

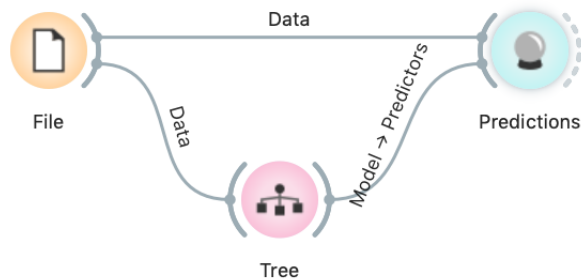
Classification

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

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

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

Let us create one now.



	iris	sepal length	sepal width
47	Iris-setosa	5.1	3.8
48	Iris-setosa	4.6	3.2
49	Iris-setosa	5.3	3.7
50	Iris-setosa	5.0	3.3
51	Iris-versicolor	7.0	3.2
52	Iris-versicolor	6.4	3.2
53	Iris-versicolor	6.9	3.1
54	Iris-versicolor	5.5	2.3
55	Iris-versicolor	6.5	2.8

Model	AUC	CA	F1	Precision	Recall
Tree	0.993	0.980	0.980	0.980	0.980

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

The *Predictions* widget also receives the data from the *File* widget. The widget uses the model to make predictions about the data

and shows them in the table.

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

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

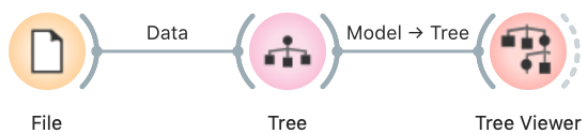
So many questions to answer!

Classification Trees

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

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

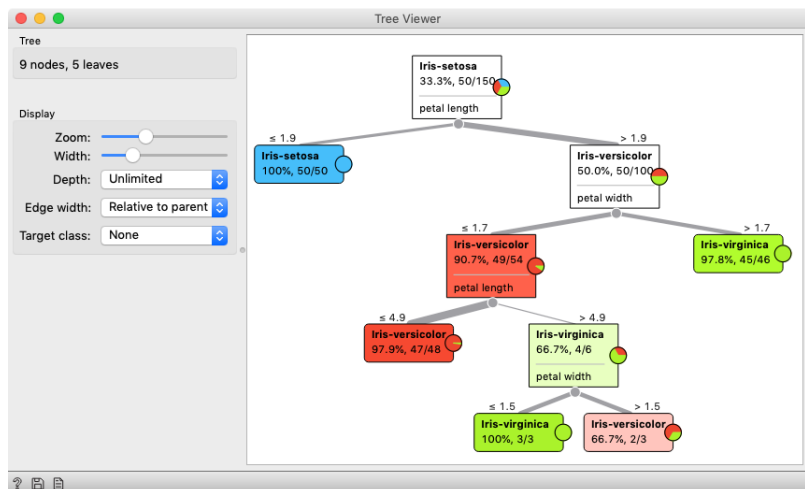
Classification trees were hugely popular in the early years of machine learning, when they were first independently proposed by the engineer Ross Quinlan (C4.5) and a group of statisticians (CART), including the father of random forests Leo Breiman.



	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



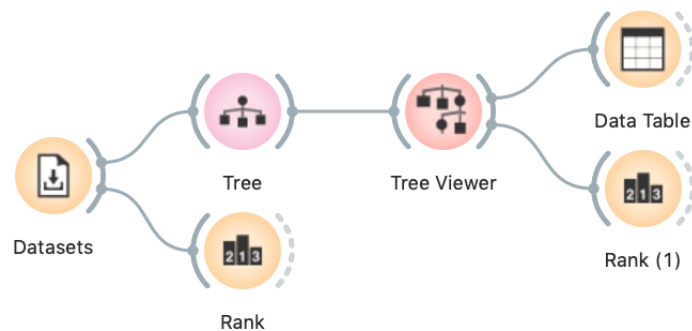
of

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*.

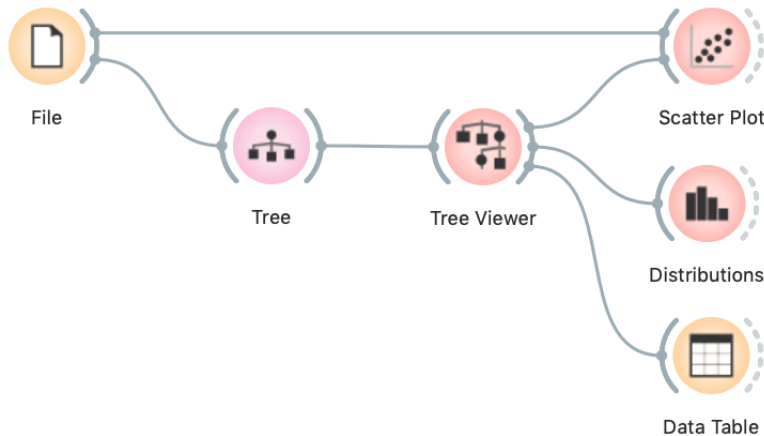


Besides the information gain, *Rank* displays several other measures (including Gain Ratio and Gini), which are often quite in agreement and were invented to better handle discrete features with many different values.

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

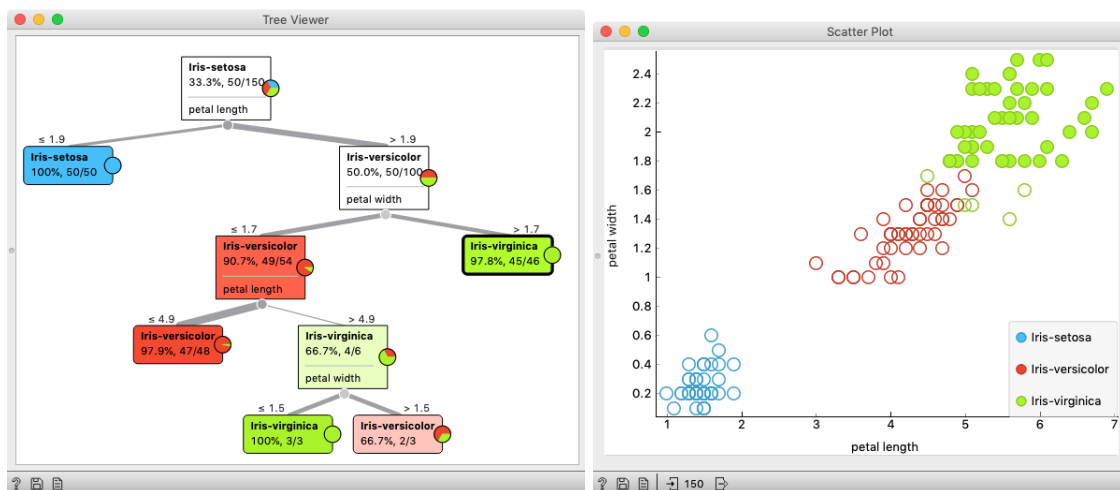
	#	Info. gain	Gain ratio	Gini
Co...ny	3	0.221	0.141	0.141
Outlook	2	0.129	0.130	0.085
Sailboat	2	0.005	0.005	0.003

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



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

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

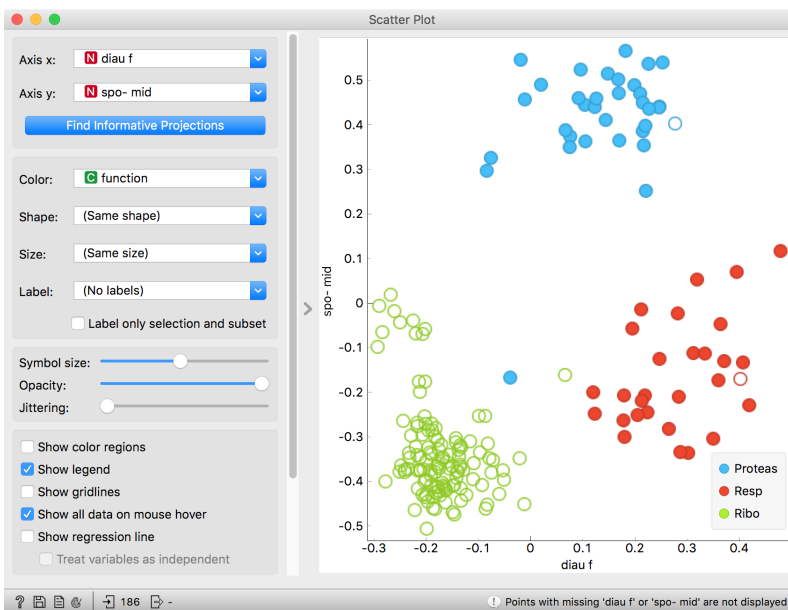
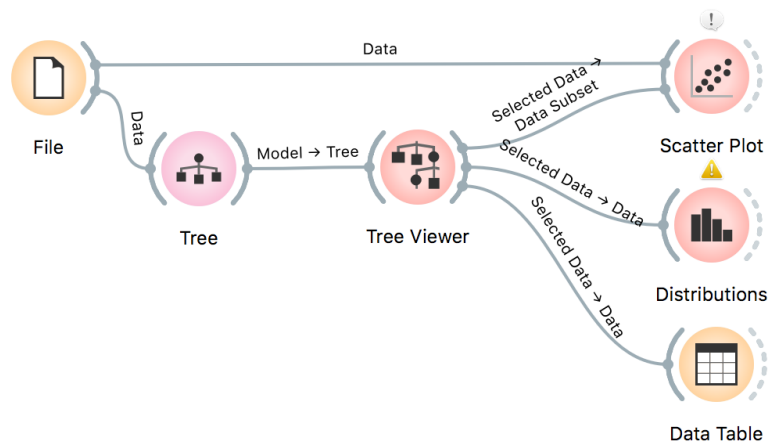


Wherever possible, visualizations in Orange are designed to support selection and passing of the data that applies to it. Finding interesting data subsets and analyzing their commonalities is a central part of explorative data analysis, a data analysis approach favored by the data visualization guru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

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

Model Inspection

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



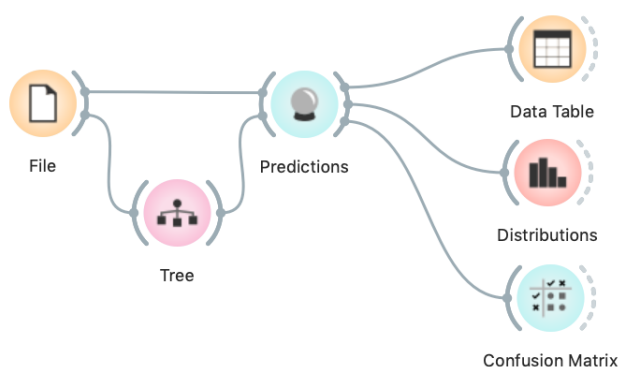
Just for fun, we have included a few other widgets in this workflow. The Tree Viewer selects data instances by inferring rules from the data itself and optimizing to obtain purer data subsets.

Classification Accuracy

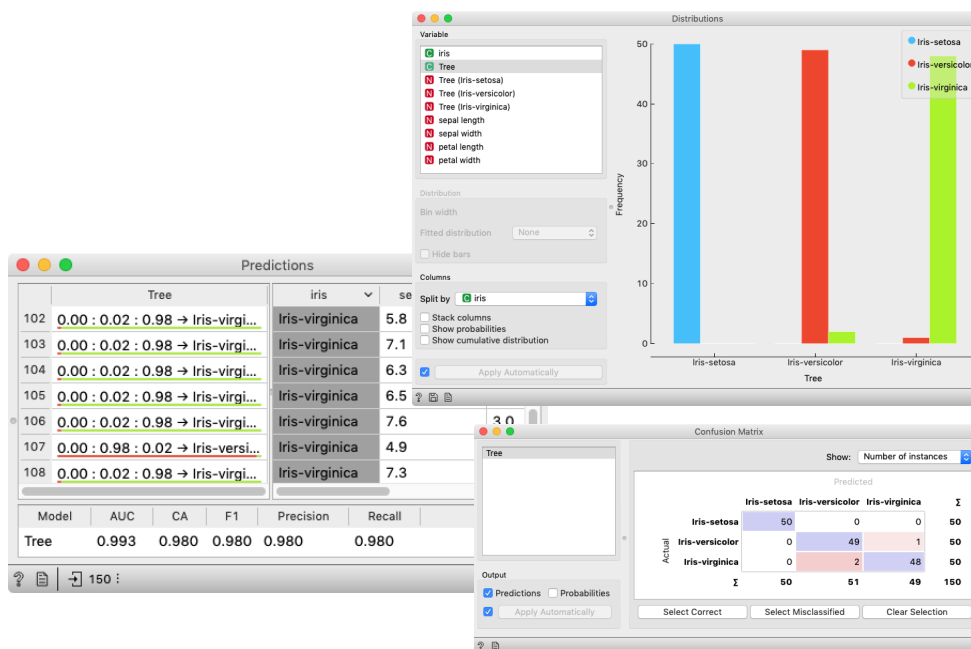
Now that we know what classification trees are, the next question is what is the quality of their predictions. For beginning, we need to define what we mean by quality. In classification, the simplest measure of quality is classification accuracy expressed as the proportion of data instances for which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

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

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



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



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

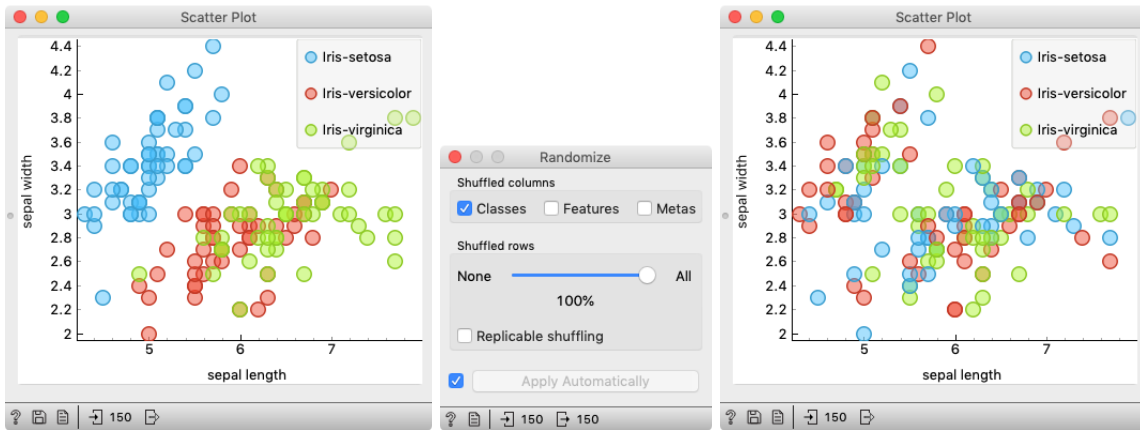
How to Cheat

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



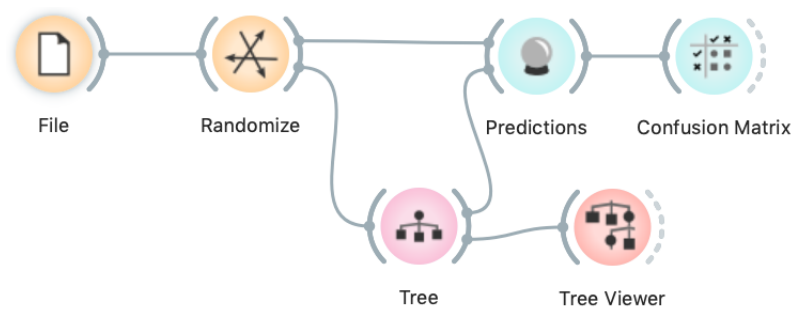
At this stage, the classification tree looks very good. There's only one data point where it makes a mistake. Can we mess up the data set so bad that the trees will ultimately fail? Like, remove any existing correlation between features and the class? We

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



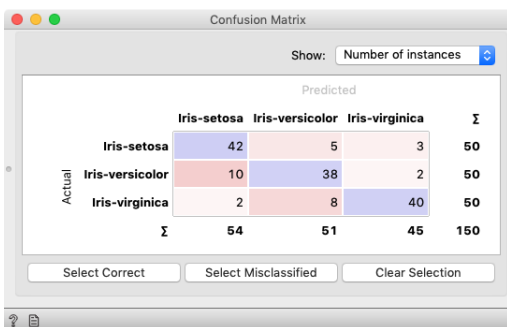
Left: scatter plot of the Iris data set before randomization; right: scatter plot after shuffling 100% of rows.

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

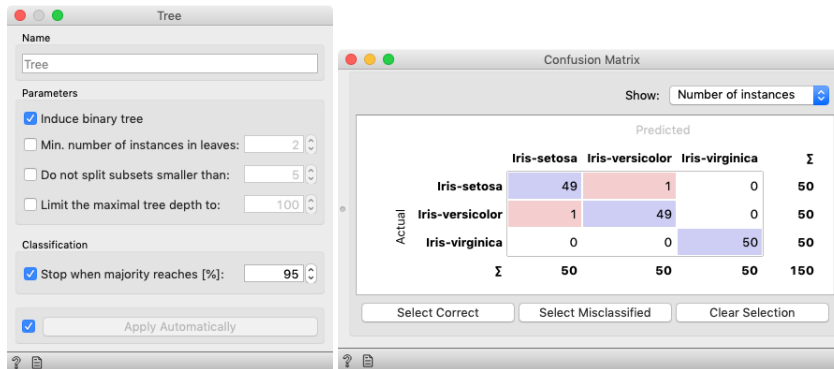


And the result? Here is a screenshot of the *Confusion Matrix*.

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



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



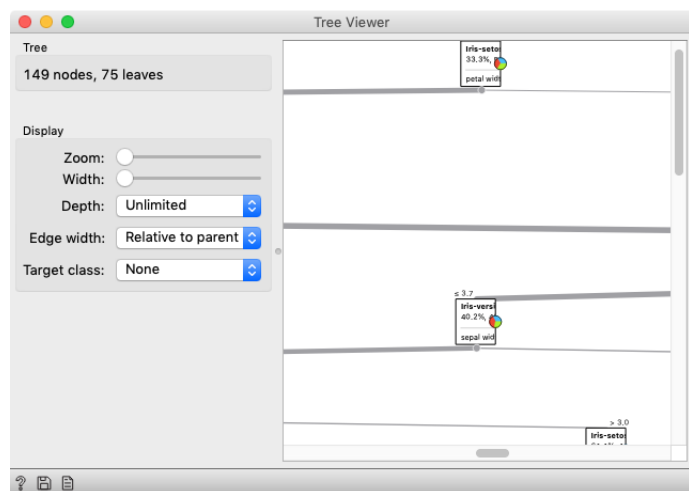
After we disable 2–4 check box in the *Tree* widget, our classifier starts behaving almost perfectly.

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

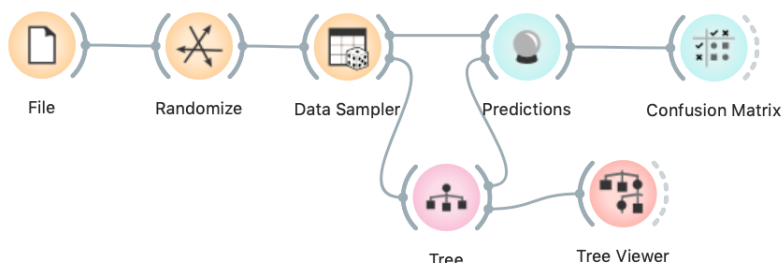
To find the answer to this riddle, open the *Tree Viewer* and check out the tree. How many nodes does it have? Are there many data instances in the leaf nodes?

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

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



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



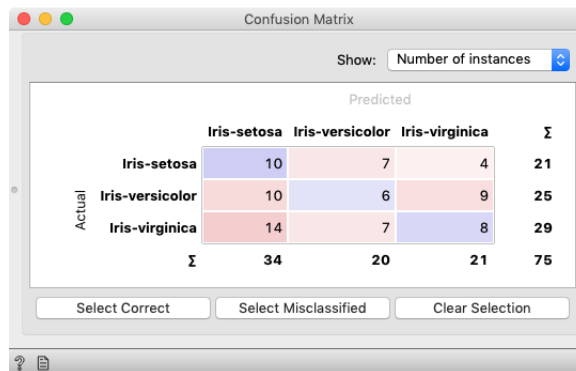
Connect the *Data Sampler* widget carefully. The *Data Sampler* splits the data to a sample and out-of-sample (so called remaining data). The sample was given to the *Tree* widget, while the remaining data was handed to the *Predictions* widget. Set the *Data Sampler* so that the size of these two data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

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

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

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

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



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

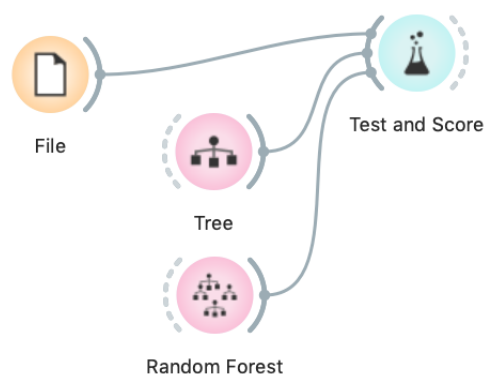
Learning is not only memorizing. Rather, it is discovering patterns that govern the data and apply to new data as well. To estimate the accuracy of a classifier, we therefore need a separate test set. This estimate should not depend on just one division of the input data set to training and test set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

Cross-Validation

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

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

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



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

Model	AUC	CA	F1	Precision	Recall
Tree	0.965	0.953	0.953	0.953	0.953
Random Forest	0.988	0.947	0.947	0.947	0.947

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.