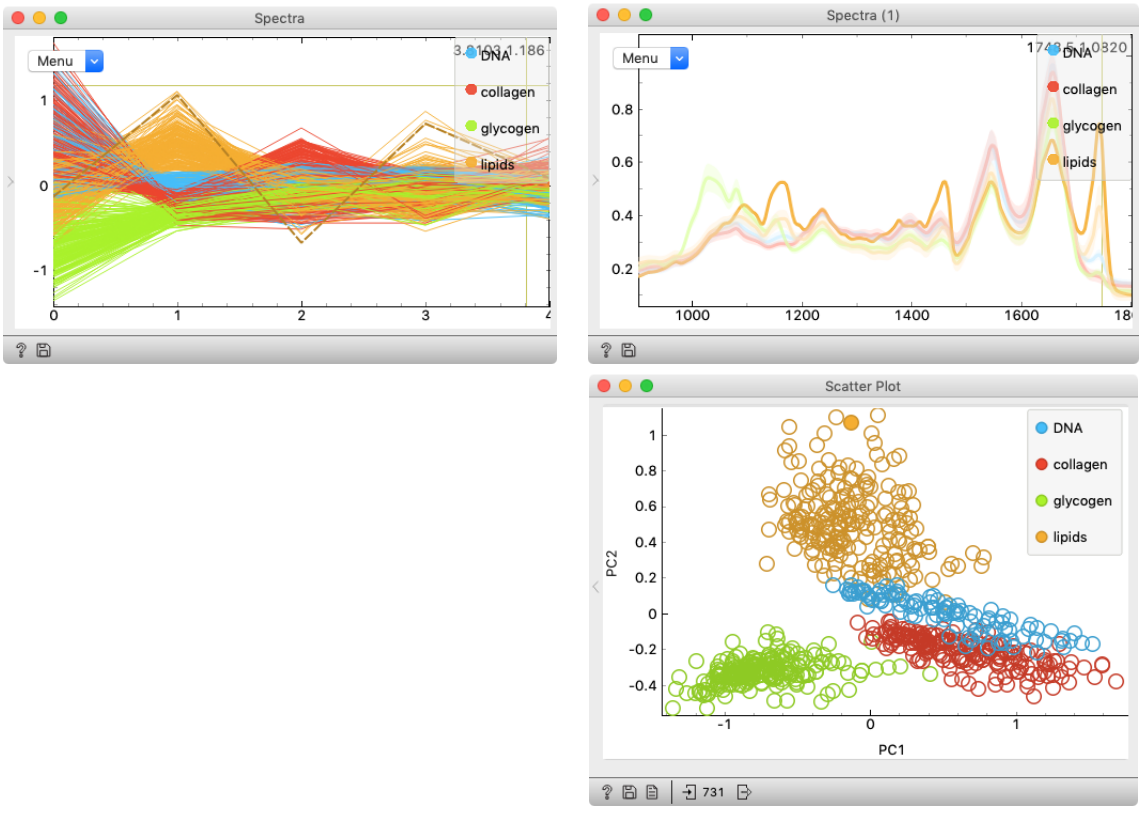
The curve under the cursor is highlighted. A tooltip will appear after some time. If clicked, the curve will be selected.

Let's extend our workflow. If we connect the *PCA* ("Transformed Data" output) to *Spectra*, we can see each transformed spectrum on a line plot. As we can see, some classes have outliers.

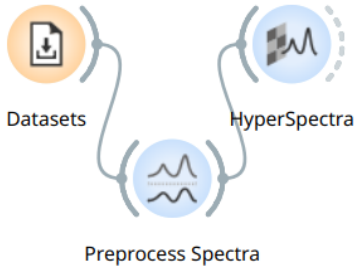
To find out more about a particular outlier, we can select it in the *Spectra* widget: move your mouse cursor to a curve—it will be highlighted—and click it. The selected curve changes to a dotted line and is sent to the output.



Then, connect the *Spectra* widget to the *Scatter Plot* and the *Spectra (1)* widgets' "Data Subset" inputs; these widgets will need two inputs to function as shown, the subset coming from the selection in *Spectra* and the whole data set. Now we can see the selected outlier in the original space (*Spectra (1)* widget) and in the space of principal components (*Scatter Plot*), both in the context of all spectra from the data set.



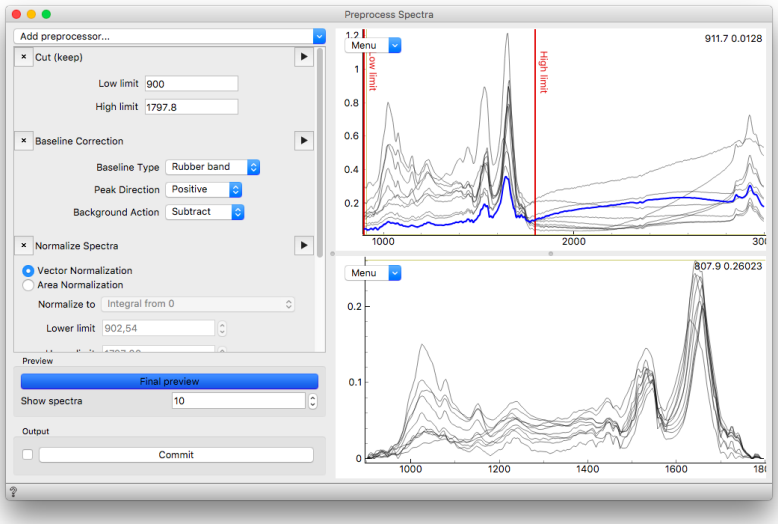
The selected spectrum's curve on the left is drawn with a dashed line and the corresponding original spectrum is highlighted on the right. The *Scatter Plot* shows the position of the selected spectrum in the PCA space.



Preprocessing spectral data

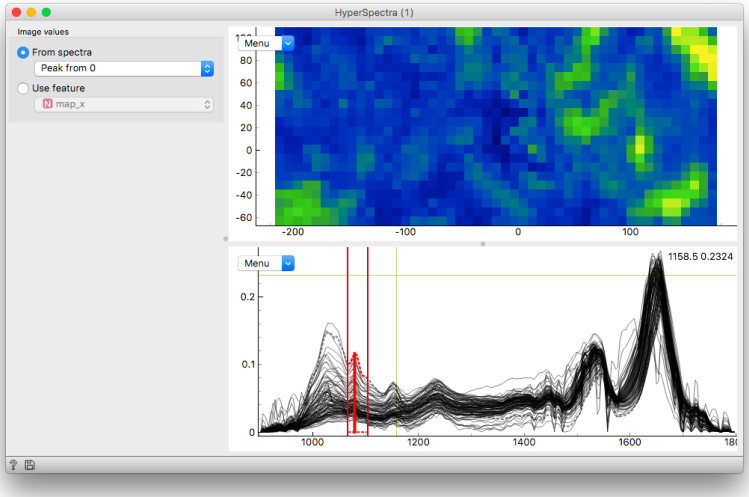
Preprocessing spectra is a very important step of data analysis. Quasar has a widget, *Preprocessing*, dedicated to different methods. The spectra from the "Liver cirrhosis" data set could use some preprocessing. There is some scattering visible and perhaps there are some artifacts due to sample thickness varying slightly.

You can add preprocessing steps from the top dropdown menu of the left panel. Then, it is possible to drag them up and down to change their order. Each preprocessor has its own parameters. In the example here we show how the Baseline correction is done: you can simply change the baseline points by dragging the red lines in the top spectrum panel. Each stage can be previewed by clicking the small triangle.

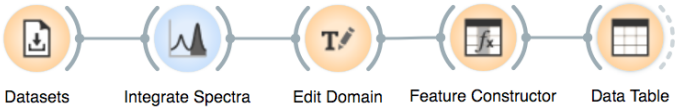


Let's see the result of our preprocessing in a *HyperSpectra* widget.

The preprocessed data in *HyperSpectra*. Did we gain anything? To investigate why the blue island disappeared, click on a pixel in it to see its spectrum.

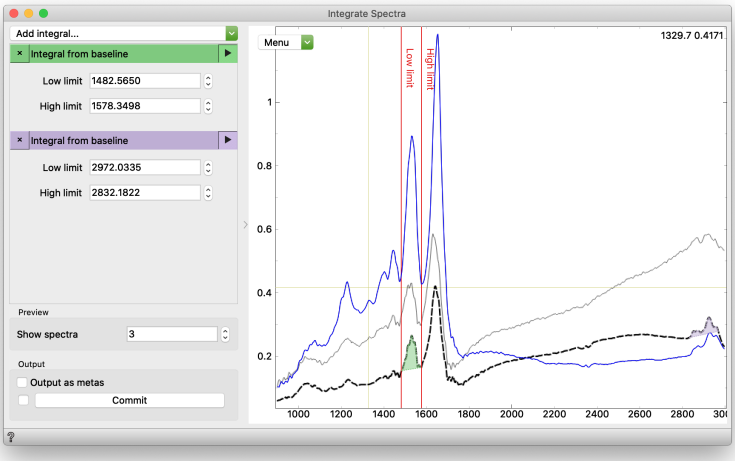


Integrals and ratios

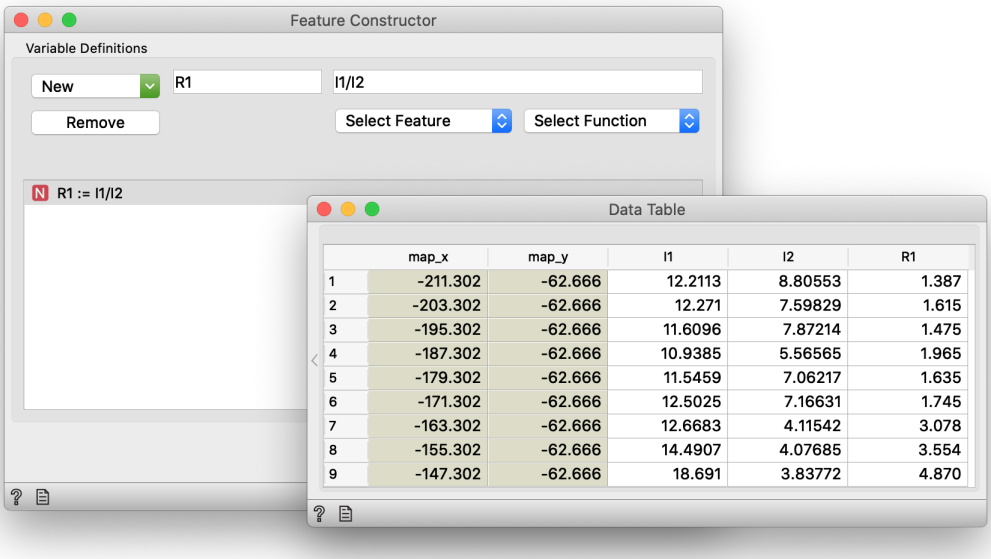


Peak integration is an essential element of spectroscopy for measuring concentrations, spectral contributions, etc.

To compute integrals, we use the *Integrate Spectra* widget. Let's compute the ratios of two integrals to establish an internal standard in our dataset. Add two integrals and then feed them into the *Feature Constructor* (*Edit Domain* is optional—we use it to simplify column names). In *Feature Constructor*, we can create new numeric features with Python expressions. To work with Feature Constructor more easily, uncheck "Output as metas", which will replace the original spectra with their integrals (and reduce the number of columns in your data). We can use *Feature Constructor* whenever we would like to create a new column from existing data.



To display a preview, select a spectrum and enable preview of individual integrals with their play buttons.

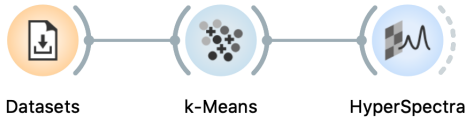


	map_x	map_y	I1	I2	R1
1	-211.302	-62.666	12.2113	8.80553	1.387
2	-203.302	-62.666	12.271	7.59829	1.615
3	-195.302	-62.666	11.6096	7.87214	1.475
4	-187.302	-62.666	10.9385	5.56565	1.965
5	-179.302	-62.666	11.5459	7.06217	1.635
6	-171.302	-62.666	12.5025	7.16631	1.745
7	-163.302	-62.666	12.6683	4.11542	3.078
8	-155.302	-62.666	14.4907	4.07685	3.554
9	-147.302	-62.666	18.691	3.83772	4.870

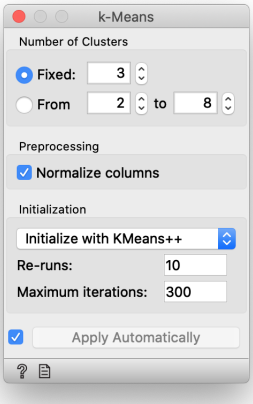
We added a Numeric feature, the ratio of I1 and I2 called R1.

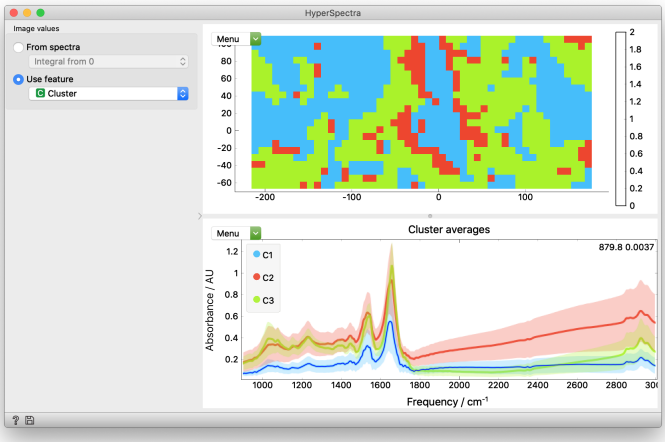
The produced data can be inspected by connecting other widgets.

Clustering Spectral Images

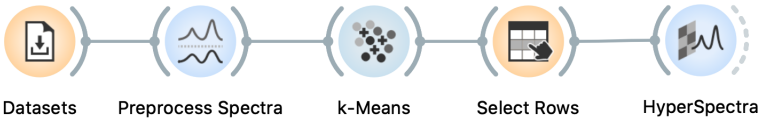


We have already seen hierarchical clustering. Another clustering algorithm, k-Means, is much faster for data with lots of rows, like images, which contain a row (a spectrum) for each pixel. Still, for the liver-cirrhosis data, both approaches are fast. Here, we use *k-Means* with k=3 clusters.

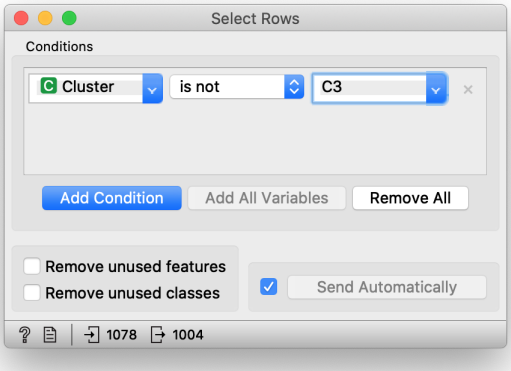


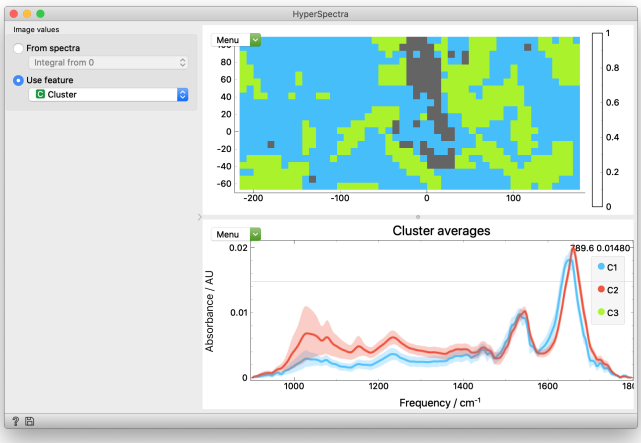


The *Spectra* widget shows wrong predictions for the DNA class.



We see no meaningful clusters. Therefore, we need to preprocess the data. If we do it well, we see that a cluster corresponds to the background. We could remove it with the *Select Rows* widget.

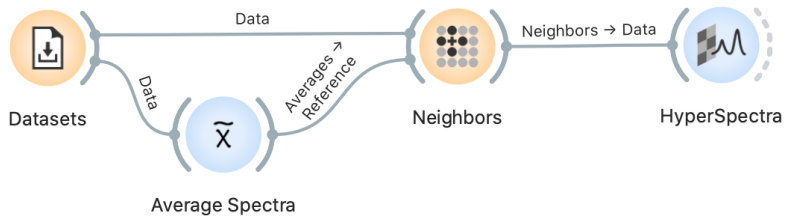




The *Spectra* widget shows wrong predictions for the DNA class.

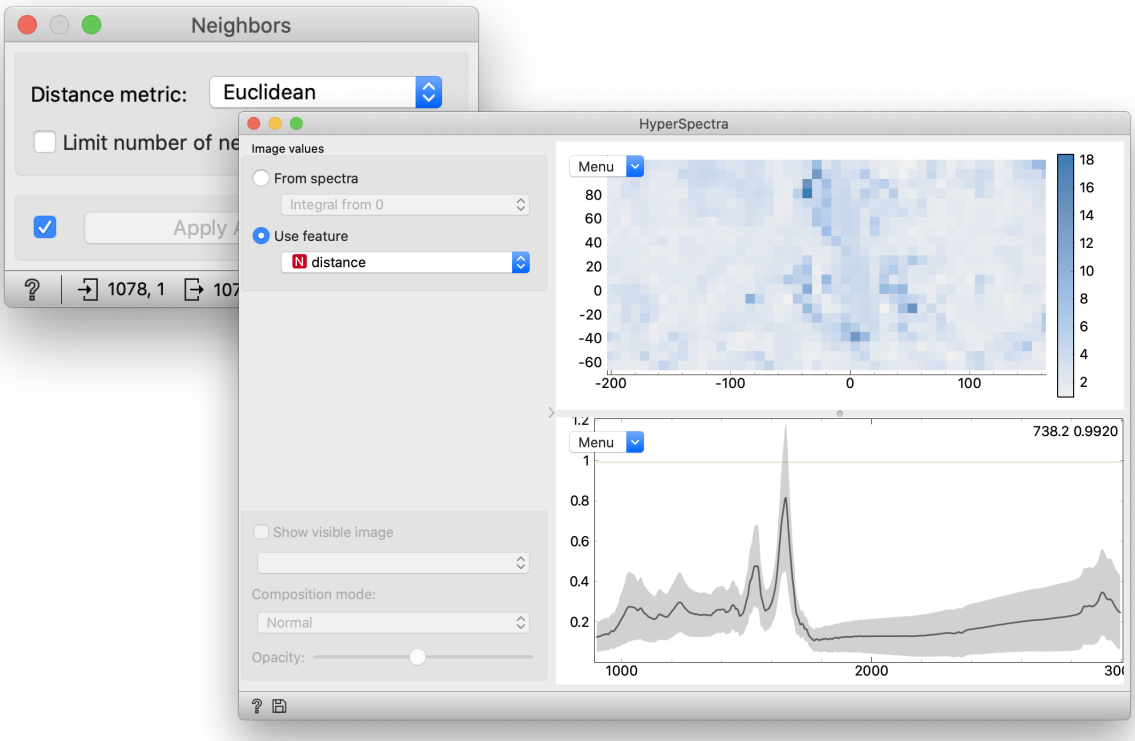
Visualize spectral distances

Let's consider a spectrum, or any other data entry for that matter, as a point in a multidimensional space. We can define distance metrics between these points and visualize the distance values from one another or from a selected reference point or reference spectrum. By doing so, we can explore how similar our measurements are to a selected reference. We can do this on a series of spectra or even on hyperspectral maps!



Load the *'Liver cirrhosis - spectral image'* dataset from the *Datasets* widget and calculate the *Euclidean distances* from the average spectrum with the *Neighbors* widget. Visualize them in *Hyperspectra*.

Can you reproduce the results below? Pay attention to the color scheme.

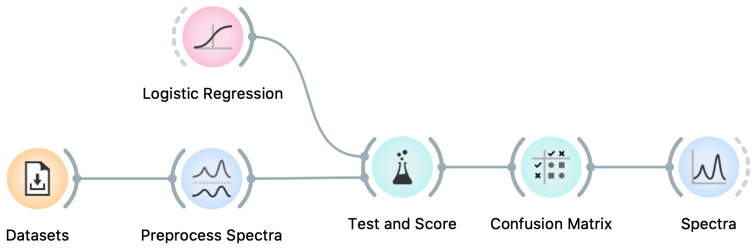


Explore different distance metrics, inspect distances in a *Data Table* widget. Don't forget, you can select points on the top map and see the corresponding spectra on the bottom in *Hyperspectra*.

Lecturer note. Possibility for discussion of the general mathematical properties of distance functions. See https://en.wikipedia.org/wiki/Metric_(mathematics)

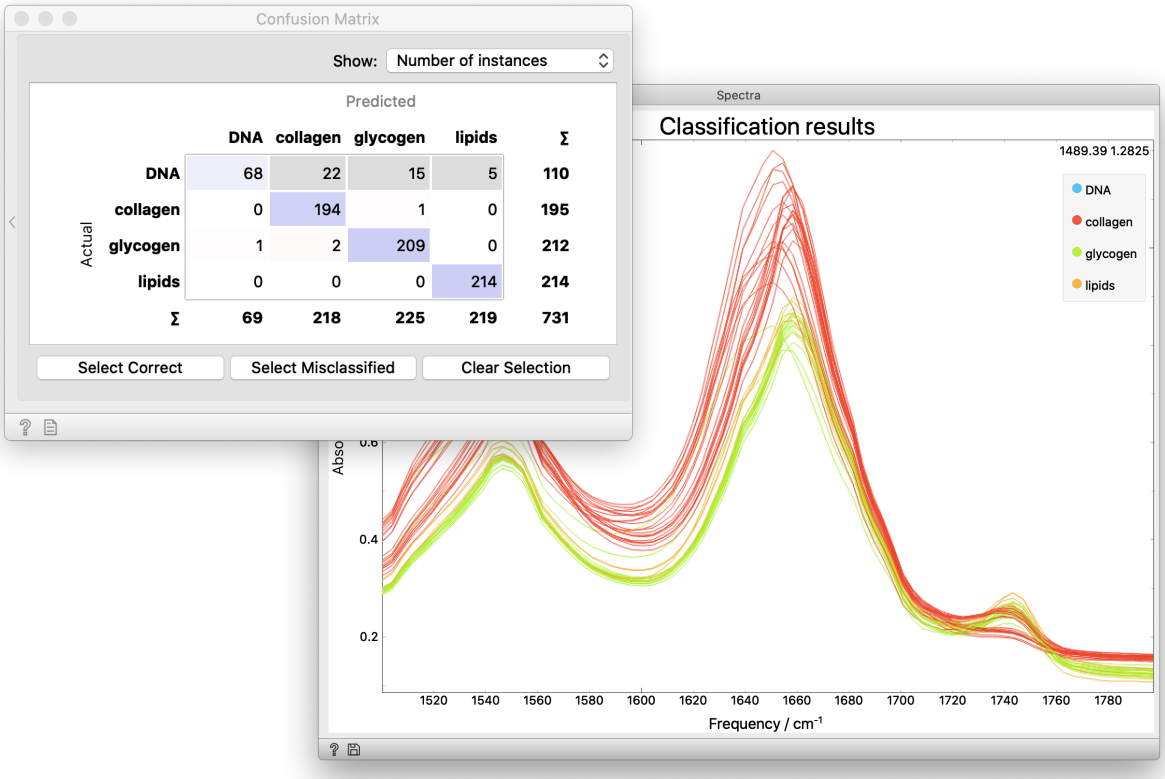
Classification of Spectra

Let's open the collagen data set again and see how well can logistic regression predict its four classes. Straightforward, right?



In *Preprocess Spectra* we also did some spectral processing: we decided to only keep columns for wavenumbers between 1500 $cm^{-1}$and 1800 cm^{-1}.

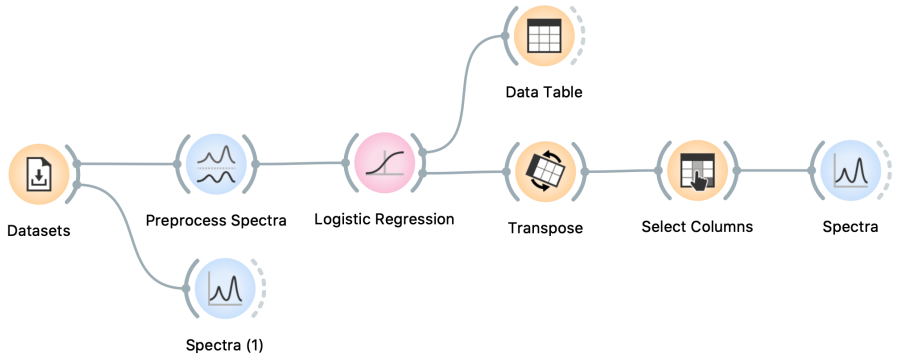
Connect *Datasets*, *Logistic Regression*, *Predictions*, *Confusion Matrix* and that's it.



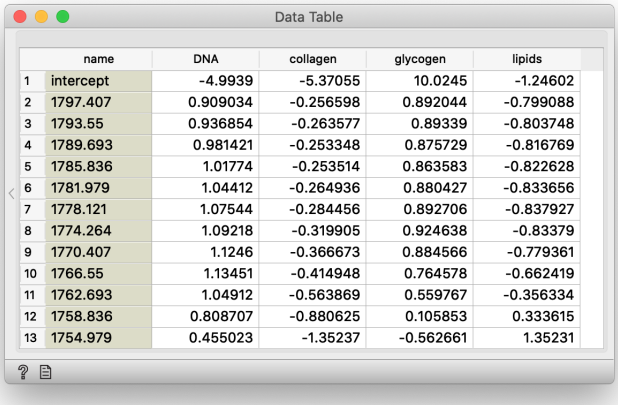
In the *Confusion Matrix* we selected wrong predictions for the actual class DNA. The connected *Spectra* widget displays them.

Let's not forget that it is pointless to predict for the same data as we used for learning. We could either use a *Data Sampler* and connect its Sample output to *Preprocess Spectra* and Remaining output to *Predictions*, or obtain predictions from the *Test and Score* widget. *Confusion Matrix* now shows the mistakes of the model (scored with cross-validation). We can select them and inspect them further in a

Spectra widget. Here we colored them by the predicted class (see the Menu).



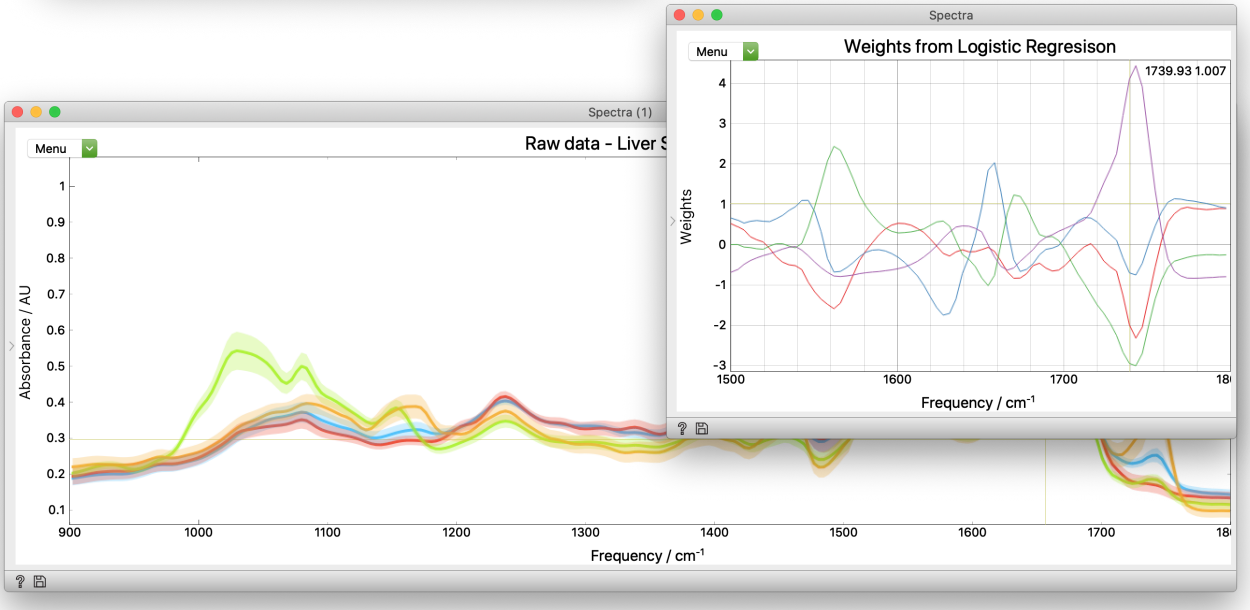
But how does the model make its decisions? We already inspected a different model, classification tree, where each node represents a decision on a value of a column. *Logistic regression* works differently. On the training data it computes weights for all columns (wavelengths), which are then used for prediction, where values are multiplied with weights. To see the weights, connect *Logistic Regression* to a *Data Table*.



	name	DNA	collagen	glycogen	lipids
1	intercept	-4.9939	-5.37055	10.0245	-1.24602
2	1797.407	0.909034	-0.256598	0.892044	-0.799088
3	1793.55	0.936854	-0.263577	0.89339	-0.803748
4	1789.693	0.981421	-0.253348	0.875729	-0.816769
5	1785.836	1.01774	-0.253514	0.863583	-0.822628
6	1781.979	1.04412	-0.264936	0.880427	-0.833656
7	1778.121	1.07544	-0.284456	0.892706	-0.837927
8	1774.264	1.09218	-0.319905	0.924638	-0.83379
9	1770.407	1.1246	-0.366673	0.884566	-0.779361
10	1766.55	1.13451	-0.414948	0.764578	-0.662419
11	1762.693	1.04912	-0.563869	0.559767	-0.356334
12	1758.836	0.808707	-0.880625	0.105853	0.333615
13	1754.979	0.455023	-1.35237	-0.562661	1.35231

We get a table that is hard to understand. What if we visualize it? First, *Transpose* the data. Then, use *Select Columns* to make the visualization prettier: in the widget remove the intercept.

Now, open *Logistic Regression* and try changing its parameters. Observe the effect on the weights.



Common Errors in Supervised Modelling

Here we shows some common errors with supervised learning. These errors are general—not limited to Quasar. Understanding them improves our understanding of supervised learning and might even influence our approach to data acquisition. We are going to simulate these errors as follows:

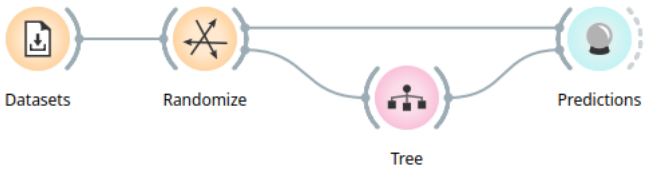
1. First, create a data set where classes should be impossible to predict.
2. Then, do some operations on the data.
3. Finally, observe that supervised learning found some way of discriminating the classes.

Since data was prepared to be inseparable, any correlation between features in the data and the classes should be spurious correlations.

Predicting on the training data

We will first create a data set with no correlation between spectra and their class variable. To do this, we open the "Liver spectroscopy" data set and shuffle the classes with *Randomize* (set it to shuffle 100%). *Randomize* destroys any connection between features and the class variable. Therefore, supervised models should perform badly on this data set. More precisely, we would expect them to perform similarly to the majority classifier (that is a classifier that assigns a sample to a class which is in the majority in the training set). **Verify with cross-validation on the randomized and non-randomized data sets**.

Next, build a *Tree* on the randomized data and look at predictions on the whole data set: *Predictions*. The *Tree* seems to perform almost perfectly. We know that classes were randomized, so there should be no correlations between spectra and the classes. How could a *Tree* even model anything?



To answer the previous question, observe trees (with a *Tree Viewer*) built on both randomized and non-randomized data. Notice that the

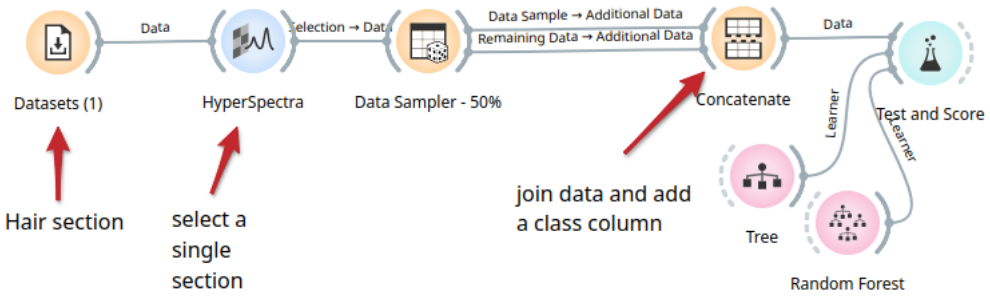
tree on randomized data is much more complex. The tree can grow indefinitely and can, therefore, model any correlations, no matter how small these correlations are. Even random data will provide enough correlations for a tree model. Fortunately, correlations between spectra and classes in randomized data differ in each subset of data. Therefore, testing on separate test sets will allows us to discover if the model is overfitted.

Some methods are more prone to overfitting than others. It is harder to overfit with PLS or logistic regression than with classification trees. However, even PLS or logistic regression have a tendency to overfit. **Try and see the difference**. In addition, some models have parameters controlling model complexity. **Explore effects of parameters in the *Tree* widget**.

We saw that we can build seemingly good models even on data where it should be impossible if we do not properly evaluate the models. Therefore, validate all models carefully.

Separate preprocessing of data corresponding to different classes may introduce spurious correlations

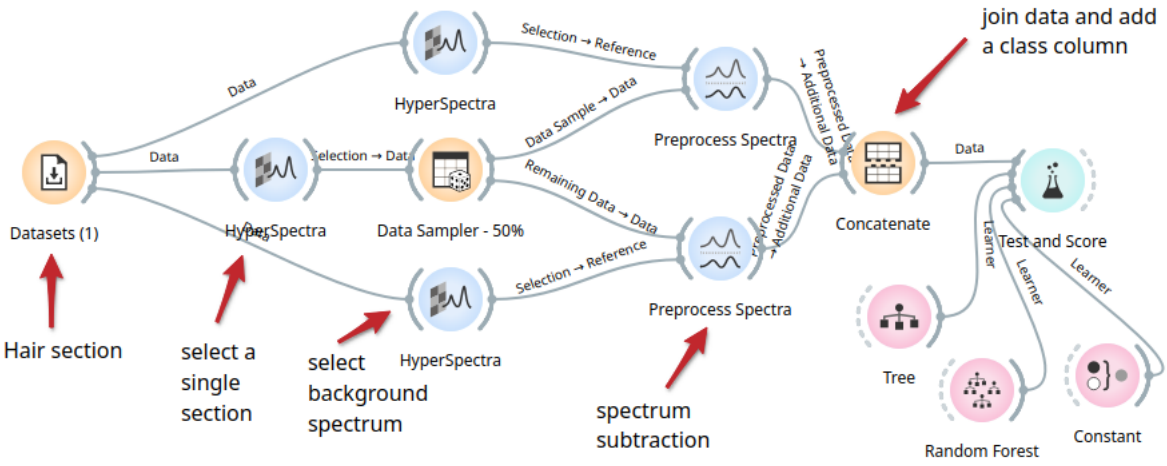
We will now create another dataset and classes that should be impossible to classify: we randomly split spectra it into two groups, and annotate each group with a different class value. For this numerical experiment, we will use the "Hair section" data set, which contains spectra on multiple hair sections organized as a spectral image. For simplicity, select one section in the *HyperSpectra* widget (use polygonal selection). Then, split the data randomly into two halves with the *Data Sampler* widget. Join the two halves together with *Concatenate* (set it to add a class column depending on the source). Now, we have a data set that is hard to classify since the classes are random. **Perform cross-validation and report the scores**.



Learning methods perform badly here. AUC should be around 0.5.

Next, let us introduce separate preprocessing for each of the classes. For each class, we will subtract a different background spectrum. Select one background spectrum per group (two *HyperSpectra* widgets) and send them as references into corresponding *Preprocess Spectra* widgets. Preprocess with Spectrum subtraction with weight set to 1.

Notice that the preprocessing effect is very slight. After concatenating, perform cross-validation. **Report the scores. How large was the effect of only slightly different preprocessing?**



Even slightly different preprocessing allows random classes to separate easily, if preprocessing was done for each class separately.

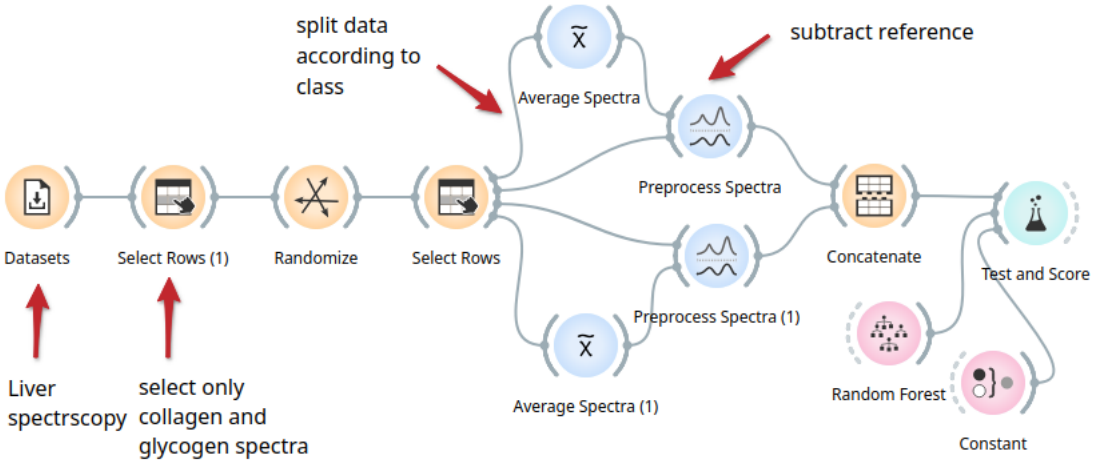
What happens with other types of preprocessing? **Try other preprocessing methods. Which methods produce separable data and which do not? What do they have in common?**

We saw that even slight differences in preprocessing create a significant fake signal. A similar 'fake effect' may happen as well when you measure samples of different classes on different days. Some physical conditions, e.g. in the instrument, may introduce a systematic error, e.g. a baseline, that may affect the spectra. It may then happen that we are able to build good models because of physical effects and not because the classes have a chemical difference that is measurable in the spectra.

Exercise: separate processing of classes evaluate for spectra obtained from liver tissue.

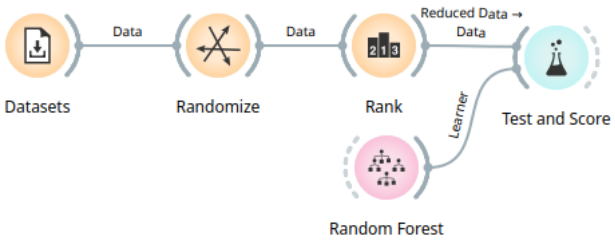
Use the "Liver spectroscopy" data set, select two of its classes, randomize it, and apply EMSC to each class separately. Observe prediction quality. Can you separate the classes? Do you achieve a clear separation?

Next, use a gentler way of preprocessing: subtract the average of each class from the data. The difference is smaller, but would you say that it is significant? If the difference is too small, how could you make it bigger?



Selection of promising features (wavenumbers) on the whole data set

This common error in supervised modelling does not influence spectral data that much due to auto-correlation. While the influence can still be observed on small spectral data sets, let us use a gene microarray data "Breast Cancer and Docetaxel Treatment". First, we randomize the classes to destroy any predictive value of the features. Then we use Rank to select five columns best connected with the class according to the ANOVA test.



Even though we are evaluating with cross-validation, which correctly separates training and testing sets, *Test and Score* shows classification accuracy around 0.9. We had to introduce unwanted bias somewhere.

In a workflow of this size, it should not take long to find the culprit. It is the *Rank* widget, or more precisely, the ANOVA test it uses to select promising features. ANOVA measures correlations between features and the class, and here lies the error: ANOVA uses information on the class before we split the data into training and testing sets. We would need to ensure that ranking is performed on the training set only to correct the error. **Verify that classes are not separable when ranking on the training set only.**

Exercise: repeat the same with spectral data.

Repeat the same exercise using spectral data. To be able to see the error there, you will need to use very few spectra. Furthermore,

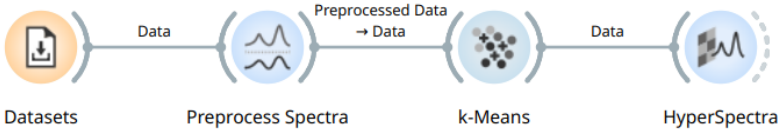
co-linearity in FTIR spectral data makes this difficult to see—also consider randomization that partly destroys auto-correlation.

Exercises

Clustering

Open the "Liver-cirrhosis" data set from the *Datasets* widget. It represents a sliced liver tissue. As tissues have holes and they also tear during the slicing some parts of the dataset has very low spectral intensities. Let's call these "background regions".

1. Remove background regions from the data set in any way you see fit. You are likely to need some preprocessing. Also clustering, as seen below, could be useful for for this. Try to make it work.



2. Afterwards, find two groups of spectra with the most obvious differences. The preprocessing used for background removal may not be the best for clustering; to obtain a selection of original data, use the *Select by Data Index* widget.

3. Describe differences between the two groups. Try at least *Spectra*, *Distributions*, *Violin Plot*, *Scatter Plot*.

Classification

1. Use either a spectral image or unannotated spectra. Manually annotate classes as you see fit. Assign classes only according to shapes in the image or filenames; classes set according to spectral content **will** yield good separation, because that is a tautology.

2. Use Random Forests and Logistic regression. Interpret the models.

3. Observe the effects of different preprocessing on classification performance and model interpretation. Also try models built on the PCA scores.

Game of bias

Start with a dataset with annotated classes. Randomize class values to destroy connections between spectra and classes. Next, think of a novel way to introduce bias in the data set so that AUC becomes significantly higher than 0.5 even with cross-validation within *Test and Score*.