BIOLAB AND COLLABORATORS

# USING QUASAR

BIOLAB

PUBLISHED BY BIOLAB

TUFTE-LATEX.GOOGLECODE.COM

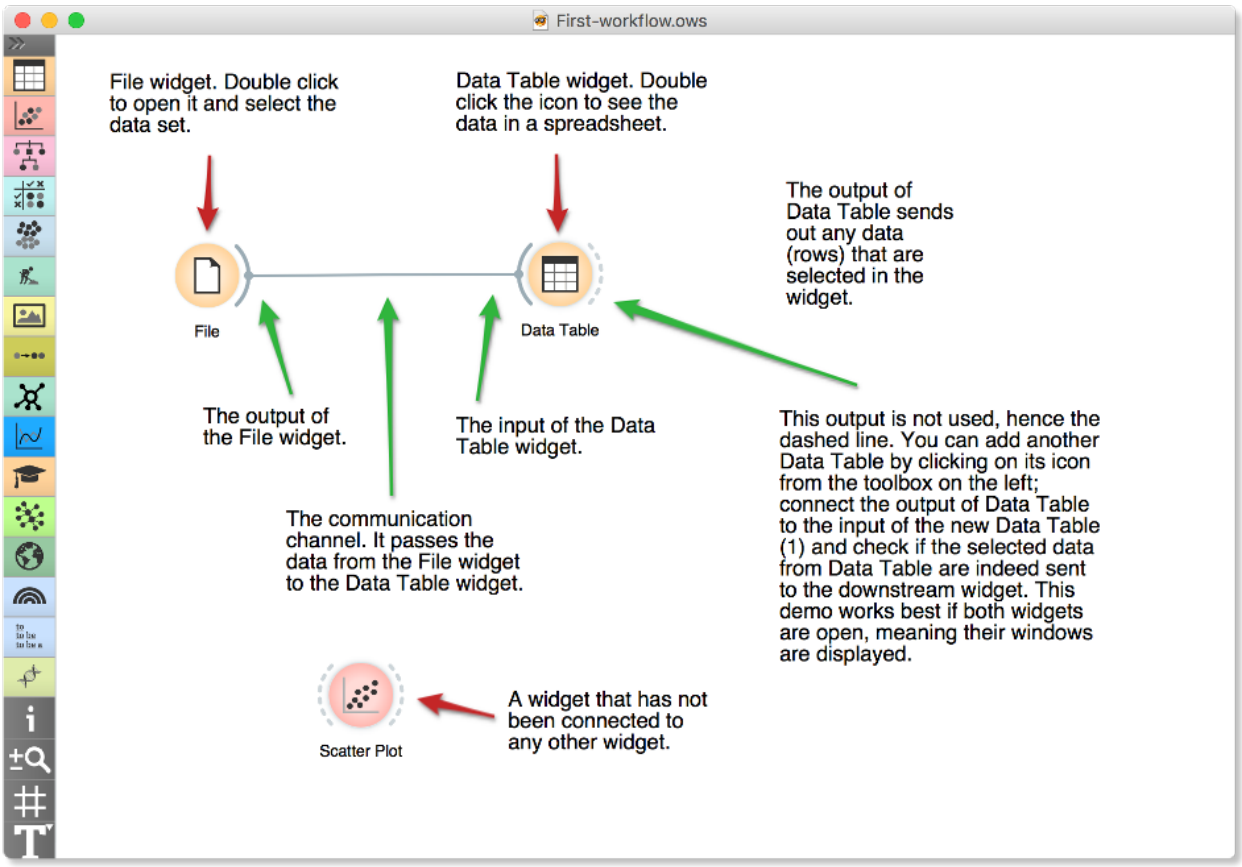


*First printing, November 2020*

# *Contents*

# *Workflows in Quasar*

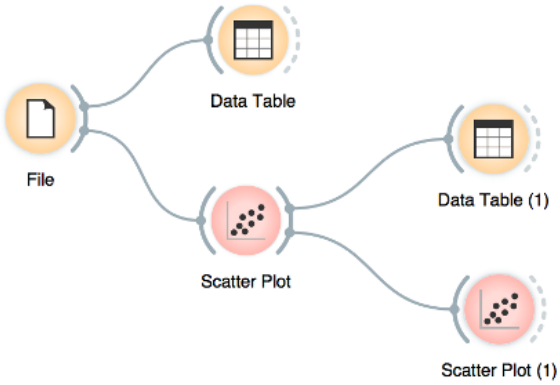
Quasar workflows consist of components that read, process and visualize data. We call them "widgets". Widgets are placed on a drawing board (the "canvas"). Widgets communicate by sending information along a communication channel. Output from one widget is used as input to another.




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.

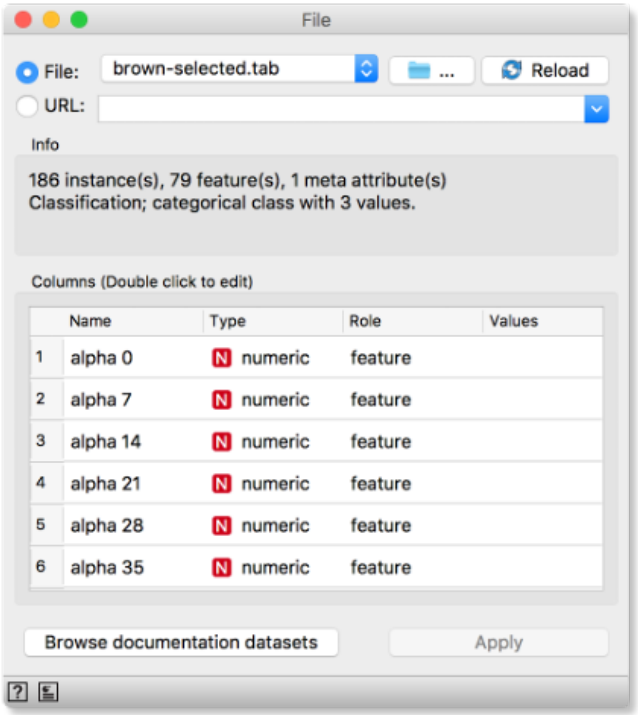
We construct workflows by dragging widgets onto the canvas and connecting them by drawing a line from the transmitting widget to the receiving widget. The widget's outputs are on the right and the inputs on the left. In the workflow above, the File widget sends data to the Data Table widget.

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




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

The File widget reads data from your local disk. Open the File widget by double clicking its icon. 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 of a 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.

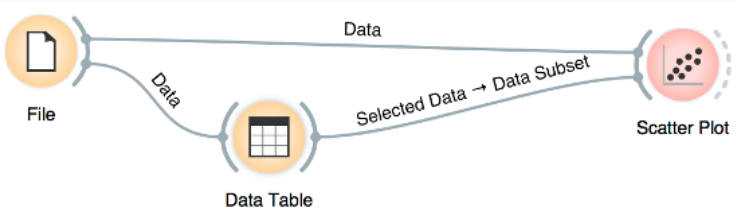
After you load the data, open the other widgets. In the Scatter Plot widget, select a few data points and watch as they appear in widget Data Table (1). Use a combination of two Scatter Plot widgets, where the second scatter plot shows a detail from a smaller region selected in the first scatter plot.

The following is more of a side note, but it won't hurt. Namely, the scatter plot for a pair of random features does not provide much information on gene function. Does this change with a different choice

of feature pairs in the visualization? Rank projections (the button on the top left of the Scatter Plot widget) can help you find a good feature pair. How do you think this works? Could the suggested pairs of features be useful to a biologist?

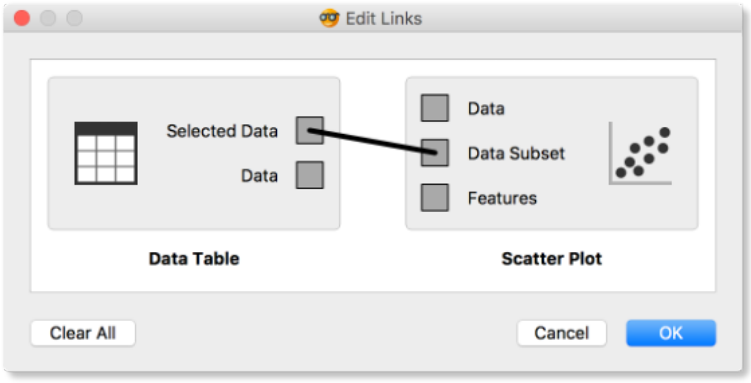

Scatter Plot and Ranking

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.

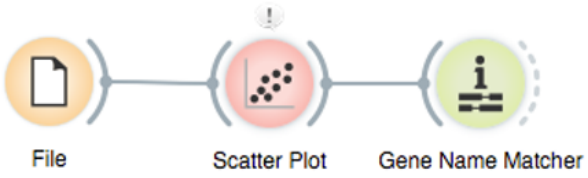



In this workflow, we have switched on the option "Show channel names between widgets" in File/Preferences.

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.



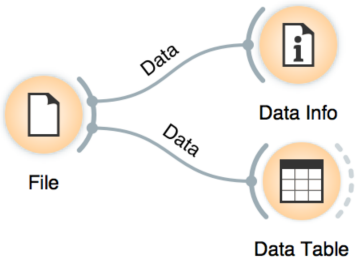

**Quasar comes with a basic set of widgets for data input, preprocessing, visualization and modeling. For other tasks, like text mining, network analysis, and bioinformatics, there are add-ons. Check them out by selecting Add-ons... from the Options menu.**

# *Basic data exploration*

LET US CONSIDER ANOTHER PROBLEM Let us consider another problem, this time from clinical medicine. We will dig for something interesting in the data and explore it a bit with visualization widgets. You will get to know Quasar better, and also learn about several interesting visualizations.

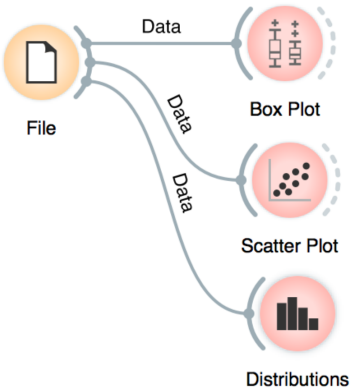
We will start with an empty canvas; to clean it from our previous lesson, use either File/New or select all the widgets and remove them (use the backspace/delete key, or Cmd-backspace if you are on Mac).

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.

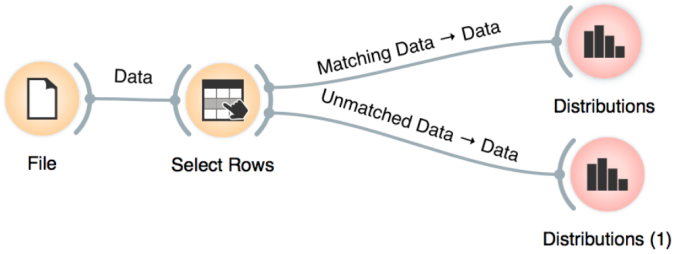
Let us check whether common visualizations tell us anything interesting. (Hint: look for gender differences. These are always interesting and occasionally even real.)

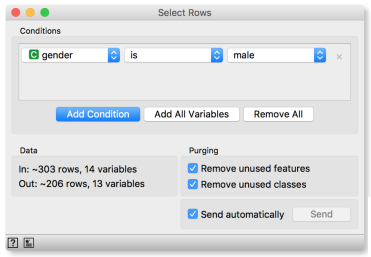



Quick check with common statistics and other visualization widgets.

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

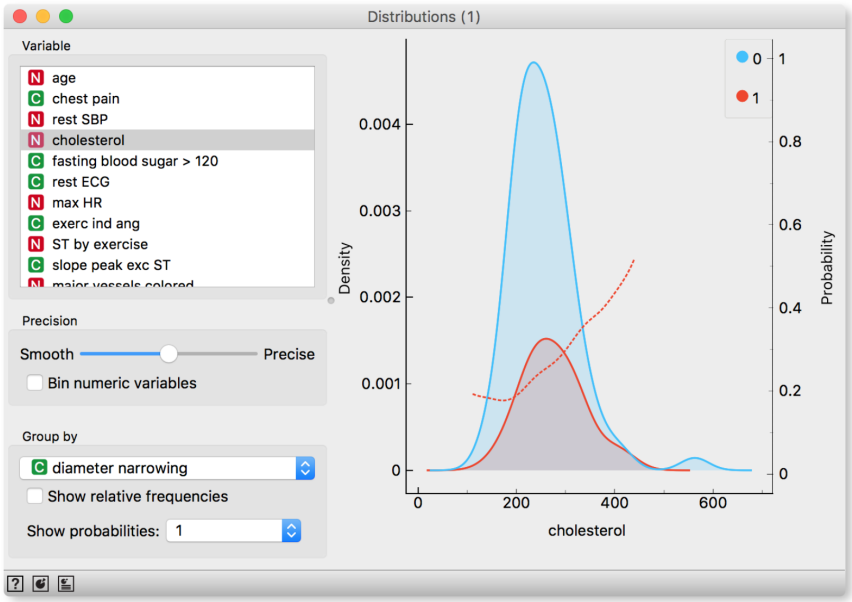
The two Distributions widgets get different data: the upper gets the selected rows and the lower gets the rest. Double-click the connection between the widgets to access setup dialog, as you've learned in the previous lesson.




Select Rows and Distributions widget

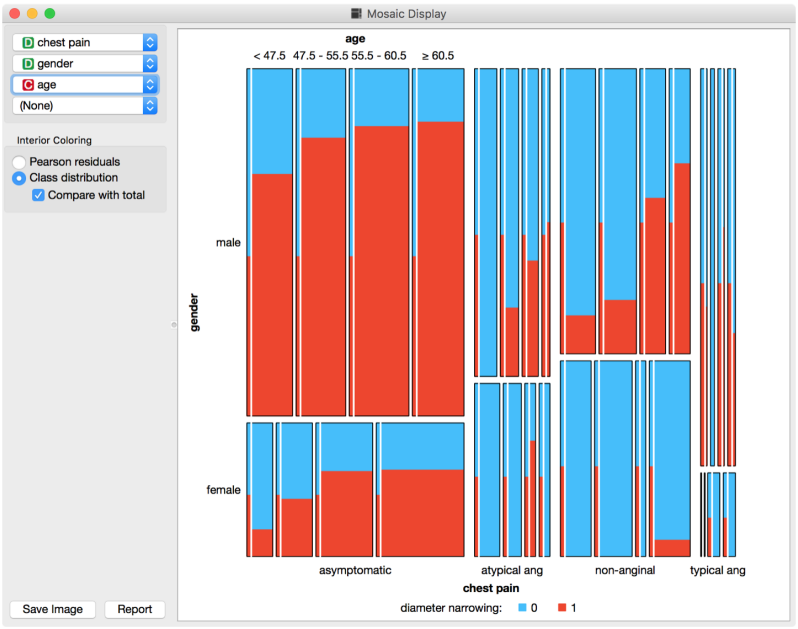
In the Select Rows widget, we select the female patients. You can also add other conditions. Selection of data instances provides a powerful combination with visualization of data distribution. Try having at least two widgets open at the same time and explore the data.




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

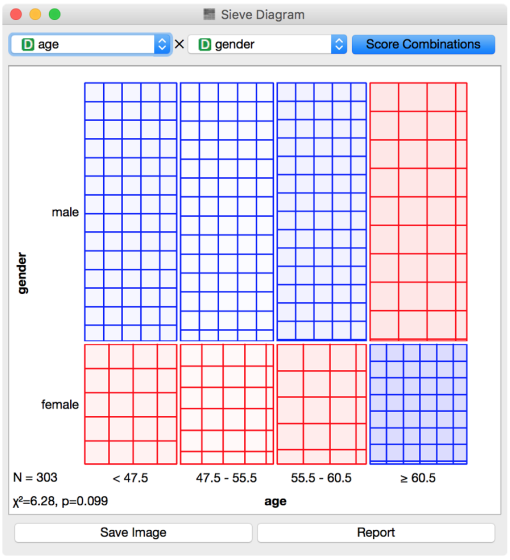
Mosaic display shows a rectangle split into columns with widths reflecting the prevalence of different types of chest pain. Each column is then further split vertically according to gender distributions within the column. The resulting rectangles are split again horizontally according to age group sizes.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

What can you read from this diagram?




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

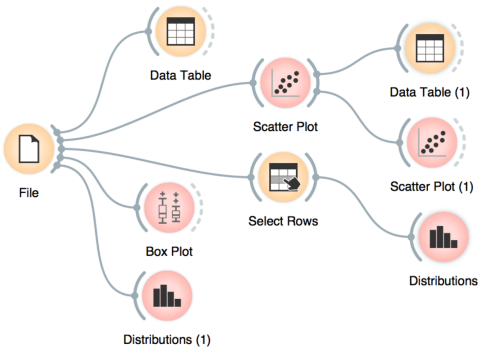
Another visualization, Sieve diagram also splits a rectangle horizontally and vertically, but with independent cuts, so the areas correspond to the expected number of data instances if the observed variables were independent. For instance, 1/4 of patients are older than 60, and 1/3 of patients are female,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




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

# *Saving your work*

At the end of a lesson, your workflow may look like this:

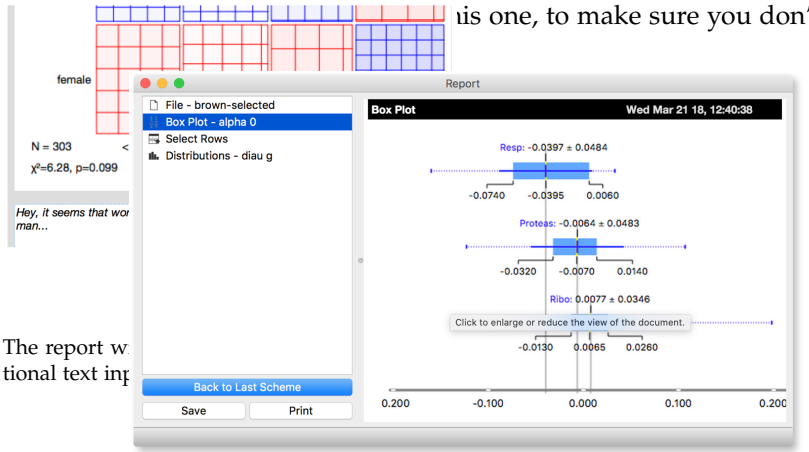



A fairly complex workflow that you would want to share or reuse at a later time.

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

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

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

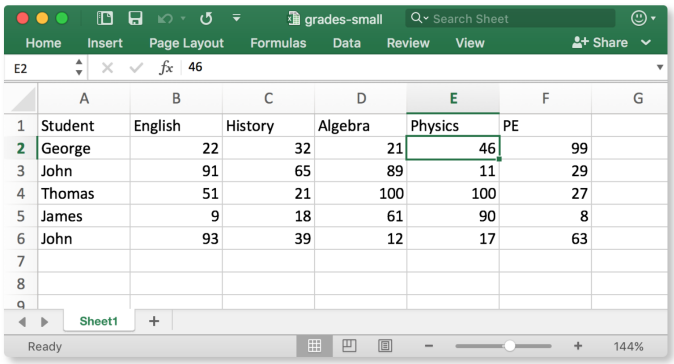



The report w
tional text inp

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*

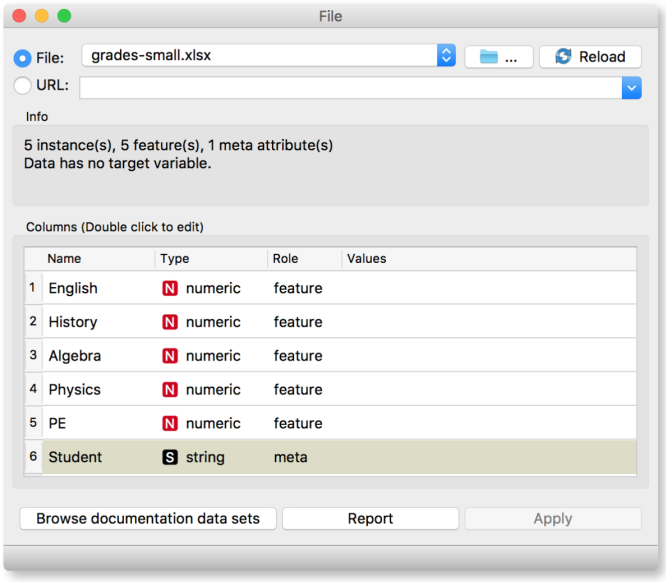
The data sets we have worked with in the previous lesson come with the Quasar installation. Quasar can read data from many file formats which include tab and comma separated and Excel files. To see how this works, let's prepare a data set (with school subjects and grades) in Excel and save it on a local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



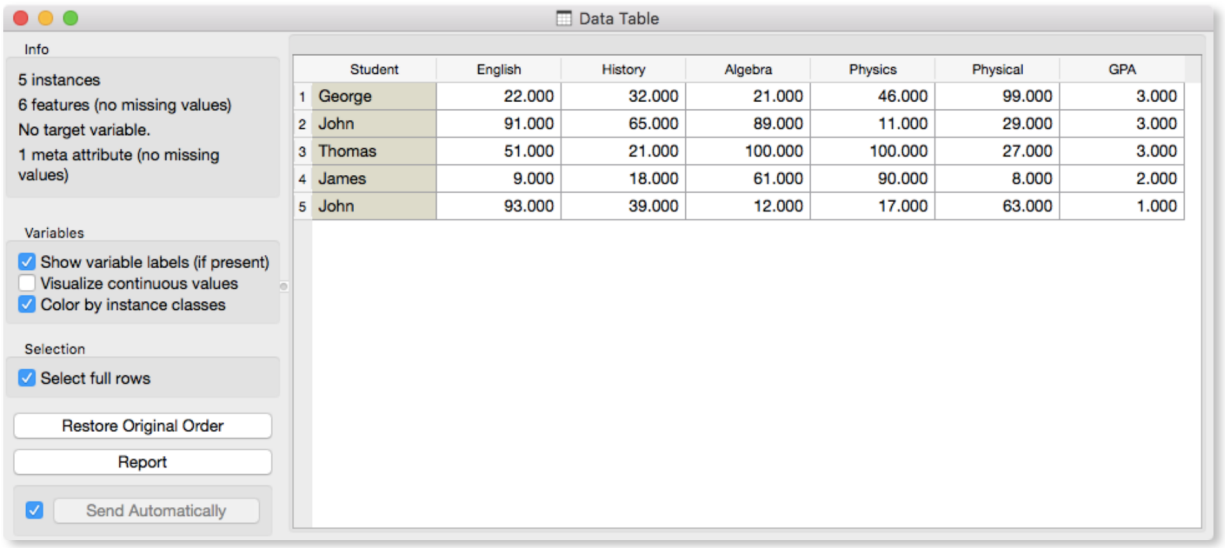

The *File* widget allows you to select a local file or even paste a URL to a Google Spreadsheet. In the Info box, you will see a quick summary about the data you loaded. By double clicking the fields, you can also edit the types of entries and their role, that will be relevant for further processing.

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,

Make the simple workflow shown on the right.


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




| | 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 |

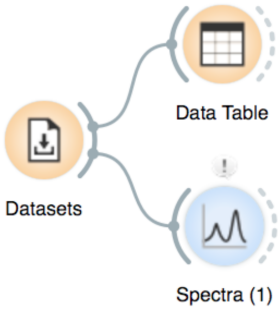
The *Data Table* widget shows the loaded data set, you can select rows, which will appear on the output of the widget. It is also possible to do simple data visualizations. Explore the functionalities!

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).

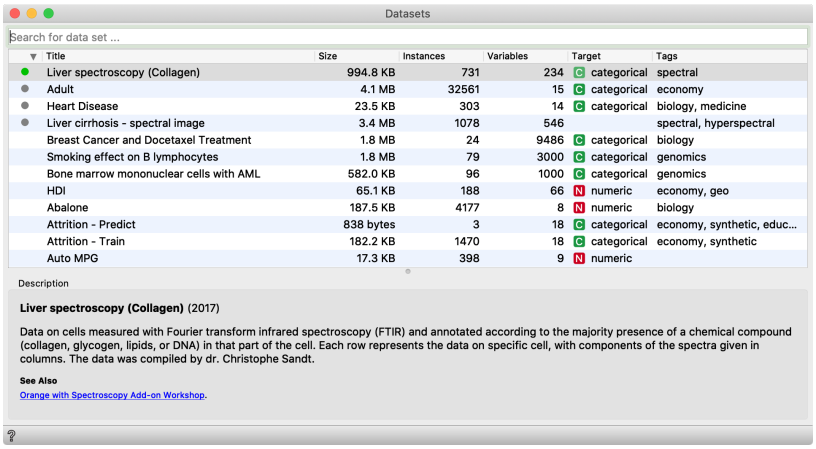
There is more to input data formatting and loading. If you would really like to dive in for more, check out the documentation page on Loading your Data, or a video tutorial on this subject.

# *Spectral data*



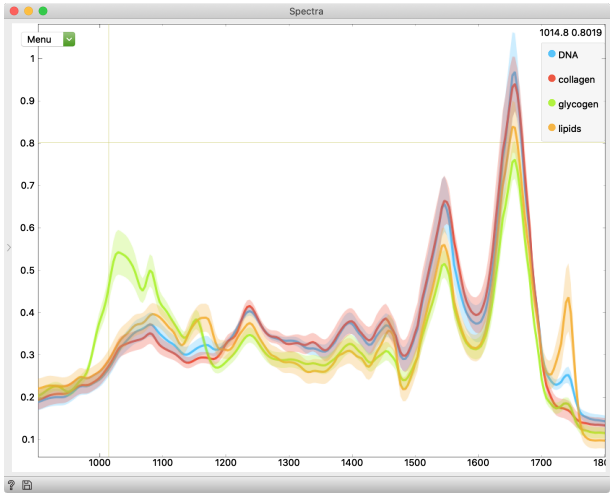

Your first spectroscopy workflow!

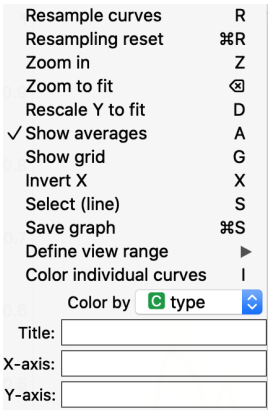
Let's make the small workflow shown on the right and open the "Liver spectroscopy" data set from Quasar's *Datasets* widget. In a *Data Table*, each row represents a spectrum. For the liver dataset, all columns, except the class column, describe absorbance at a specific wavenumber. Their column names 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

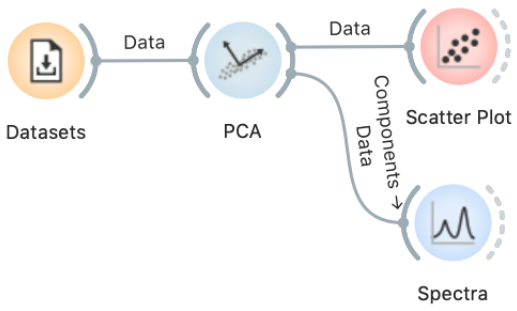
Connect the data to a *Spectra* widget from the **Spectroscopy** toolbox. To see the graph below, choose the feature for coloring in the top-left Menu (or click on the graph and press "c").



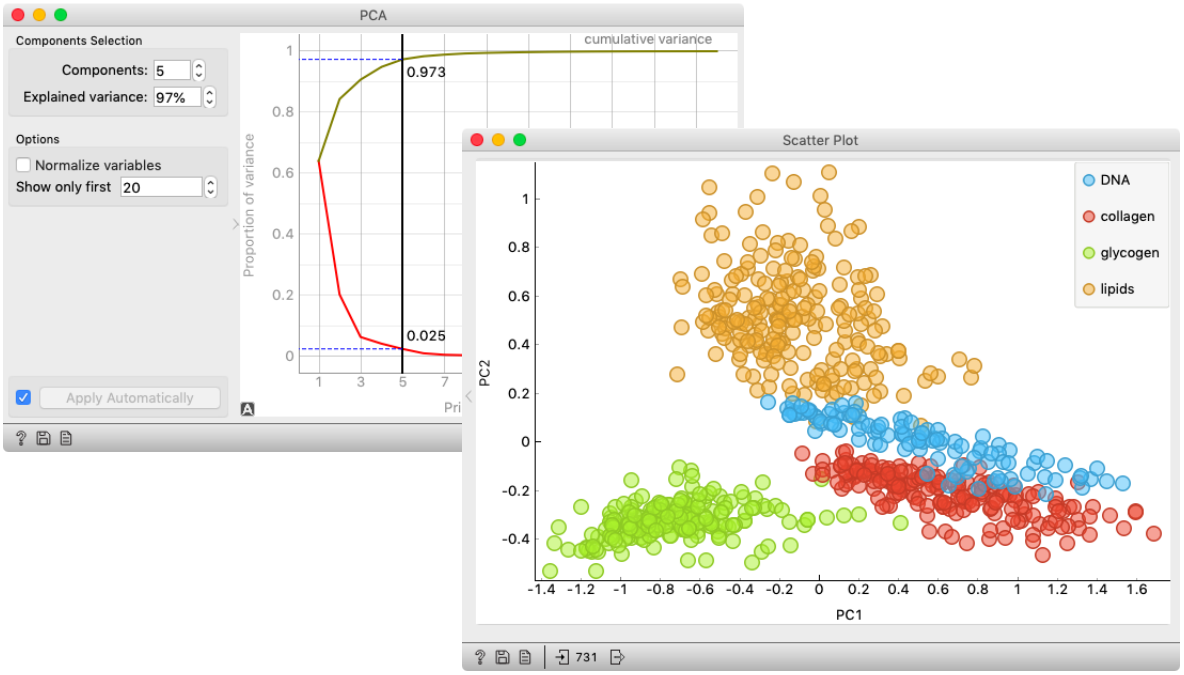

The *Spectra* widget and it's options. Try to use keyboard shortcuts on the right for frequent actions.

# *PCA on spectral data*



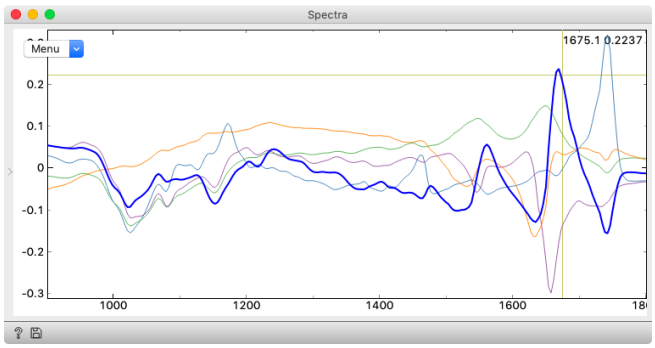

In this lesson we will explore the capabilities of Quasar for principal component analysis (PCA) on spectroscopy data. As usual, we will use the Liver Spectroscopy dataset. Connect *Datasets* to the *PCA* widget, choose the first 5 principal components and then connect *PCA*'s default output, "Data",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

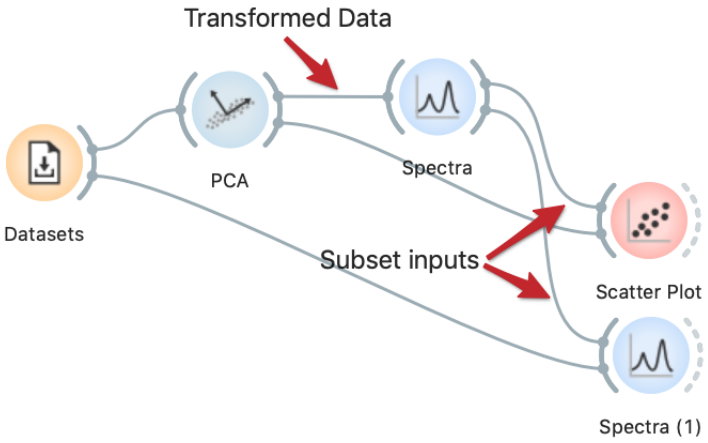
To see what different principal components represent, connect *PCA's* "Components" output (be careful, *PCA* has 4 outputs) into *Spectra*. Wondering which principal component is highlighted in the following screenshot? Wait for the tooltip...



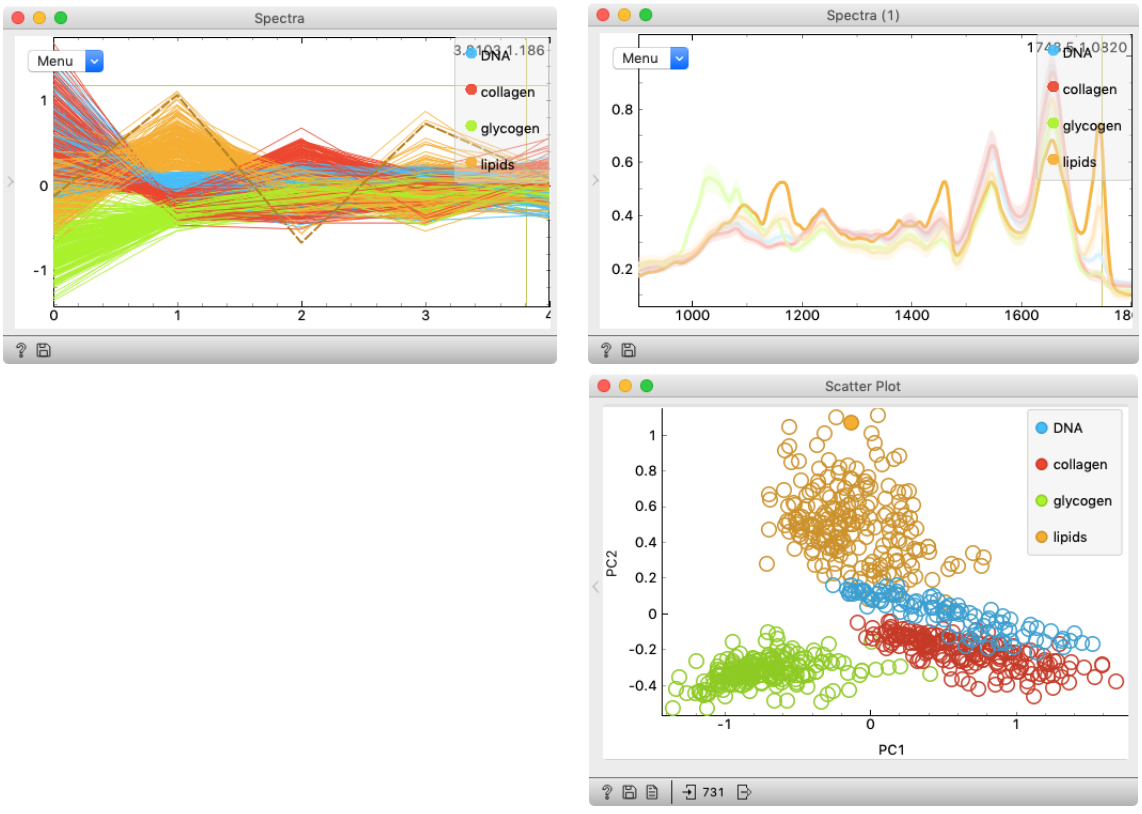

The curve under the cursor is highlighted. A tooltip will appear after some time. If clicked, the curve will be selected.

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.

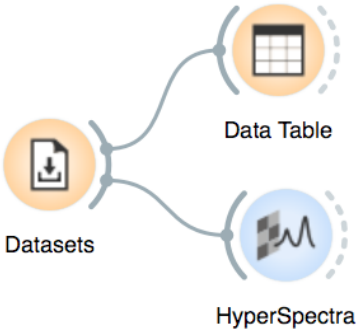
To find out more about a particular outlier, we can select it in the *Spectra* widget: move your mouse cursor to a curve—it will be highlighted—and click it. The selected curve changes to a dotted line and is sent to the output.




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.



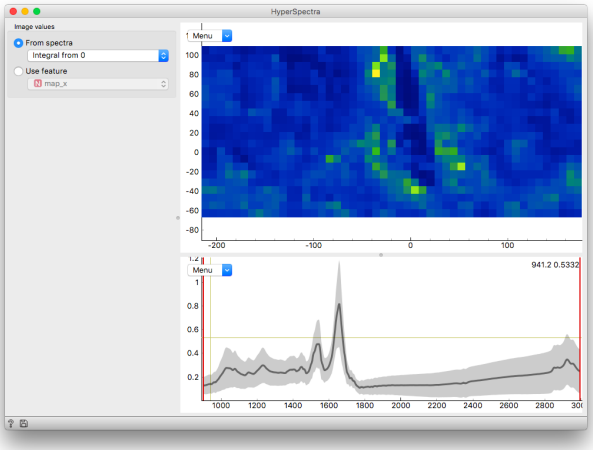

The selected spectrum's curve on the left is drawn with a dashed line and the corresponding original spectrum is highlighted on the right. The *Scatter Plot* shows the position of the selected spectrum in the PCA space.




# *Working with hyperspectral data*

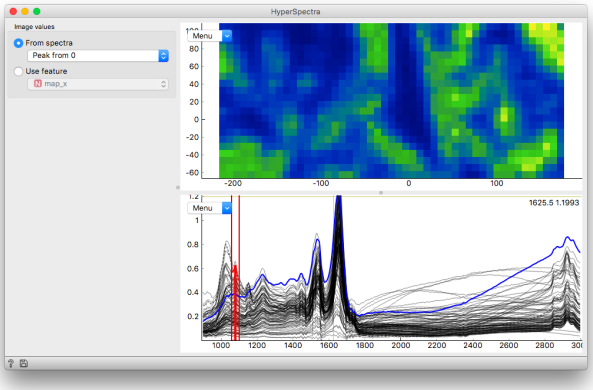
We can also visualize hyperspectral data sets. In the *Datasets* widget you will find "Liver cirrhosis" data. Connecting to a *Data Table*, you will see that each spectrum contains information (in meta variables) about image positions (map_x and map_y). Quasar can recognize the image positioning features 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.



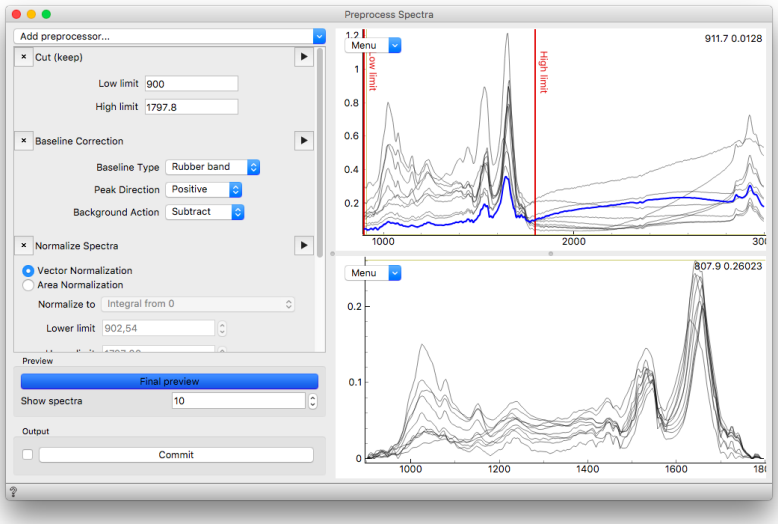

By default, the image is the 2D representation of the whole integral of each spectrum. To change it, move the red lines on the spectrum plot. With the dropdown menu on the left panel you can select other representations.

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



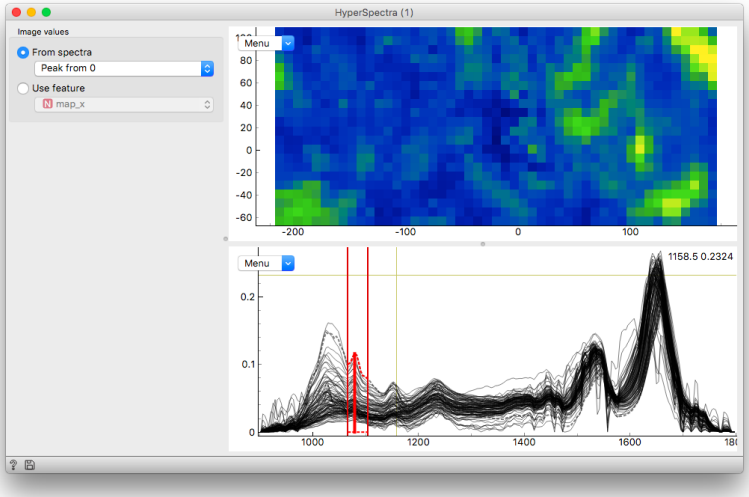
# *Preprocessing spectral data*


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.

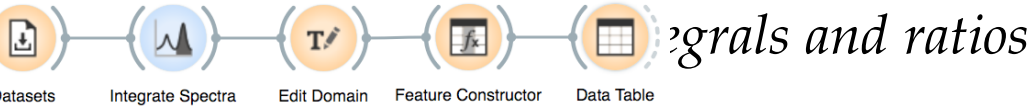



You can add preprocessing steps from the top dropdown menu of the left panel. Then, it is possible to drag them up and down to change their order. Each preprocessor has its own parameters. In the example here we show how the Baseline correction is done: you can simply change the baseline points by dragging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



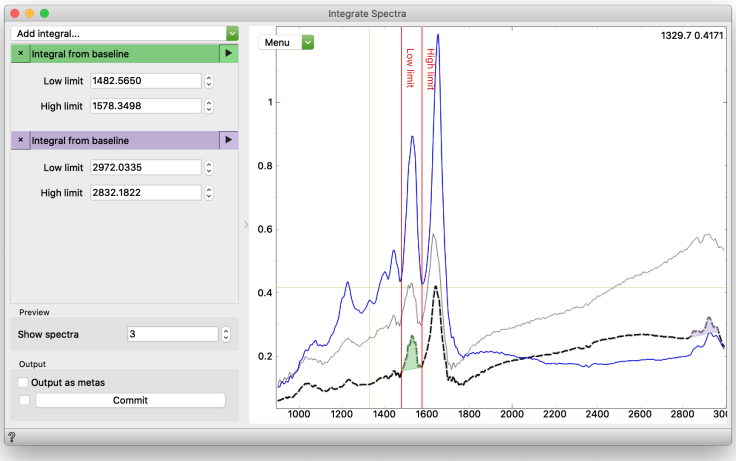

The preprocessed data in *HyperSpectra*. Did we gain anything? To investigate why the blue island disappeared, click on a pixel in it to see its spectrum.




# ...grals and ratios

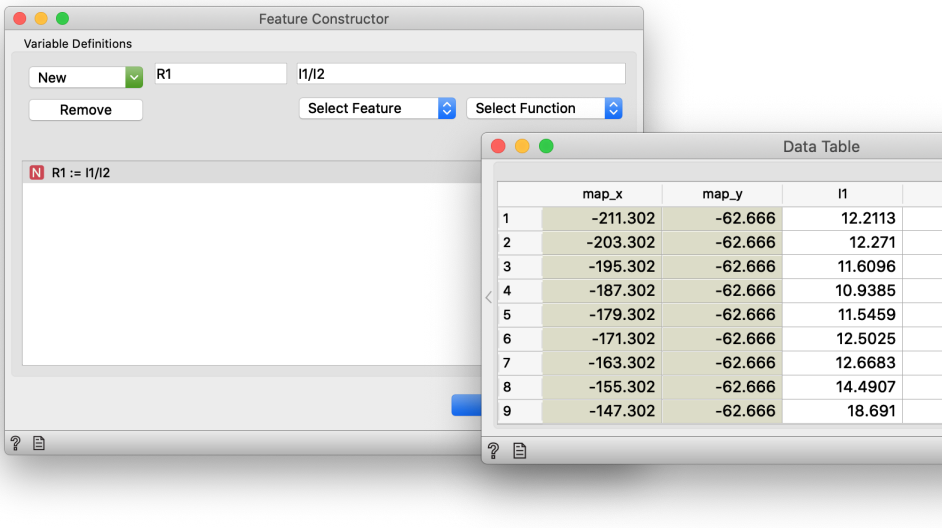
Peak integration is an essential element of spectroscopy for measuring concentrations, spectral contributions, etc.

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.

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



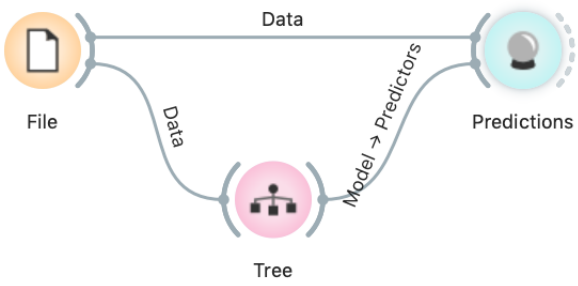

The produced data can be inspected by connecting other widgets.

# *Classification*

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.

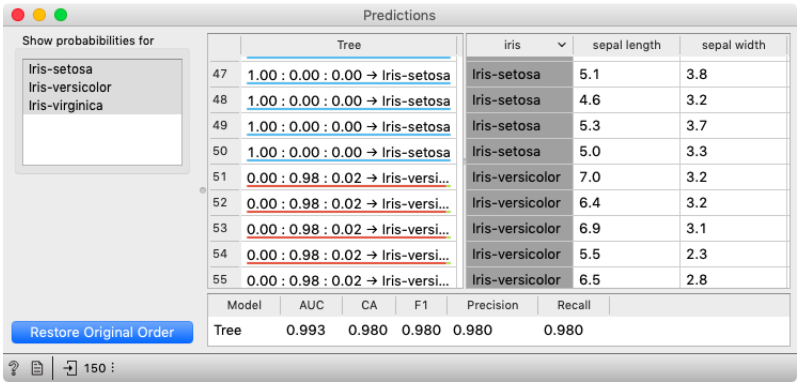
Let us create one now.

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




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

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?

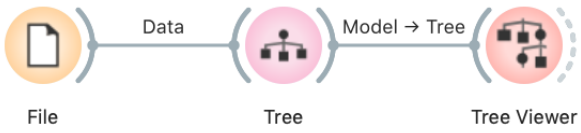
So many questions to answer!

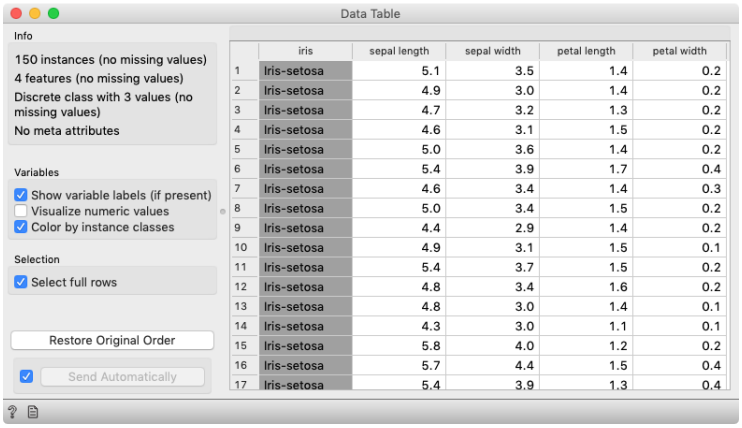
# *Classification Trees*

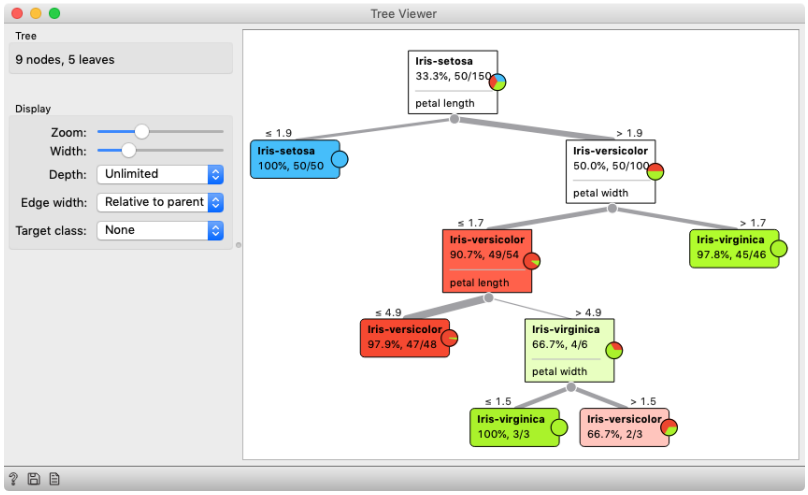
Classification trees were hugely popular in the early years of machine learning, when they were first independently proposed by the engineer Ross Quinlan (C4.5) and a group of statisticians (CART), including the father of random forests Leo Brieman.

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*.



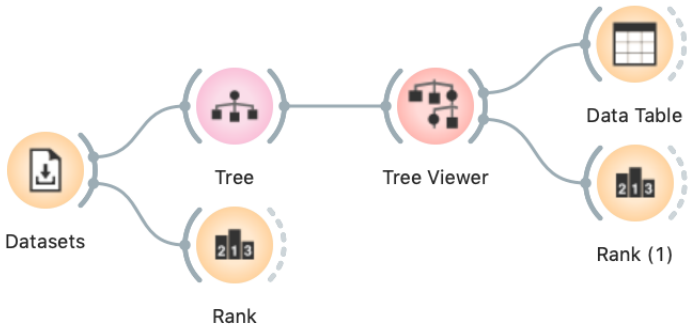

We read the tree from top to bottom. Looks like the column *petal length* best separates the iris variety *setosa* from the others, and in the next step, *petal width* then almost perfectly separates the remaining two varieties.

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).

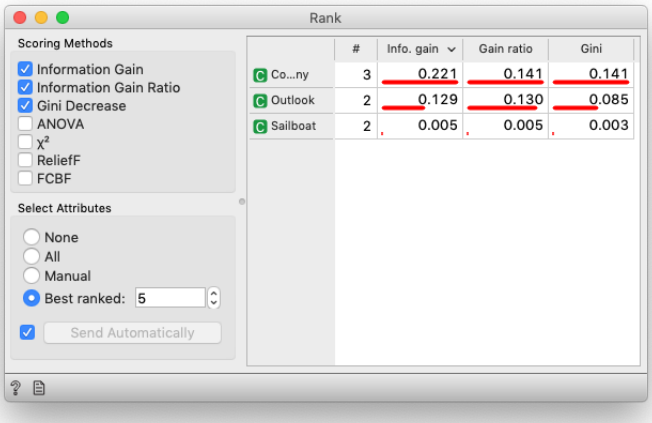
We still have not been very explicit about what we mean by "the most useful" feature. There are many ways to measure the quality of features, based on how well they distinguish between classes. We will illustrate the general idea with information gain. We can compute this measure in Orange using the *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*.

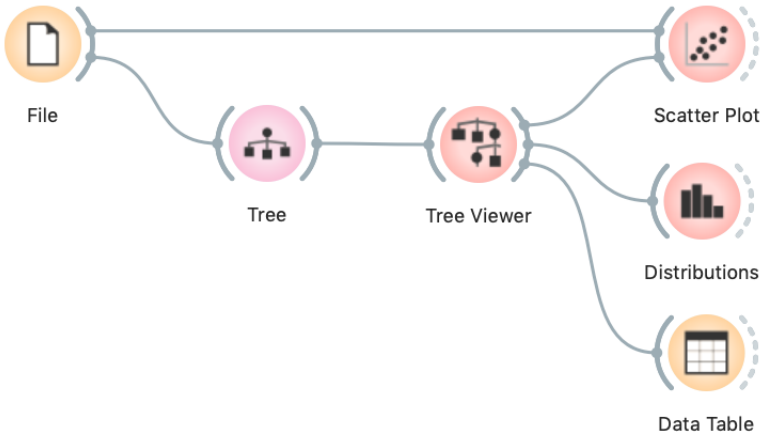
Besides the information gain, *Rank* displays several other measures (including Gain Ratio and Gini), which are often quite in agreement and were invented to better handle discrete features with many different values.



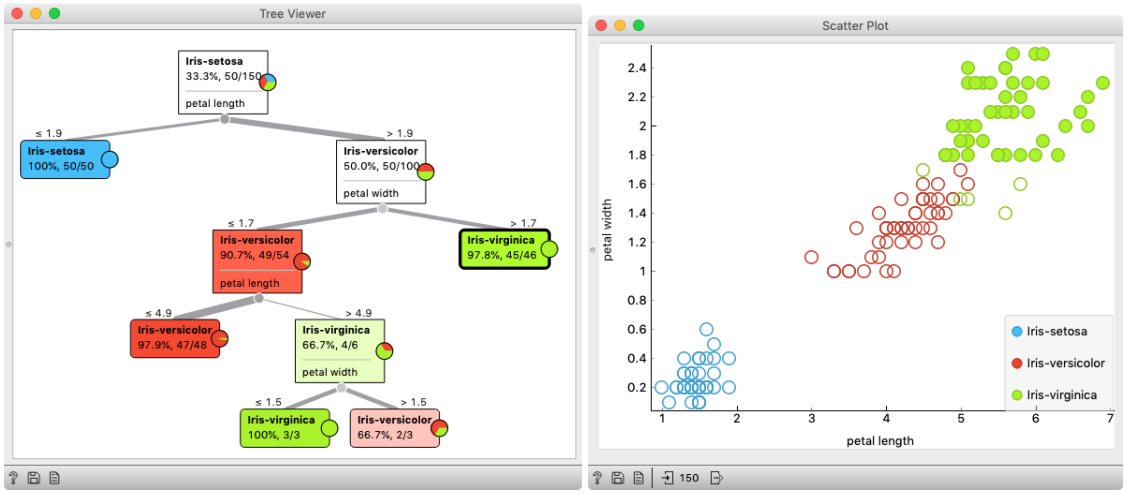

For the whole *Sailing* data set, *Company* is the most class-informative feature according to all measures shown.

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.

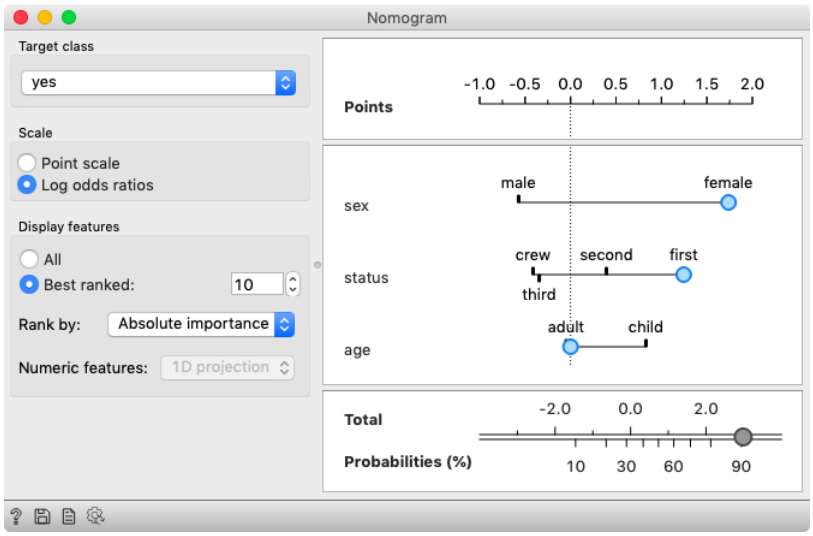
# *Naive Bayes*

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 se 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.

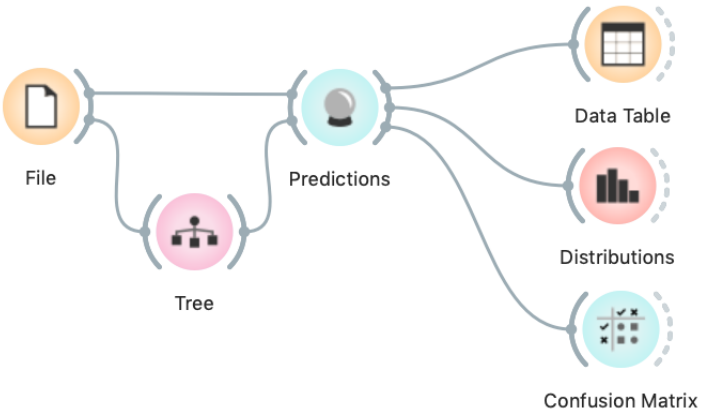



According to the probability theory individual contributions should be multiplied. Nomograms get around this by working in a log-space: a sum in the log-space is equivalent to multiplication in the original space. Therefore nomograms sum contributions (in the log-space) of all feature values and then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

# *Classification Accuracy*

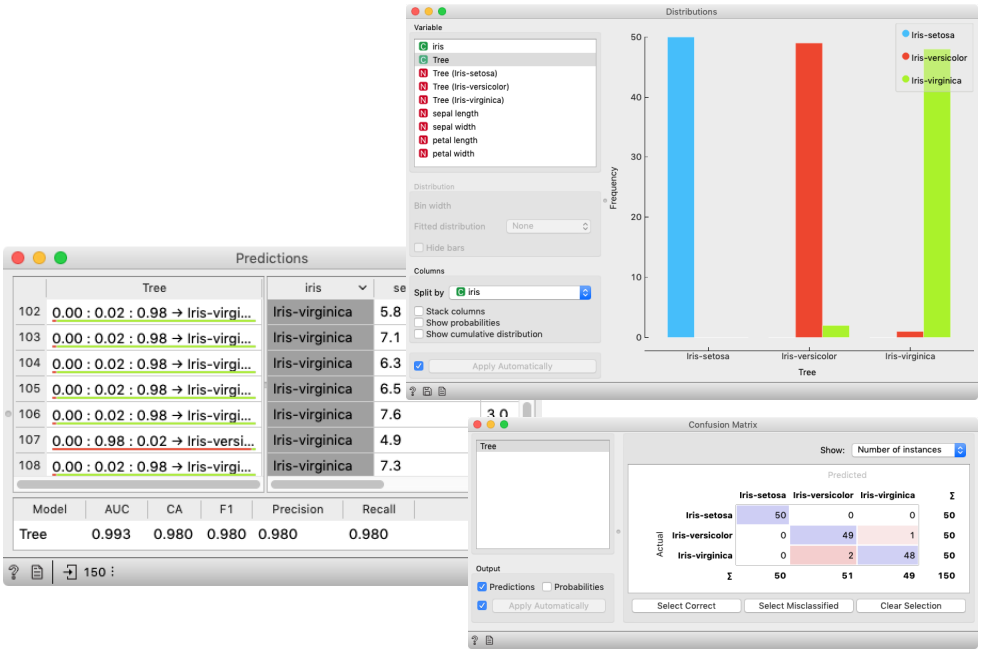
$$\text{accuracy} = \frac{\#\{\text{correct}\}}{\#\{\text{all}\}}$$

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.

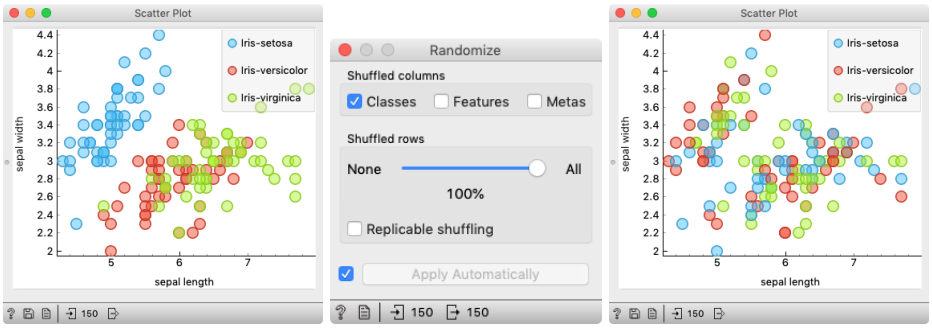



The *Confusion Matrix* shows 3 incorrectly classified examples, which makes the accuracy $(150 - 3)/150 = 98\%$.

# *How to Cheat*

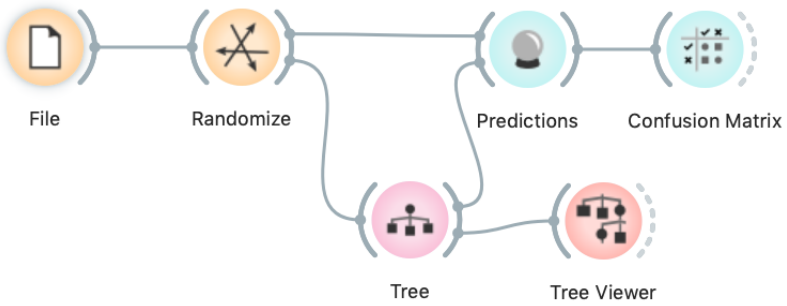
This lesson has a strange title and it is not obvious why it was chosen. Maybe you, the reader, should tell us what does this lesson have to do with cheating.

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!

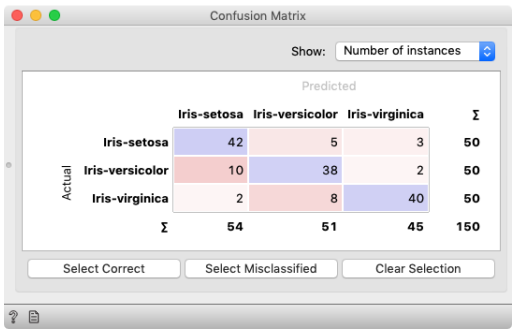

Left: scatter plot of the *Iris* data set before randomization; right: scatter plot after shuffling 100% of rows.

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



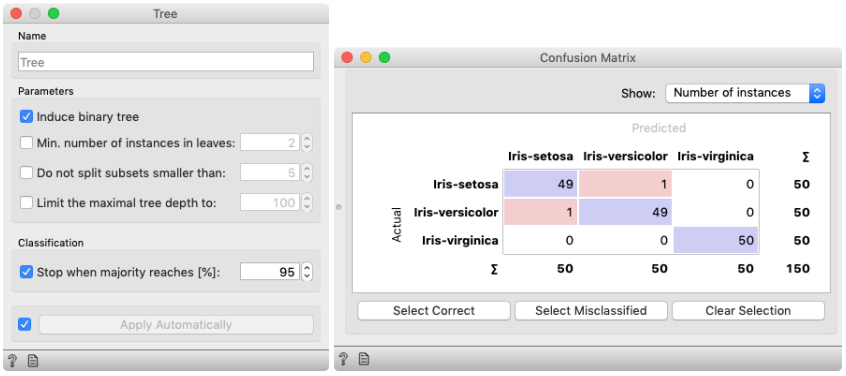

And the result? Here is a screenshot of the *Confusion Matrix*.

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



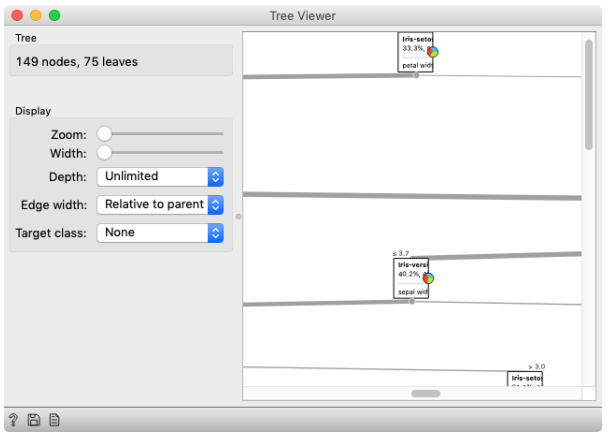
Can we further improve accuracy on the shuffled data? Let us try to change some properties of the induced trees: in the *Tree* widget, disable all early stopping criteria.

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?

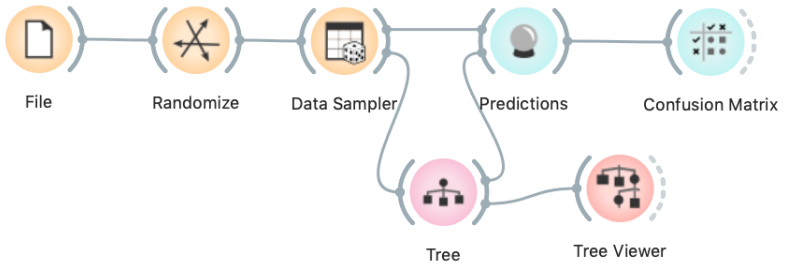



In the build tree, there are 75 leaves. Remember, there are only 150 rows in the *Iris* data set.

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.

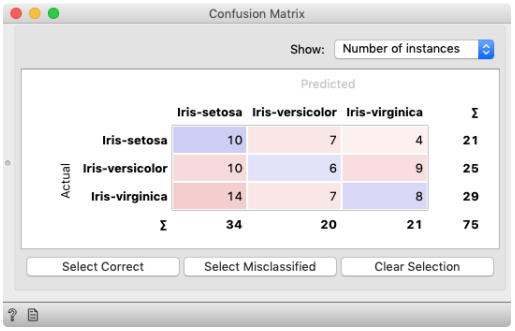
Connect the *Data Sampler* widget carefully. The *Data Sampler* splits the data to a sample and out-of-sample (so called remaining data). The sample was given to the *Tree* widget, while the remaining data was handed to the *Predictions* widget. Set the *Data Sampler* so that the size of these two data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

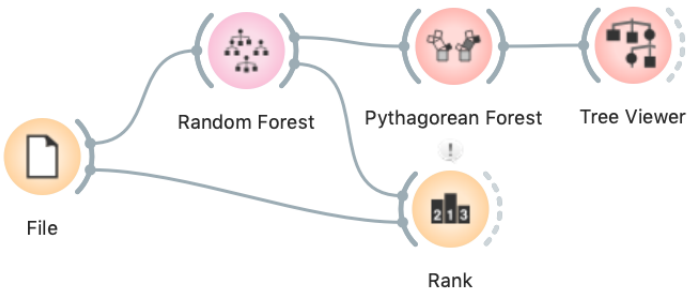



Confusion matrix if we estimate accuracy on a data set that was not used in learning.

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.

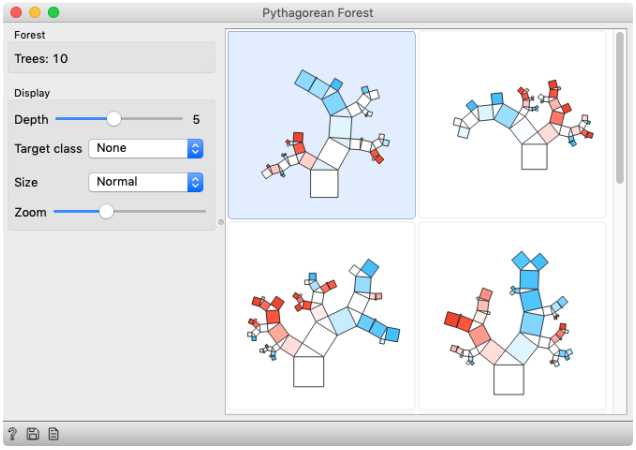
Learning is not only memorizing. Rather, it is discovering patterns that govern the data and apply to new data as well. To estimate the accuracy of a classifier, we therefore need a separate test set. This estimate should not depend on just one division of the input data set to training and test set (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 a 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.

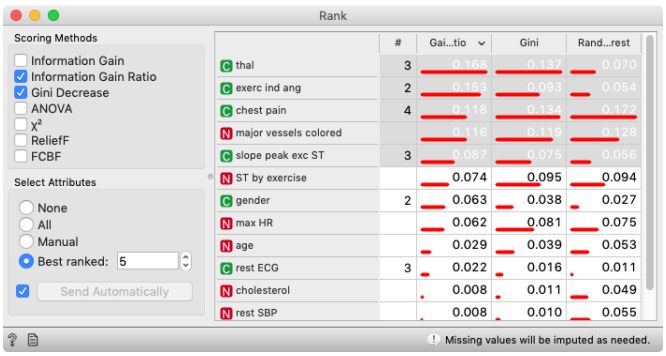
The *Pythagorean Forest* widget shows us how random the trees are. If we select a tree, we can observe it in a *Tree Viewer*.



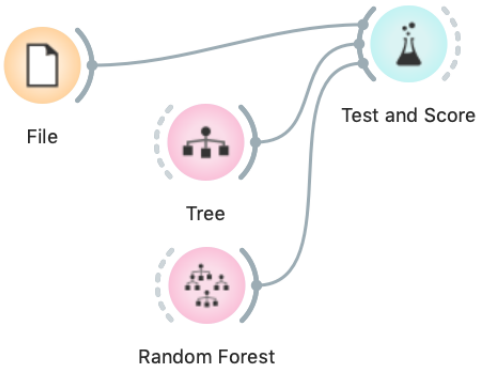

There are two sources of randomness: (1) training data is sampled with replacement, and (2) the best feature for a split is chosen among a subset of randomly chosen features.

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*

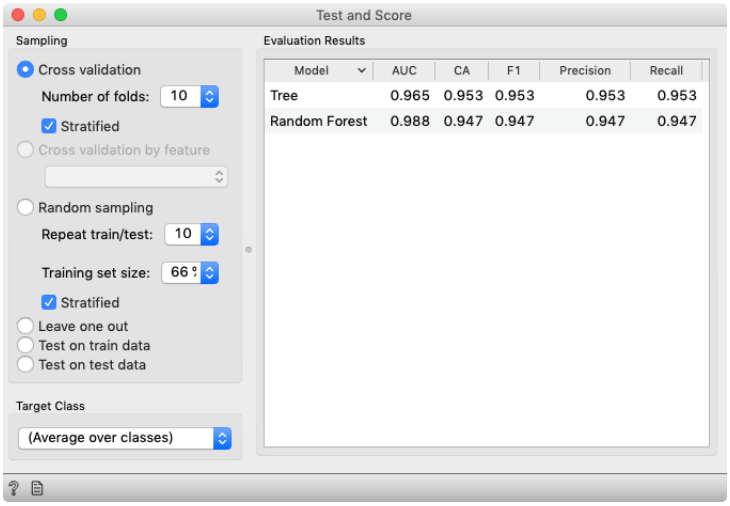



Estimating the accuracy may depend on a particular split of the data set. To increase robustness, we can repeat the measurement several times, each time choosing a different subset of the data for training. One such method is cross-validation. It is available in Orange through the *Test and Score* widget.

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.

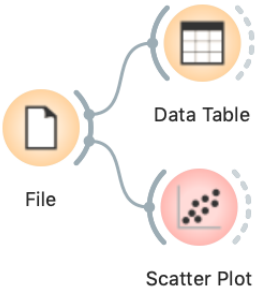
This is another way to use the *Tree* widget. In the workflows from the previous lessons we have used another of its outputs, called *Model*; its construction required data. This time, no data is needed for *Tree*, because all that we need from it is a *Learner*.




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.

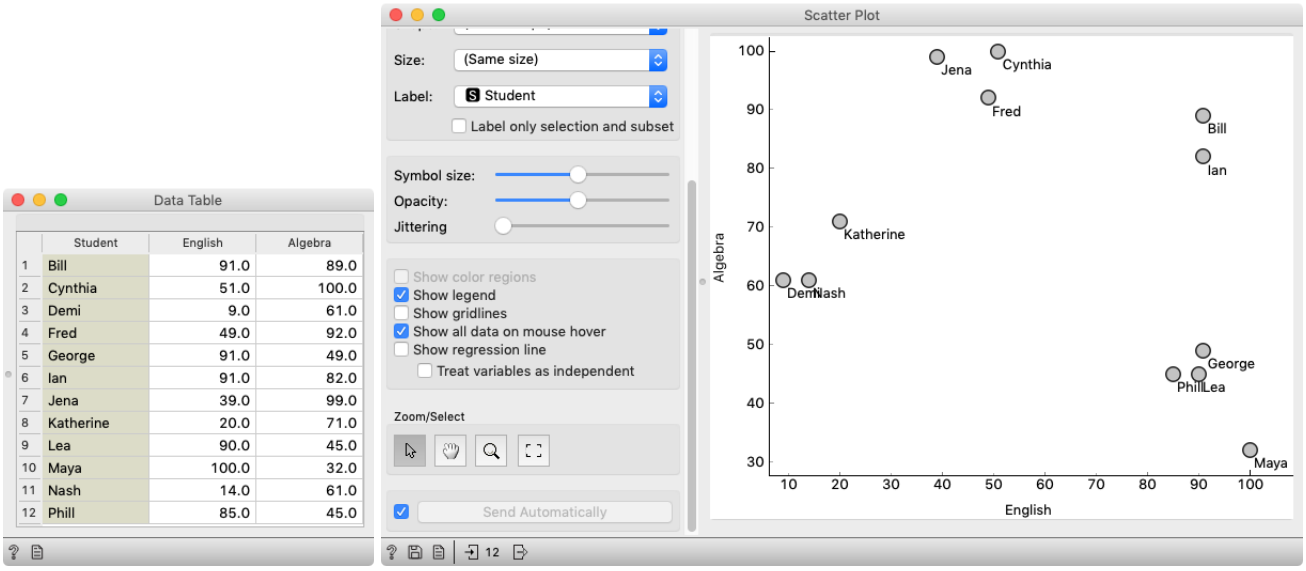
# *Hierarchical Clustering*




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`.

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

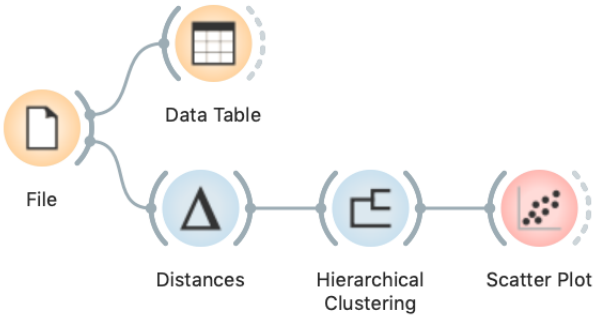
First, we need to define what we mean by "similar". We will assume that all our data instances are described (profiled) with continuous features. One simple measure of similarity is the Euclidean distance. So, we would like to group data instances with small Euclidean distances.

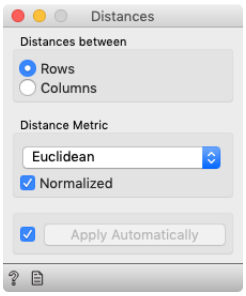



There are different ways to measure the similarity between clusters. The estimate we have described is called average linkage. We could also estimate the distance through the two closest points in each clusters (single linkage), or through the two points that are furthest away (complete linkage).

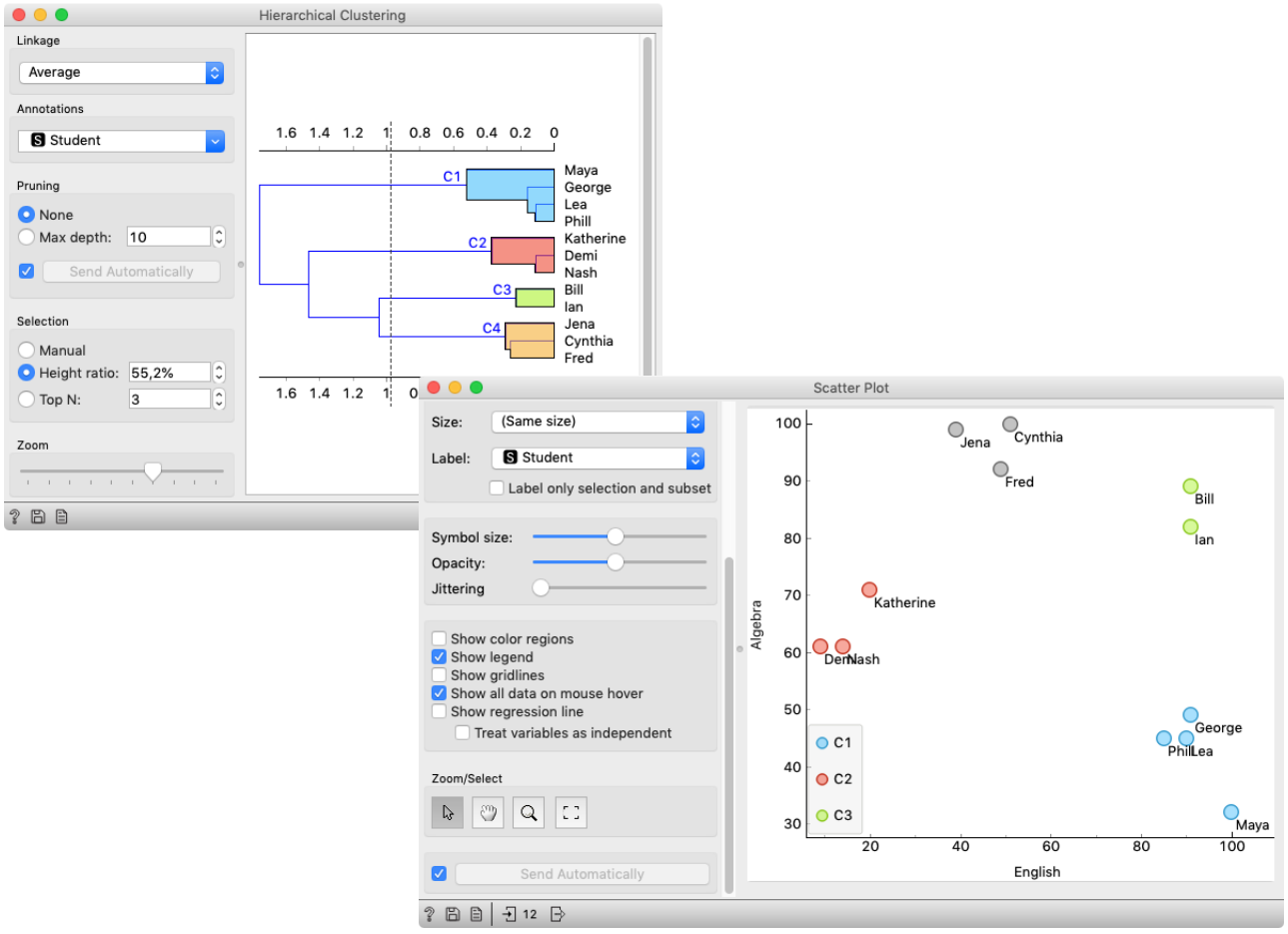
Next, we need to define a clustering algorithm. Say that we start with each data instance being its own cluster, and then, at each step, we join the clusters that are closest together. We estimate the distance between the clusters with, say, the average distance between all their pairs of data points.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

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



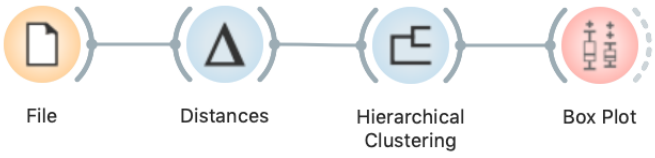

Couldn't be simpler. Load the data, measure the distances, use them in hierarchical clustering, and visualize the results in a scatter plot. The *Hierarchical Clustering* widget allows us to cut the hierarchy at a certain distance score and output the corresponding clusters:




# *Animal Kingdom*

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

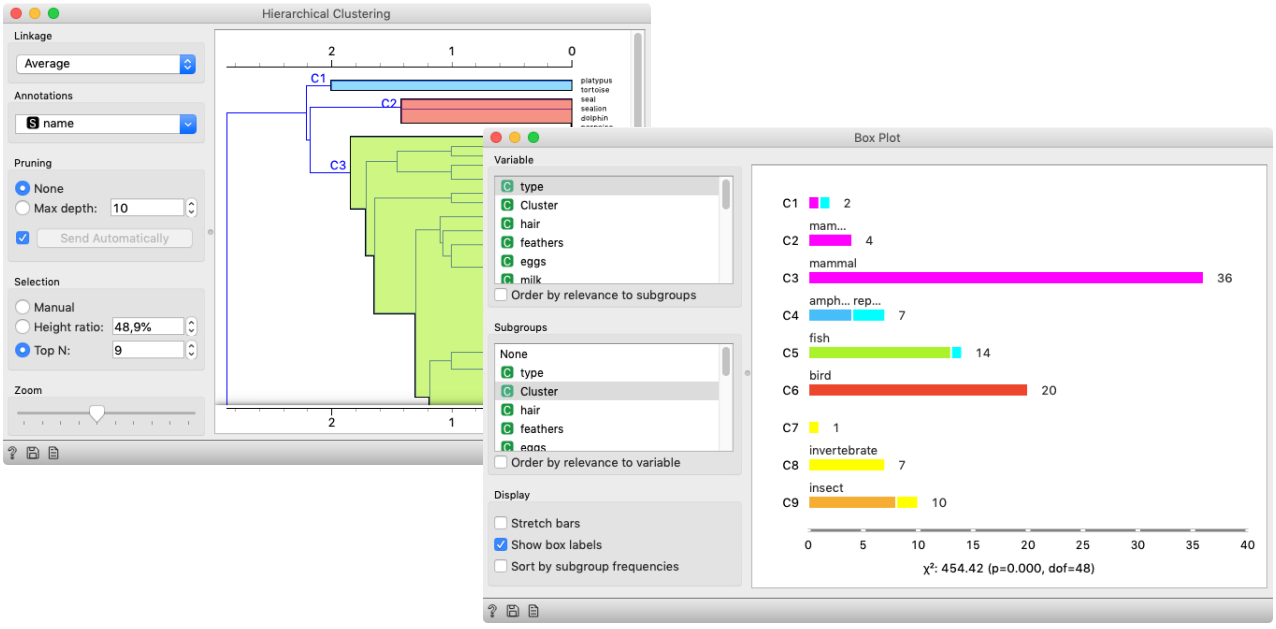
Funny stuff was we never understood the order in which the cards were laid out in the album. We later learned about taxonomy, but being more inclined to engineering we never mastered learning it in our biology classes. Luckily, there's data mining and the idea that taxonomy simply stems from measuring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




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

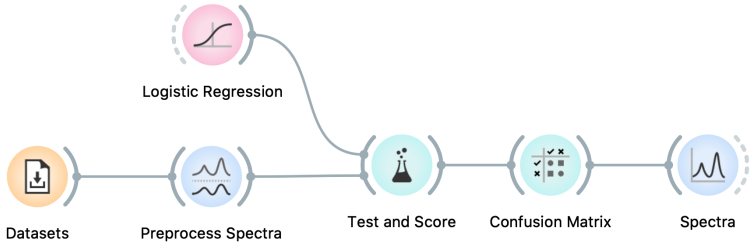
Here we use zoo data (from the documentation data sets) with attributes that report on various features of animals (has hair, has feathers, lays eggs). We measure the distance and compute the clustering. Animals in this data set are annotated with type (mammal, insect, bird, and so on). It would be cool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

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

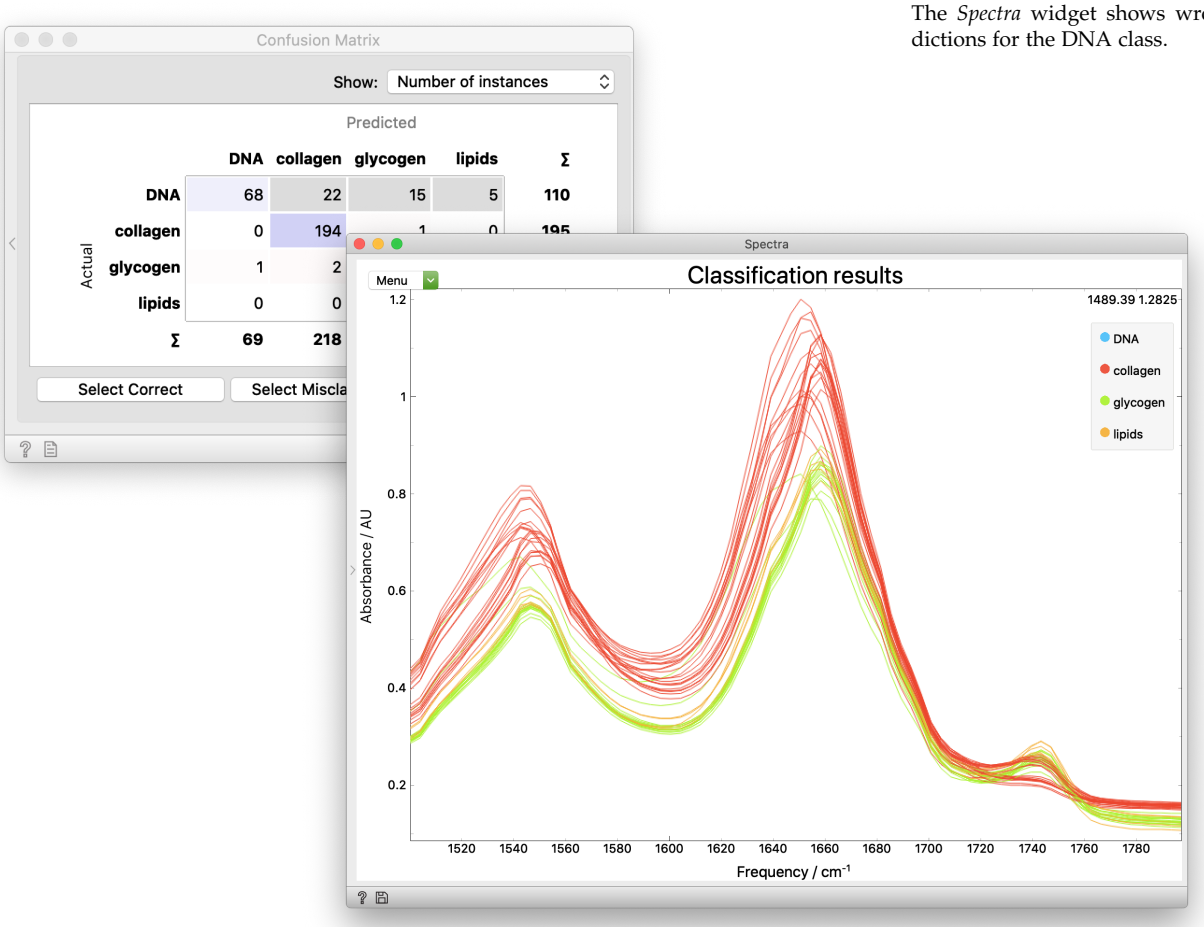



What is wrong with those mammals? Why can't they be in one single cluster? Two reasons. First, they represent 40% of the data instances. Second, they include some weirdos. Who are they?

# *Classification of Spectra*

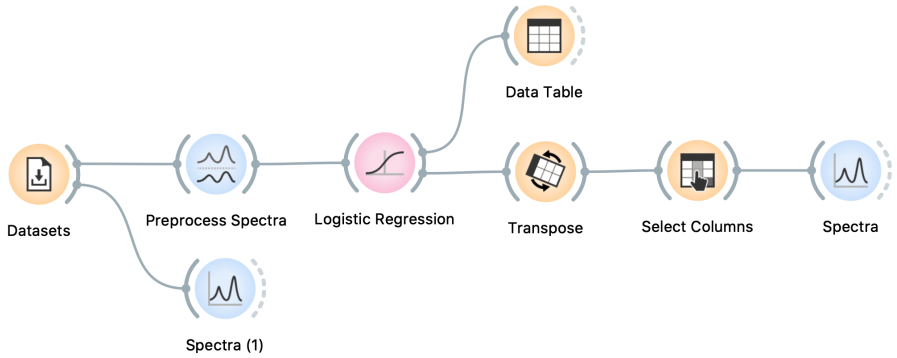



Let's open the collagen data set again and see how well can logistic regression predict its four classes. Straightforward, right? Connect *Datasets*, *Logistic Regression*, *Predictions*, *Confusion Matrix* and that's it. We would also like to do some spectral processing (we will only keep the columns for wavenumbers between 1500 $cm^{-1}$and 1800 $cm^{-1}$).




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

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).



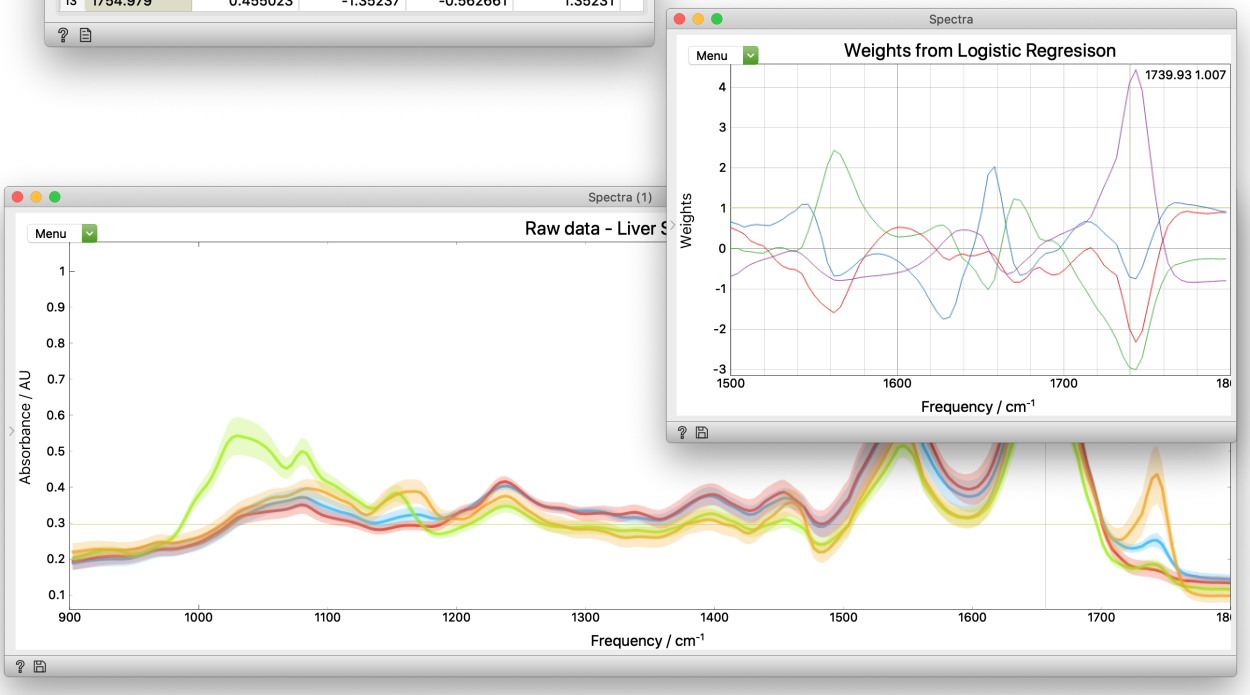

But how does the model make its decisions? We already inspected a different model, classification tree, where each node represents a decision on a value of a column. *Logistic regression* works differently. On the training data it computes weights for all columns (wavelengths), which are then used for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

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.

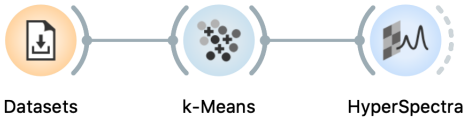
Now, open *Logistic Regression* and try changing its parameters. Observe the effect on the weights.

Data Table

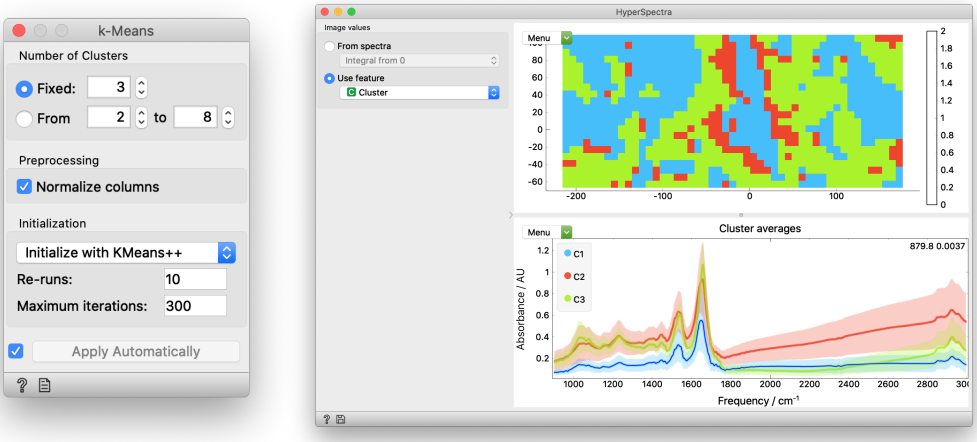
| | name | DNA | collagen | glycogen | lipids |
|---|---|---|---|---|---|
| 1 | intercept | -4.9939 | -5.37055 | 10.0245 | -1.24602 |
| 2 | 1797.407 | 0.909034 | -0.256598 | 0.892044 | -0.799088 |
| 3 | 1793.55 | 0.936854 | -0.263577 | 0.89339 | -0.803748 |
| 4 | 1789.693 | 0.981421 | -0.253348 | 0.875729 | -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 |



# *Clustering Spectral Images*

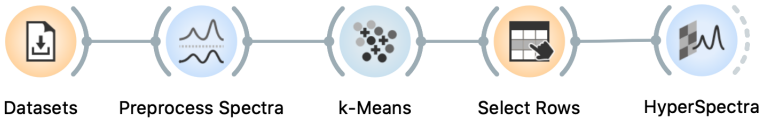


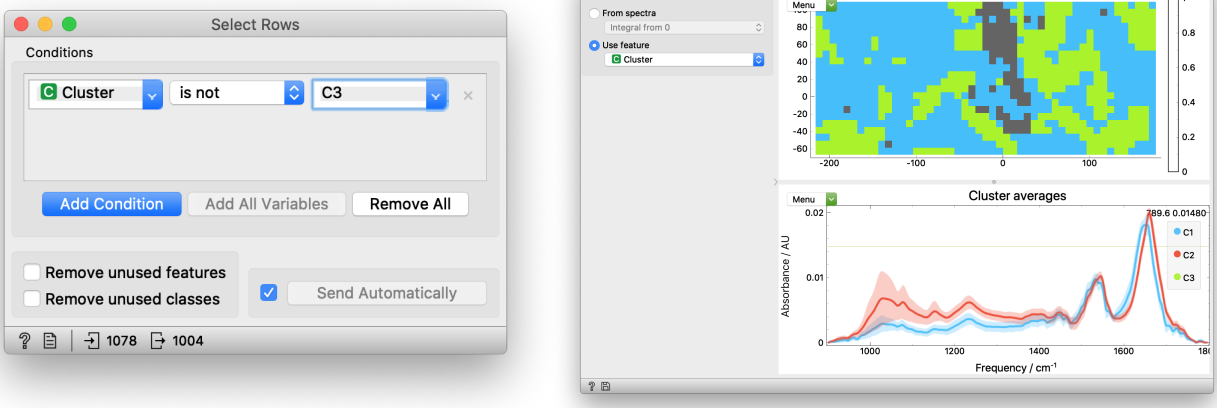

We have already seen hierarchical clustering. Another clustering algorithm, k-Means, is much faster for data with lots of rows, like images, which contain a row (a spectrum) for each pixel. Still, for the liver-cirrhosis data, both approaches are fast. Here, we use *k-Means* with k=3 clusters.




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.



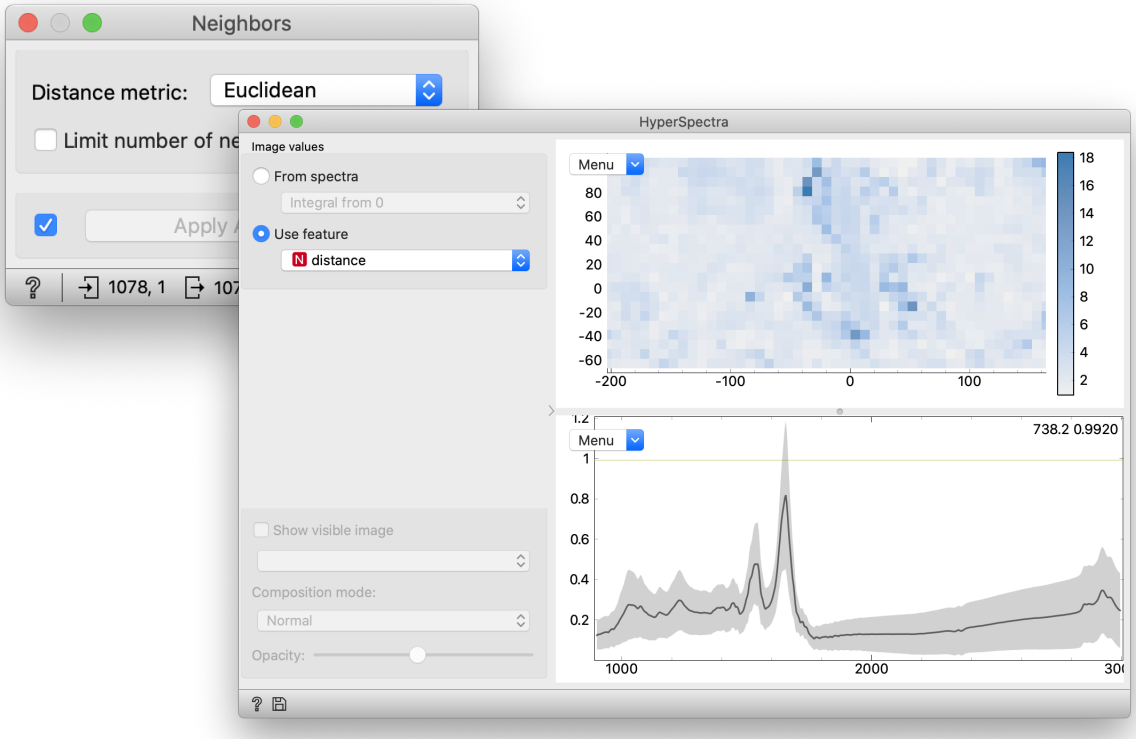

The *Spectra* widget shows wrong predictions for the DNA class.

# *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!

If you build the workflow shown above this paragraph in Quasar you will be able to explore various distance metrics. First, let's use the *'Liver cirrhosis - spectral image'* dataset provided by the *Datasets* widget, then calculate the *Euclidean distances* from the average spectrum with the *Neighbors* widget and 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)

# *Bibliography*

# *Index*