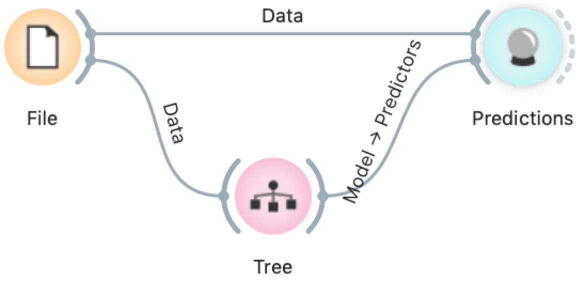
Classification

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.

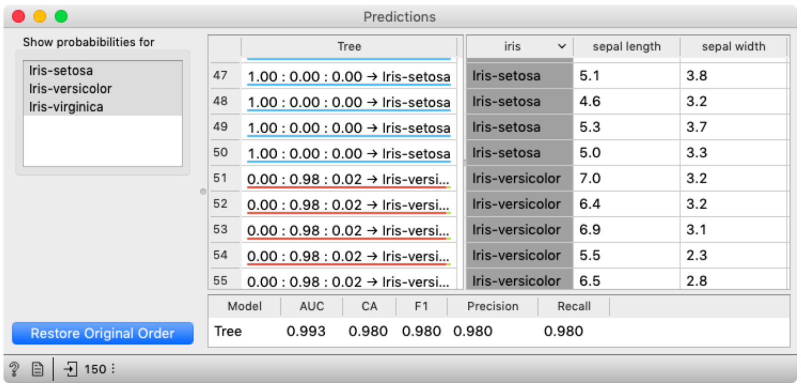
Let us create one now.

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.



Predictions

Show probabilities for

Iris-setosa
Iris-versicolor
Iris-virginica

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

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

Restore Original Order

150

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?

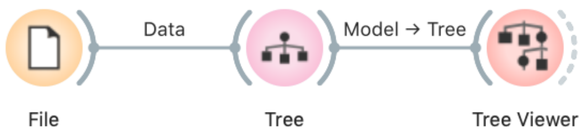
So many questions to answer!

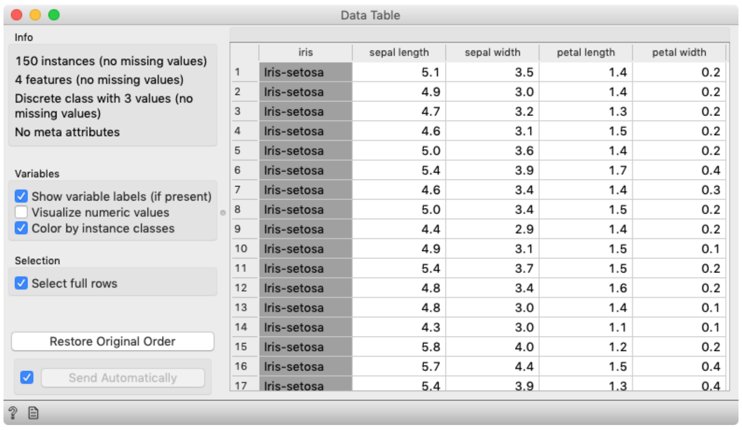
Classification Trees

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.*




Data Table

Info

150 instances (no missing values)
4 features (no missing values)
Discrete class with 3 values (no missing values)
No meta attributes

Variables

- [✓] Show variable labels (if present)
- [ ] Visualize numeric values
- [✓] Color by instance classes

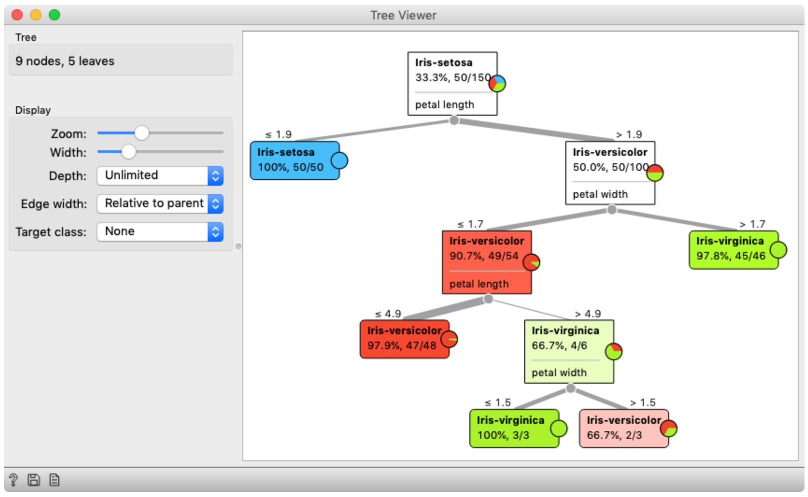
Selection

- [✓] Select full rows

Restore Original Order

[✓] Send Automatically

	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


Tree Viewer

Tree

9 nodes, 5 leaves

Display

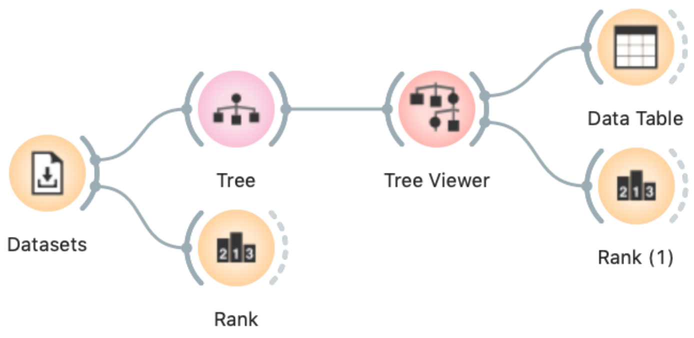
Zoom:
Width:
Depth: Unlimited
Edge width: Relative to parent
Target class: None

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).

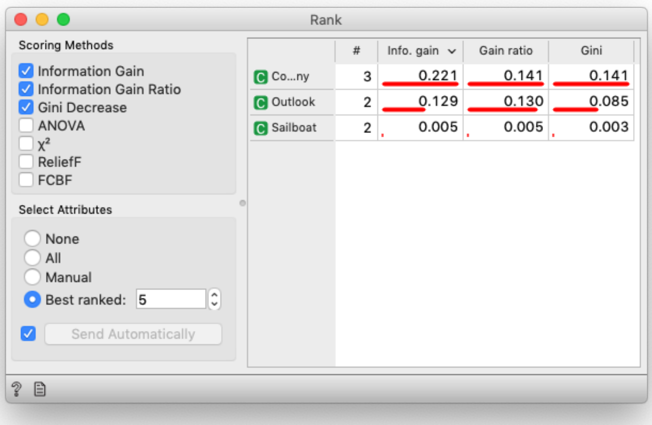
We still have not been very explicit about what we mean by "the most useful" feature. There are many ways to measure the quality of features, based on how well they distinguish between classes. We will illustrate the general idea with information gain. We can compute this measure in Orange using the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



Scoring Methods
- ☑ Information Gain
- ☑ Information Gain Ratio
- ☑ Gini Decrease
- ☐ ANOVA
- ☐ χ^2
- ☐ ReliefF
- ☐ FCBF

Select Attributes
- ○ None
- ○ All
- ○ Manual
- ● Best ranked: 5

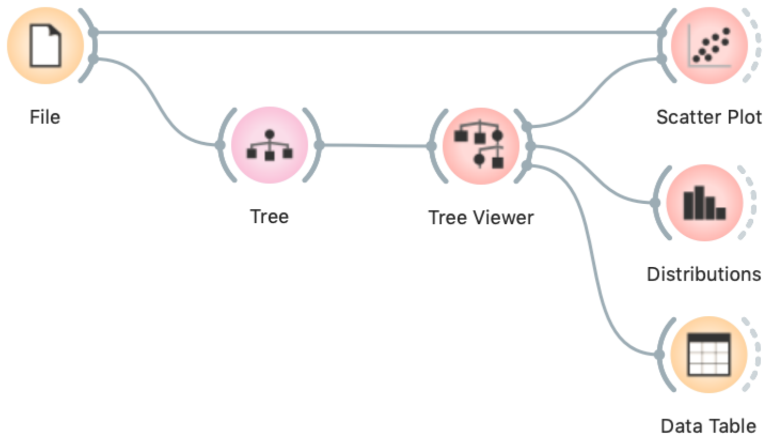
☑ Send Automatically

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

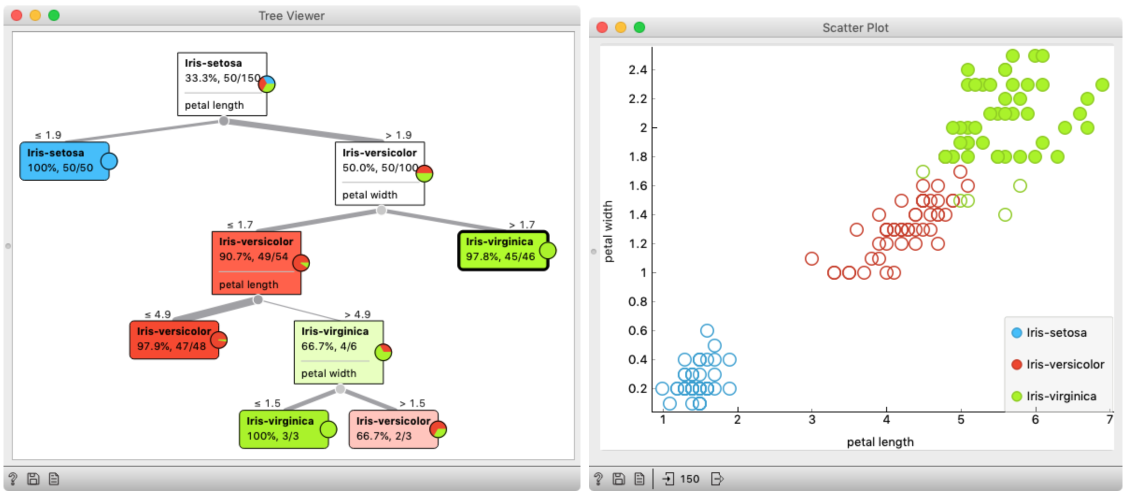
For the whole Sailing data set, Company is the most class-informative feature according to all measures shown.

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

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



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

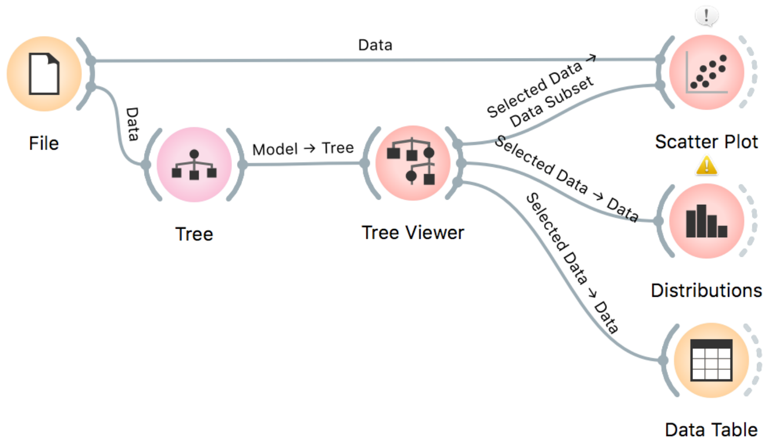


In the Tree Viewer we selected the rightmost node. All data instances coming to the selected node are highlighted in Scatter Plot.

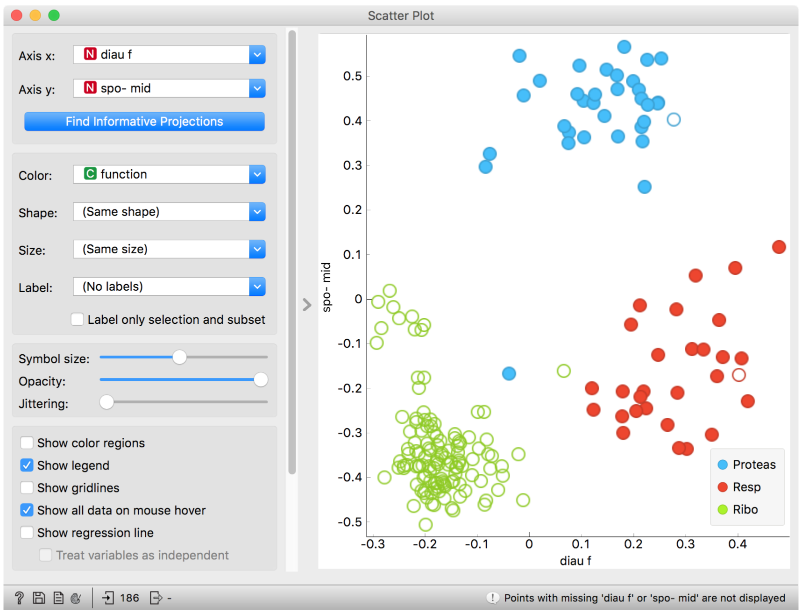
Wherever possible, visualizations in Orange are designed to support selection and passing of the data that applies to it. Finding interesting data subsets and analyzing their commonalities is a central part of explorative data analysis, a data analysis approach favored by the data visualization guru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

Model Inspection

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



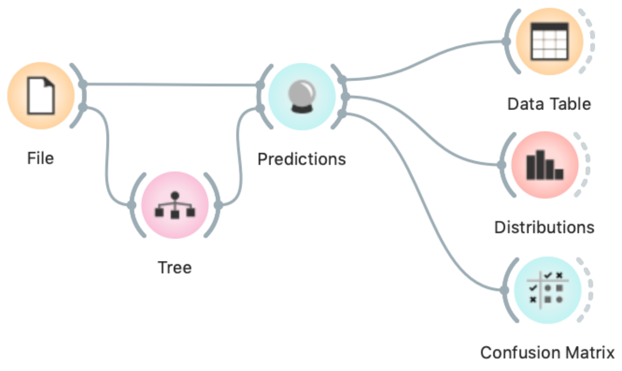
Just for fun, we have included a few other widgets in this workflow. The Tree Viewer selects data instances by inferring rules from the data itself and optimizing to obtain purer data subsets.



Classification Accuracy

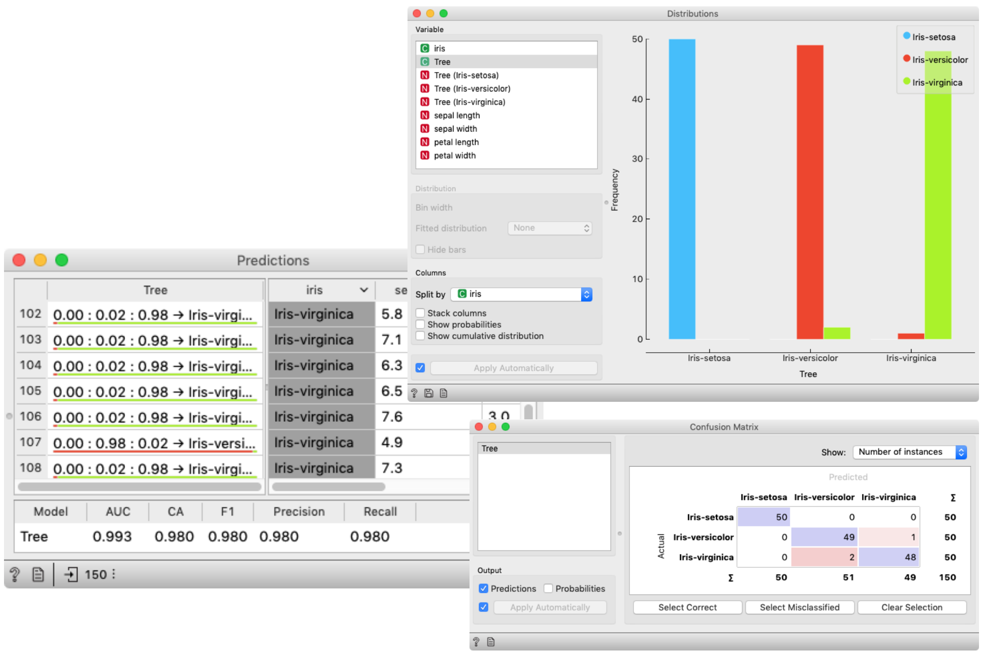
$$\text{accuracy} = \frac{\#\{\text{correct}\}}{\#\{\text{all}\}}$$

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.



	Tree	iris	se
102	0.00 : 0.02 : 0.98 → Iris-virgi...	Iris-virginica	5.8
103	0.00 : 0.02 : 0.98 → Iris-virgi...	Iris-virginica	7.1
104	0.00 : 0.02 : 0.98 → Iris-virgi...	Iris-virginica	6.3
105	0.00 : 0.02 : 0.98 → Iris-virgi...	Iris-virginica	6.5
106	0.00 : 0.02 : 0.98 → Iris-virgi...	Iris-virginica	7.6
107	0.00 : 0.98 : 0.02 → Iris-versi...	Iris-virginica	4.9
108	0.00 : 0.02 : 0.98 → Iris-virgi...	Iris-virginica	7.3

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

Predicted

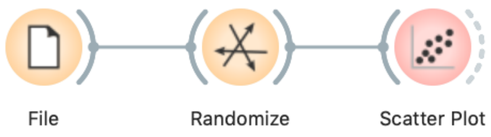
Actual	Iris-setosa	Iris-versicolor	Iris-virginica	Σ
Iris-setosa	50	0	0	50
Iris-versicolor	0	49	1	50
Iris-virginica	0	2	48	50
Σ	50	51	49	150

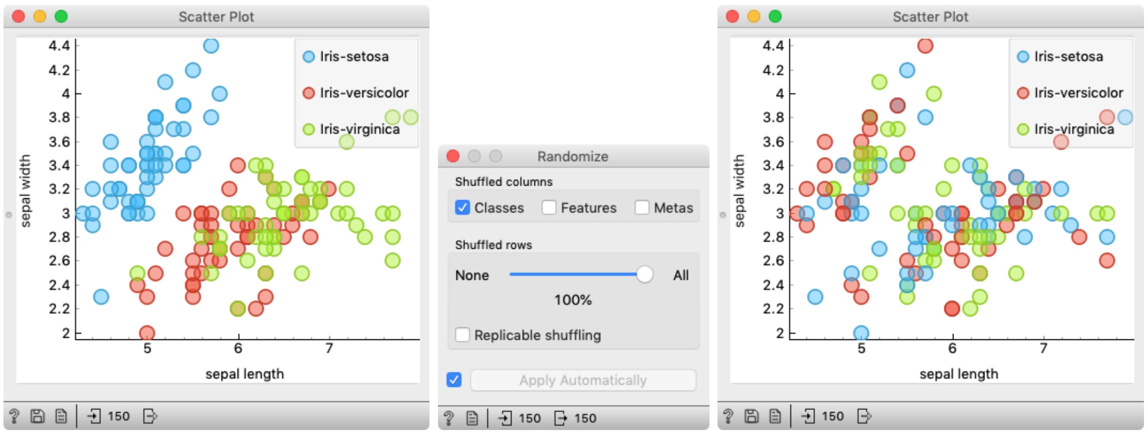
The Confusion Matrix shows 3 incorrectly classified examples, which makes the accuracy $(150 - 3)/150 = 98\%$**.**

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!





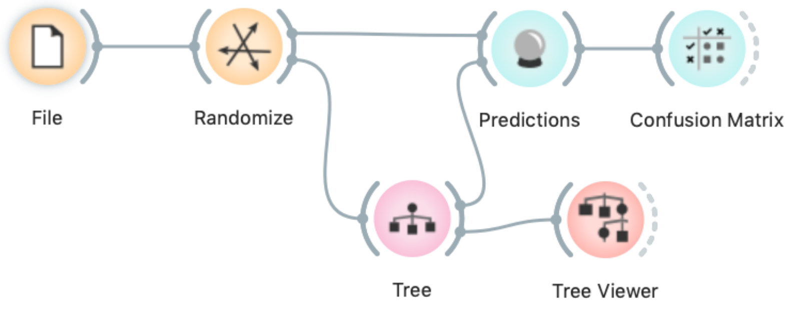
Shuffled columns
☑ Classes ☐ Features ☐ Metas

Shuffled rows
None — All
100%

☐ Replicable shuffling

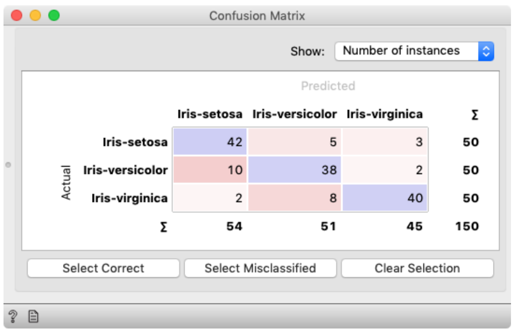
Left: scatter plot of the Iris data set before randomization; right: scatter plot after shuffling 100% of rows.

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*.

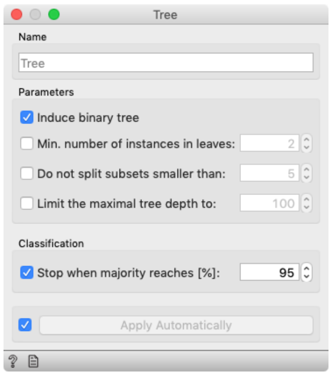
Most unusual. Despite shuffling all the classes, which destroyed any connection between features and the class variable, about 80% of predictions were still correct.



Actual \ Predicted	Iris-setosa	Iris-versicolor	Iris-virginica	Σ
Iris-setosa	42	5	3	50
Iris-versicolor	10	38	2	50
Iris-virginica	2	8	40	50
Σ	54	51	45	150

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.



Tree

Name: Tree

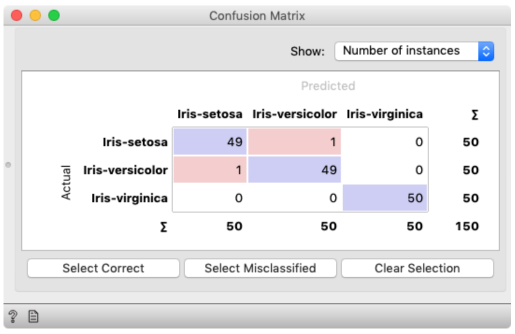
Parameters

- [✓] Induce binary tree
- [ ] Min. number of instances in leaves: 2
- [ ] Do not split subsets smaller than: 5
- [ ] Limit the maximal tree depth to: 100

Classification

- [✓] Stop when majority reaches [%]: 95

[✓] Apply Automatically

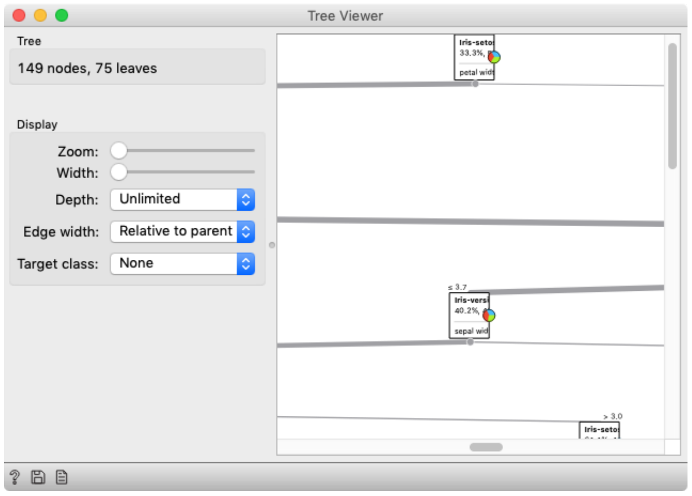


Confusion Matrix

Show: Number of instances

Actual \ Predicted	Iris-setosa	Iris-versicolor	Iris-virginica	Σ
Iris-setosa	49	1	0	50
Iris-versicolor	1	49	0	50
Iris-virginica	0	0	50	50
Σ	50	50	50	150

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



Tree Viewer

Tree: 149 nodes, 75 leaves

Display

Zoom:
Width:
Depth: Unlimited
Edge width: Relative to parent
Target class: None

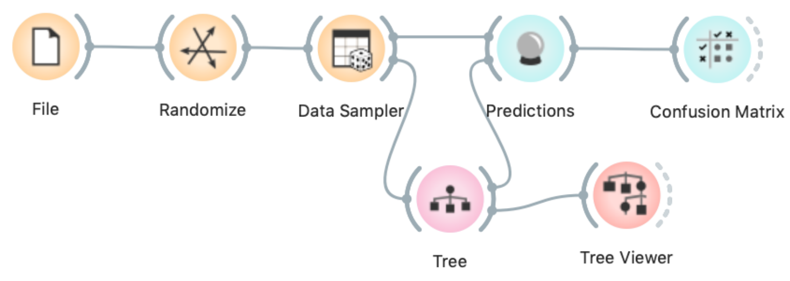
In the build tree, there are 75 leaves. Remember, there are only 150 rows in the Iris data set.

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.

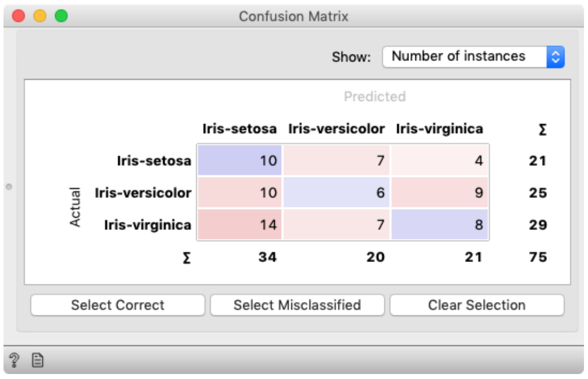
Ha, if this is so, if a classifier remembers everything from a data set but without discovering any general patterns, it should perform miserably on any new data set. Let us check this out. We will split our data set into two sets, training and testing, train the classification tree on the training data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

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.



Show: Number of instances

Predicted

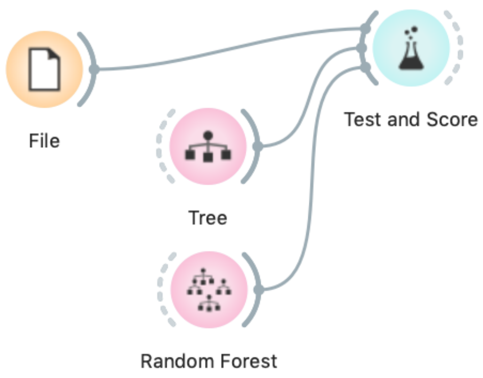
Actual	Iris-setosa	Iris-versicolor	Iris-virginica	Σ
Iris-setosa	10	7	4	21
Iris-versicolor	10	6	9	25
Iris-virginica	14	7	8	29
Σ	34	20	21	75

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

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

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

Cross-Validation



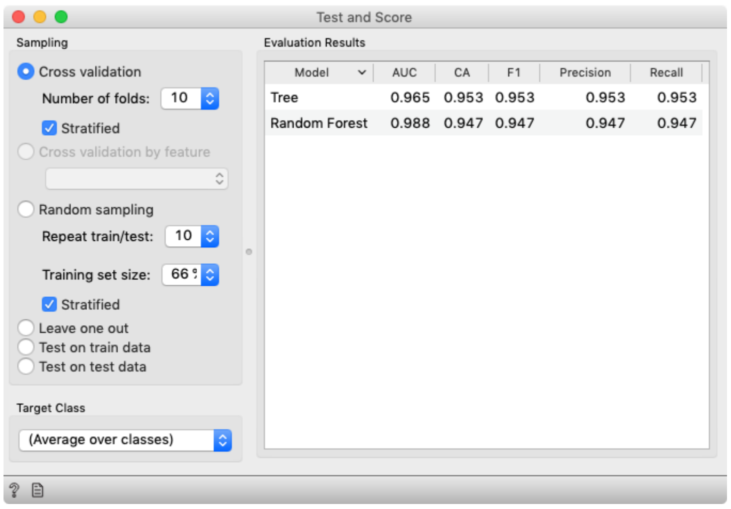
Estimating the accuracy may depend on a particular split of the data set. To increase robustness, we can repeat the measurement several times, each time choosing a different subset of the data for training. One such method is cross-validation. It is available in Orange in the *Test and Score* widget.

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

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

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

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



Sampling: Cross validation (selected)
Number of folds: 10
✓ Stratified
Repeat train/test: 10
Training set size: 66 %
✓ Stratified
Target Class: (Average over classes)

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

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

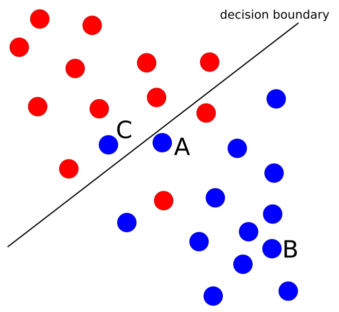
Logistic Regression

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.

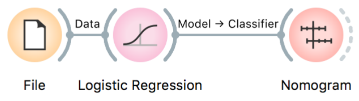
Second, it tries to find an optimal plane that separates instances with the target value from the rest. Then it uses logistic function to transform the distance to the plane into probabilities. The further away from the plane an instance will be, the higher the probability it belongs to the class on 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).

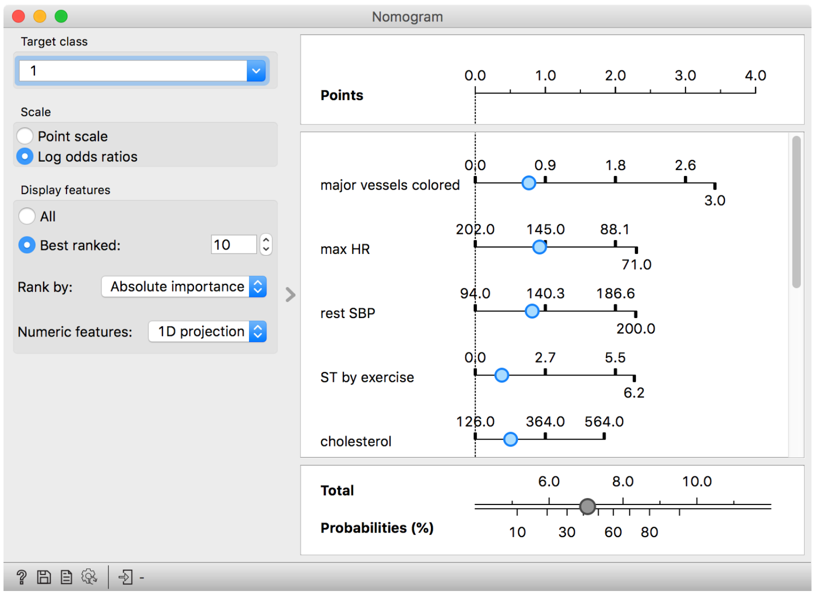
Logistic regression tries to find such a plane that all points from one class are as far away from the boundary (in the correct direction) as possible.

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?



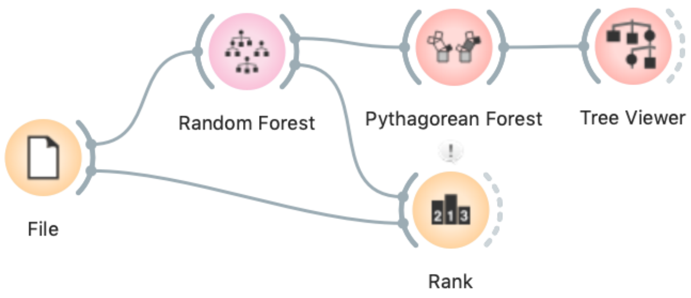


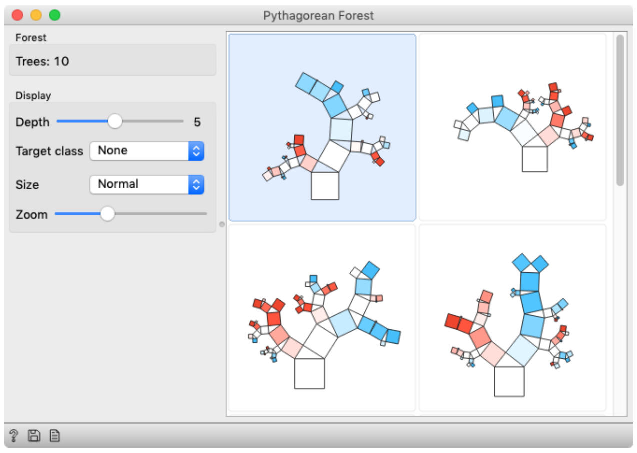
Another characteristic of logistic regression is that it observes all variables at once and takes the correlation into account. If some variables are correlated, their importance will be spread among them.

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?

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.





Forest

Trees: 10

Display

Depth 5

Target class None

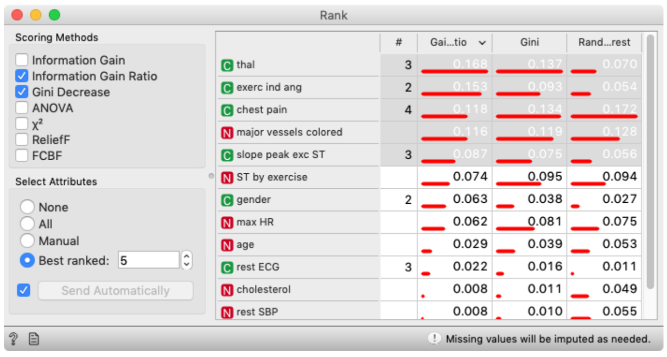
Size Normal

Zoom

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.



Scoring Methods

- [ ] Information Gain
- [✓] Information Gain Ratio
- [✓] Gini Decrease
- [ ] ANOVA
- [ ] χ^2
- [ ] ReliefF
- [ ] FCBF

Select Attributes

None / All / Manual / [✓ Best ranked: 5]

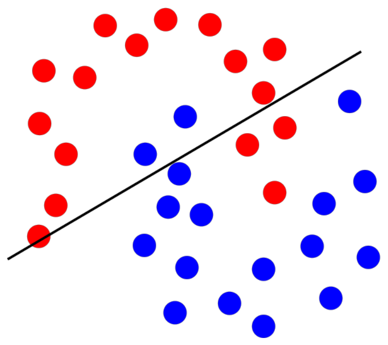
[✓] Send Automatically

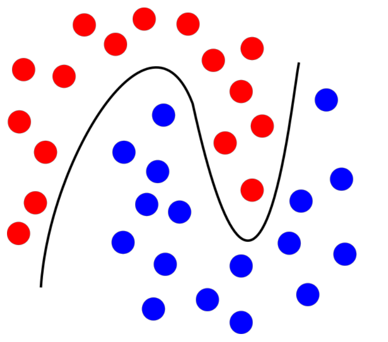
	#	Gai...tio	Gini	Rand...rest
C thal	3	0.168	0.137	0.070
C exerc ind ang	2	0.153	0.093	0.054
C chest pain	4	0.118	0.134	0.172
N major vessels colored		0.116	0.119	0.128
C slope peak exc ST	3	0.087	0.075	0.056
N ST by exercise		0.074	0.095	0.094
C gender	2	0.063	0.038	0.027
N max HR		0.062	0.081	0.075
N age		0.029	0.039	0.053
C rest ECG	3	0.022	0.016	0.011
N cholesterol		0.008	0.011	0.049
N rest SBP		0.008	0.010	0.055

Missing values will be imputed as needed.

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

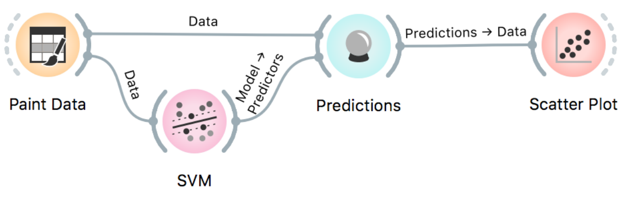
Support Vector Machines


Decision boundary of a linear regression classifier.

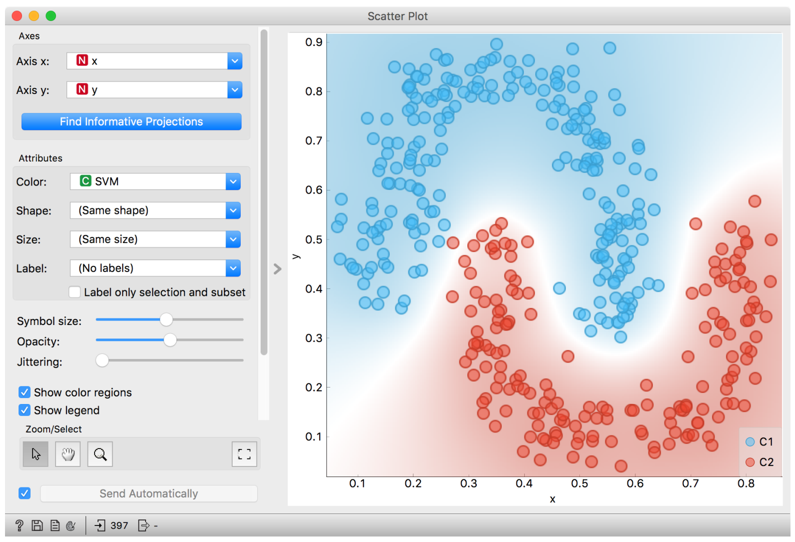

Decision boundary of a support vector machine classifier with an RBF kernel.

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

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



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


Scatter Plot

Axes
Axis x: x
Axis y: y
Find Informative Projections

Attributes
Color: SVM
Shape: (Same shape)
Size: (Same size)
Label: (No labels)
☐ Label only selection and subset

Symbol size:
Opacity:
Jittering:

☑ Show color regions
☑ Show legend

Zoom/Select

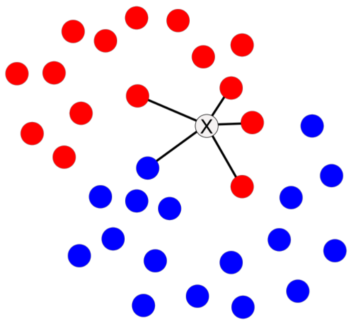
☑ Send Automatically

397 -

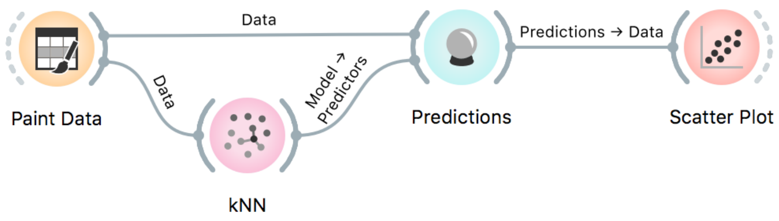
k-Nearest Neighbors

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

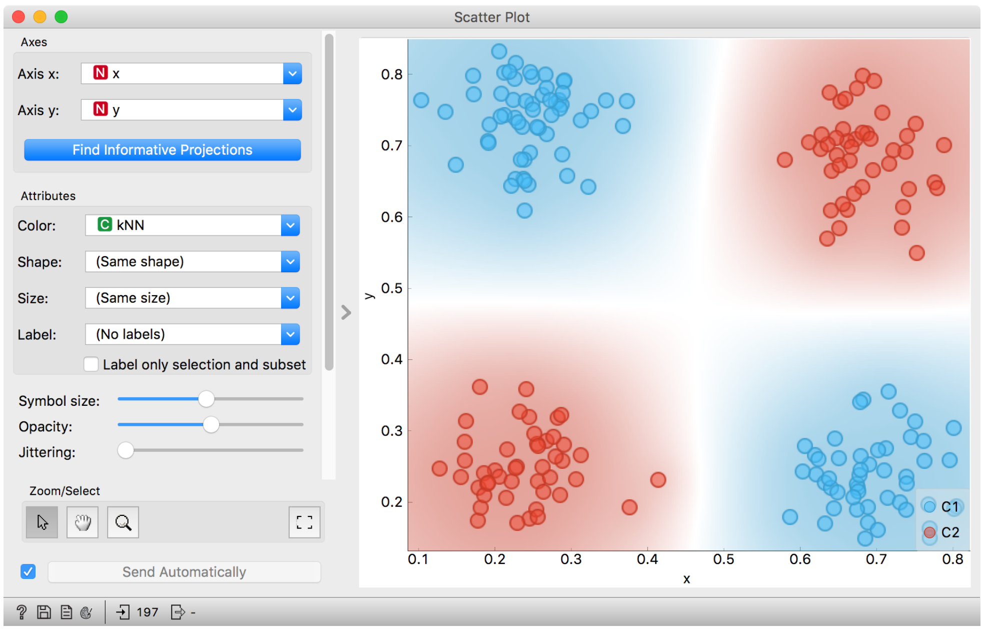
Unlike most other algorithms, kNN does not construct a model but just stores the data. This kind of learning is called *lazy learning*.



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



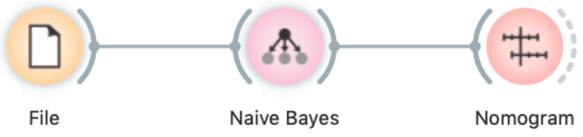
The advantage of kNN algorithm is that it can successfully model the data, where classes are not linearly separably. It can also be retrained quickly, because new data instances effect model only locally. However, the first training is can be slow for large data sets, as the model has to estimate k distances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



Naive Bayes

Naive Bayes 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.

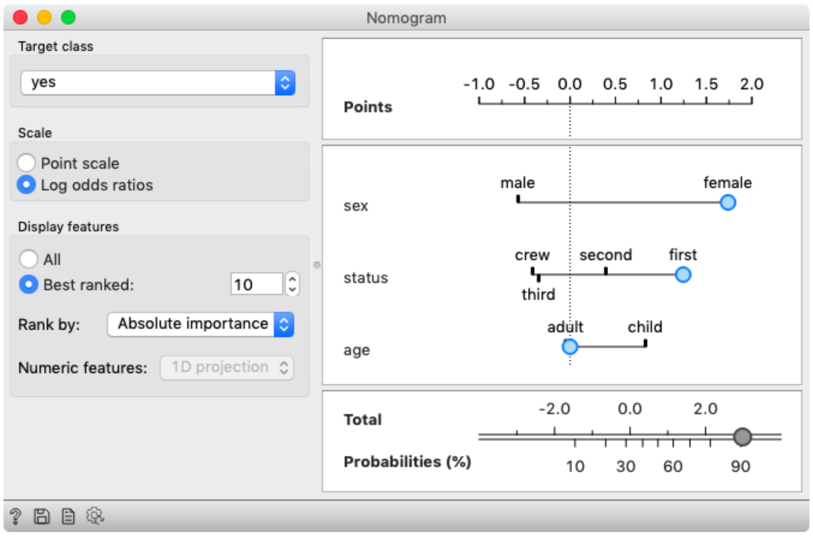
Naive Bayes is also a classification method. To see how naive Bayes works, we will use a data set on passengers' survival in the Titanic disaster of 1912. The *Titanic* data set describes 2201 passengers, with their tickets (first, second, thirds class or crew), age and gender.



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

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

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.


Target class: yes

Scale: Point scale [✓ Log odds ratios]

Display features: All [✓ Best ranked: 10]

Rank by: Absolute importance

Numeric features: 1D projection