

1 2 3 4 5

T E S T 1 2 3 4 5

BIOLAB

Copyright © 2022 12345

PUBLISHED BY BIOLAB

TUFTE-LATEX.GOOGLECODE.COM

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at <http://www.apache.org/licenses/LICENSE-2.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

First printing, April 2022

Contents

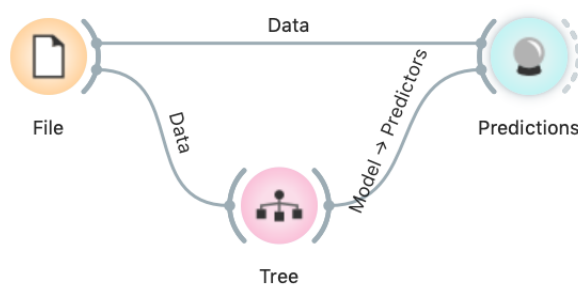
<i>Classification</i>	5
<i>Classification Trees</i>	6
<i>Model Inspection</i>	9
<i>Naive Bayes</i>	10
<i>Classification Accuracy</i>	11
<i>Assignment: Decision Boundaries</i>	12
<i>How to Cheat</i>	13
<i>Random Forests</i>	16
<i>Cross-Validation</i>	17
<i>Assignment: Overfitting</i>	18
<i>Linear Regression</i>	19
<i>Regularization</i>	22
<i>Network from Text</i>	24
<i>Bibliography</i>	27

<i>Index</i>	28
--------------	----

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!

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

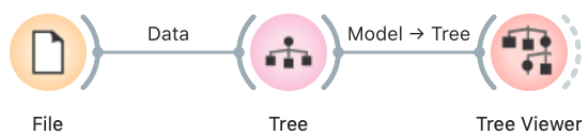
iris	sepal length	sepal width
Iris-setosa	5.1	3.8
Iris-setosa	4.6	3.2
Iris-setosa	5.3	3.7
Iris-setosa	5.0	3.3
Iris-versicolor	7.0	3.2
Iris-versicolor	6.4	3.2
Iris-versicolor	6.9	3.1
Iris-versicolor	5.5	2.3
Iris-versicolor	6.5	2.8

Classification Trees

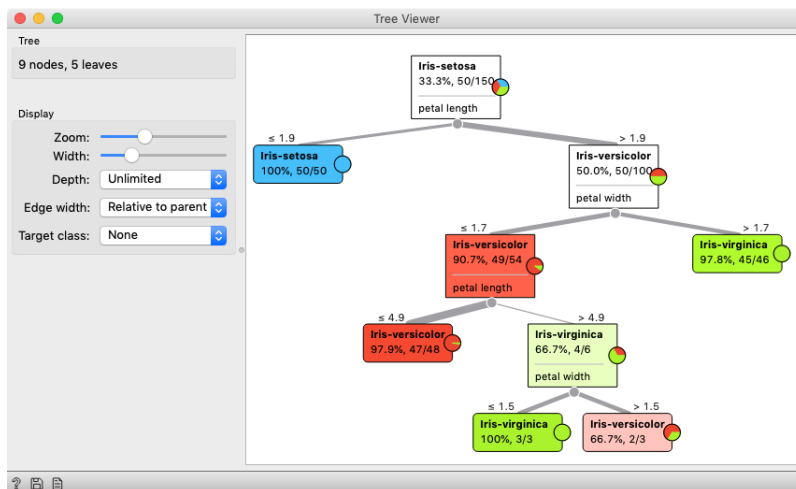
Classification trees were hugely popular in the early years of machine learning, when they were first independently proposed by the engineer Ross Quinlan (C4.5) and a group of statisticians (CART), including the father of random forests Leo Brieman.

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					
<input checked="" type="checkbox"/> Show variable labels (if present)					
<input type="checkbox"/> Visualize numeric values					
<input checked="" type="checkbox"/> Color by instance classes					
Selection					
<input checked="" type="checkbox"/> Select full rows					
Restore Original Order					
<input checked="" type="checkbox"/> Send Automatically					
iris					
1	Iris-setosa	5.1	3.5	1.4	0.2
2	Iris-setosa	4.9	3.0	1.4	0.2
3	Iris-setosa	4.7	3.2	1.3	0.2
4	Iris-setosa	4.6	3.1	1.5	0.2
5	Iris-setosa	5.0	3.6	1.4	0.2
6	Iris-setosa	5.4	3.9	1.7	0.4
7	Iris-setosa	4.6	3.4	1.4	0.3
8	Iris-setosa	5.0	3.4	1.5	0.2
9	Iris-setosa	4.4	2.9	1.4	0.2
10	Iris-setosa	4.9	3.1	1.5	0.1
11	Iris-setosa	5.4	3.7	1.5	0.2
12	Iris-setosa	4.8	3.4	1.6	0.2
13	Iris-setosa	4.8	3.0	1.4	0.1
14	Iris-setosa	4.3	3.0	1.1	0.1
15	Iris-setosa	5.8	4.0	1.2	0.2
16	Iris-setosa	5.7	4.4	1.5	0.4
17	Iris-setosa	5.4	3.9	1.3	0.4



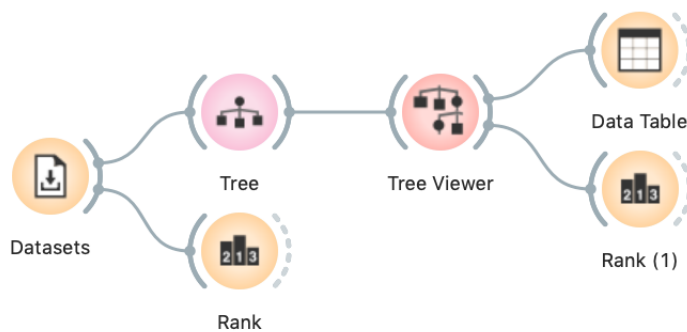
We read the tree from top to bottom. Looks like the column *petal length* best separates the iris variety *setosa* from the others, and in the next step, *petal width* then almost perfectly separates the remaining two varieties.

Trees place the most useful feature at the root. What would be the most useful feature? The feature that splits the data into two purest possible subsets. It then splits both subsets further, again by their most useful features, and keeps doing so until

it reaches subsets in which all data belongs to the same class (leaf nodes in strong blue or red) or until it runs out of data instances to

split or out of useful features (the two leaf nodes in white).

We still have not been very explicit about what we mean by "the most useful" feature. There are many ways to measure the quality of features, based on how well they distinguish between classes. We will illustrate the general idea with information gain. We can compute this measure in Orange using the *Rank* widget, which estimates the quality of data features and ranks them according to how informative they are about the class. We can either estimate the information gain from the whole data set, or compute it on data corresponding to an internal node of the classification tree in the *Tree Viewer*. In the following example we use the *Sailing* data set.



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

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

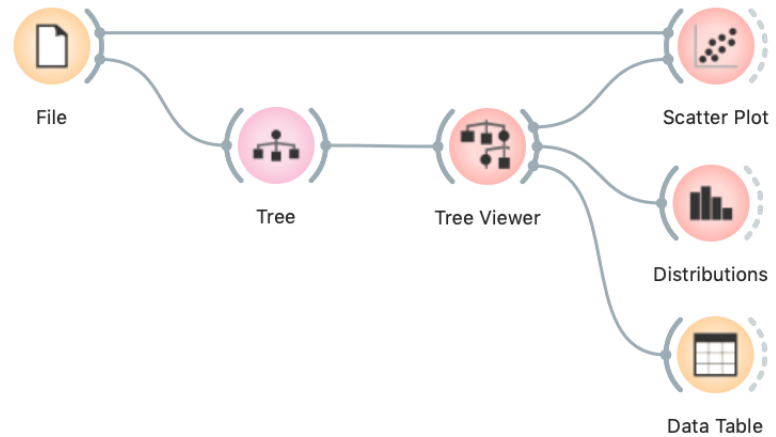
Besides the information gain, *Rank* displays several other measures (including Gain Ratio and Gini), which are often quite in agreement and were invented to better handle discrete features with many different values.

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

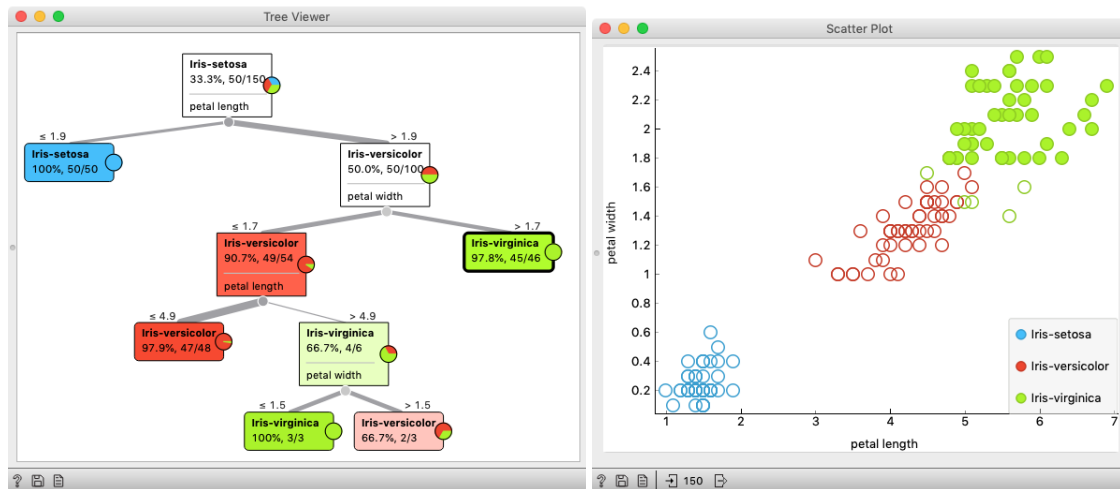
For the whole *Sailing* data set, *Company* is the most class-informative feature according to all measures shown.

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.

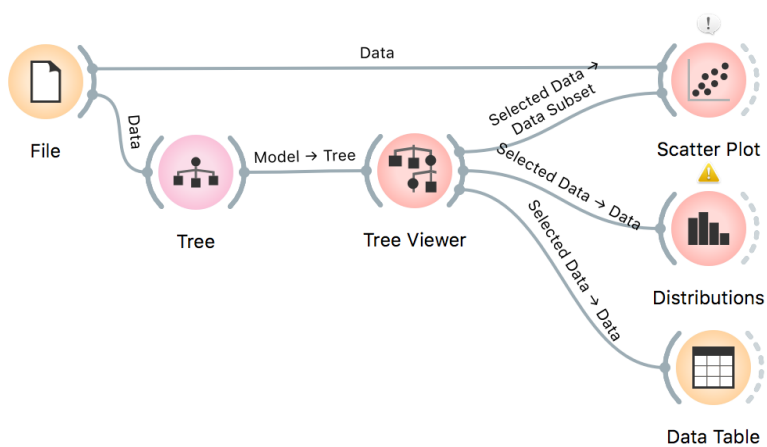


In the *Tree Viewer* we selected the rightmost node. All data instances coming to the selected node are highlighted in *Scatter Plot*.

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.



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.

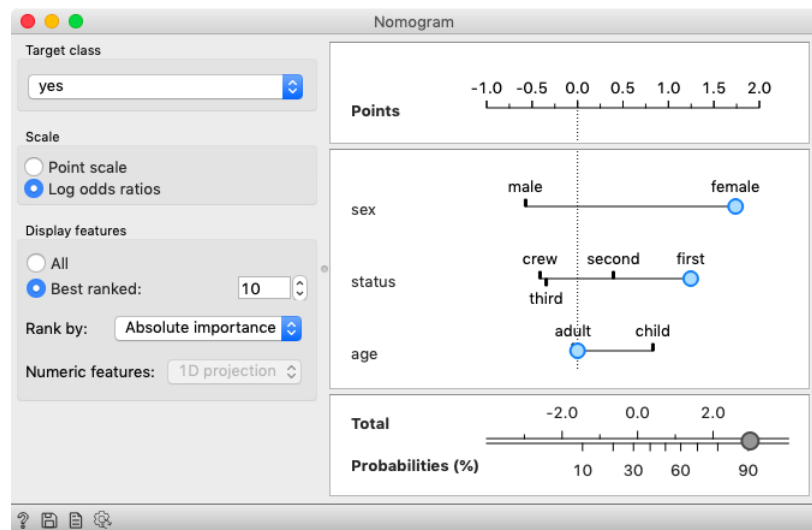
Naive Bayes is also a classification method. To see how naive Bayes works, we will use a data set on passengers' survival in the Titanic disaster of 1912. The *Titanic* data set describes 2201 passengers, with their tickets (first, second, thirds class or crew), age and gender.



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.



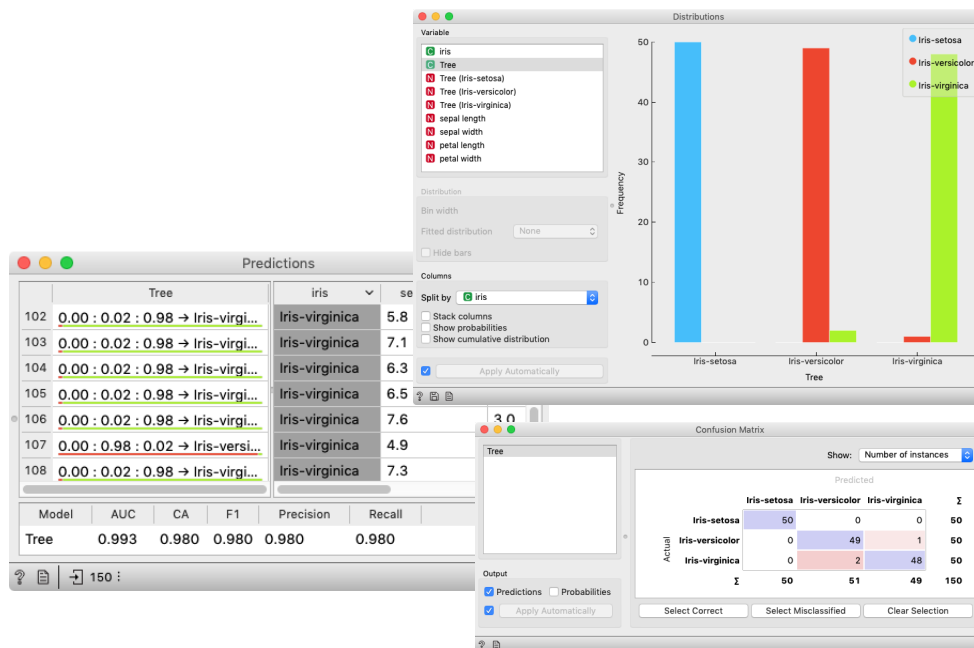
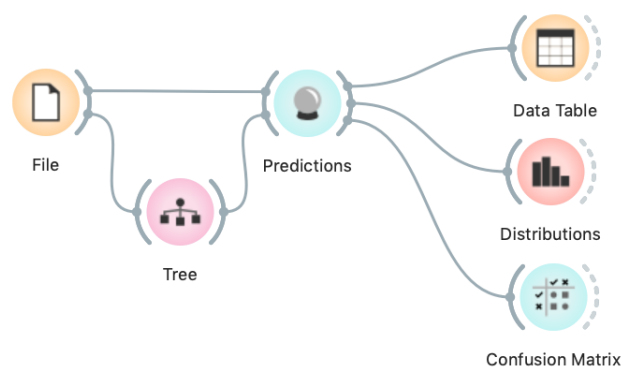
Classification Accuracy

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.

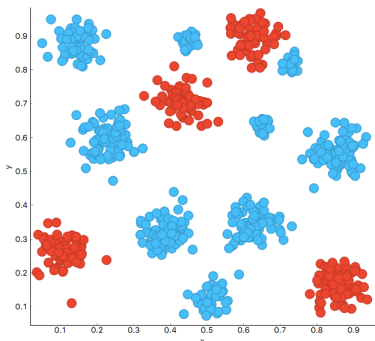
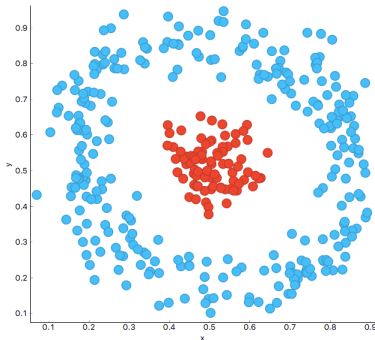
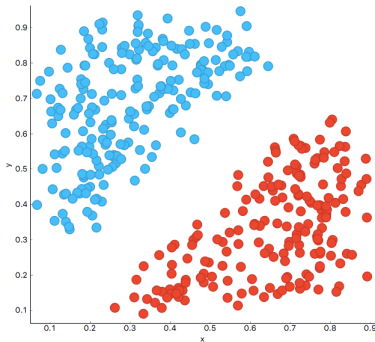
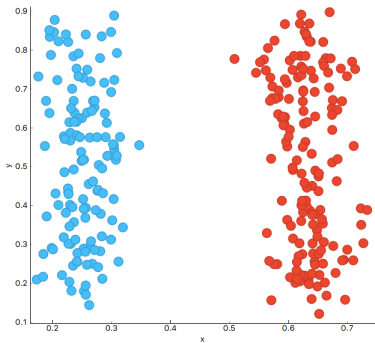
For precise statistics of correctly and incorrectly classified examples open the *Confusion Matrix* widget.



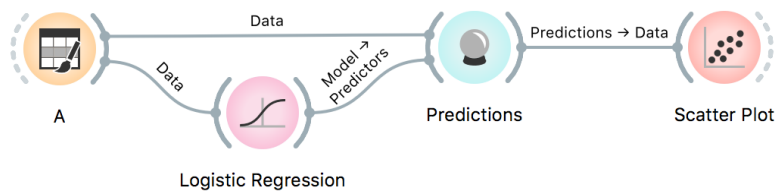
The *Confusion Matrix* shows 3 incorrectly classified examples, which makes the accuracy $(150 - 3)/150 = 98\%$.

Assignment: Decision Boundaries

You can try painting the data yourself or download it from [here](#).



CLASSIFIERS COME IN ALL SHAPES AND SIZES. What we mean by that is that each has its own way of learning from the data, its own strengths and weaknesses. Knowing how classifiers work is crucial for selecting the right algorithm for your task.



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

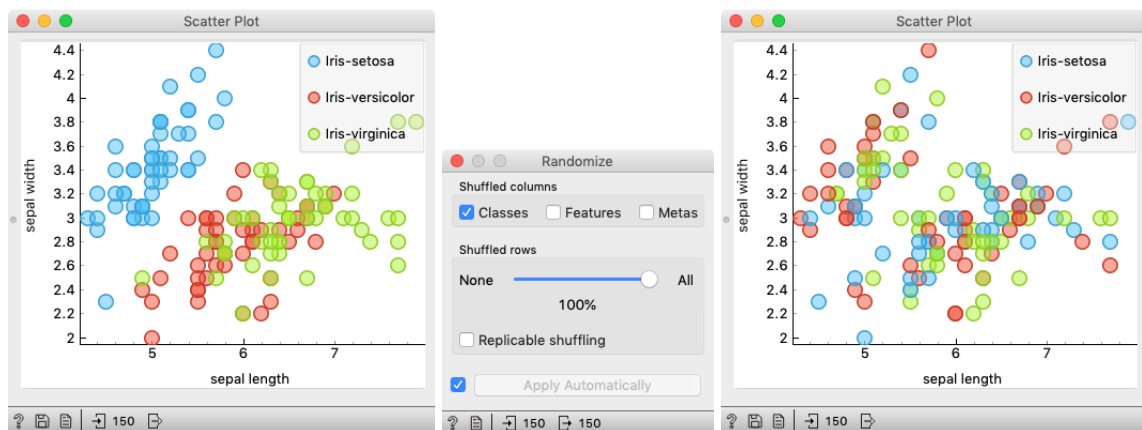
1. Which classifiers work well with data set A? Which with B, C, and D?
2. Which classifier is struggling the most? Which one the least? Why?
3. Look at the Tree with *Tree Viewer* for data set C. What do you notice?
4. In the above example, you can separate classes with a single stroke of a pen. Now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

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.

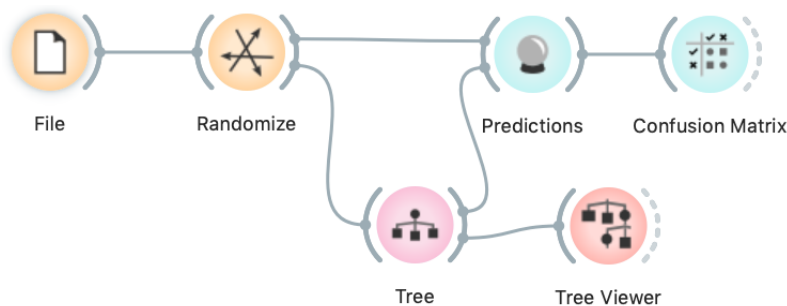
At this stage, the classification tree looks very good. There's only one data point where it makes a mistake. Can we mess up the data set so bad that the trees will ultimately fail? Like, remove any existing correlation between features and the class?

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.

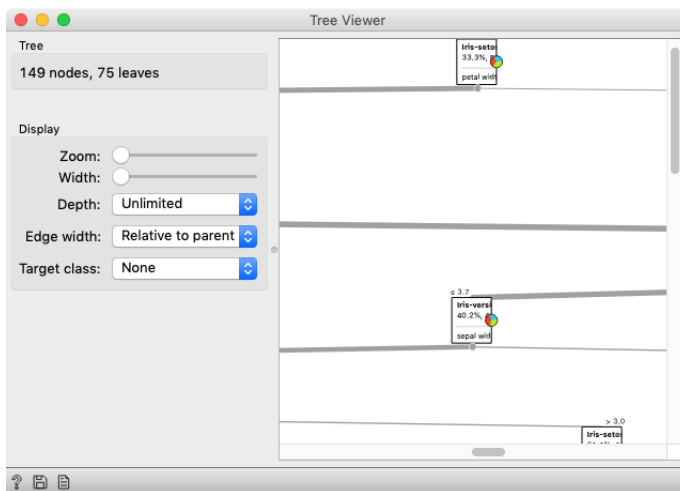
		Predicted			
		Iris-setosa	Iris-versicolor	Iris-virginica	Σ
Actual	Iris-setosa	42	5	3	50
	Iris-versicolor	10	38	2	50
	Iris-virginica	2	8	40	50
Σ		54	51	45	150

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.



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



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

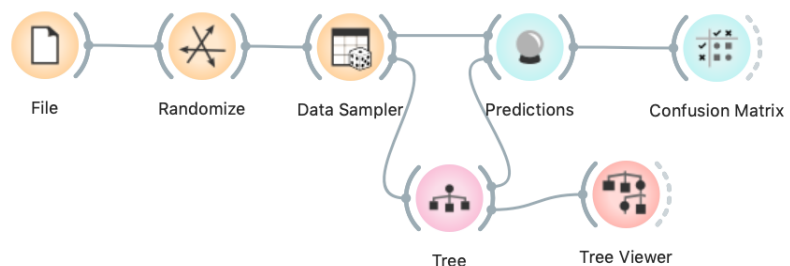
To find the answer to this riddle, open the *Tree Viewer* and check out the tree. How many nodes does it have? Are there many data instances in the leaf nodes?

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.

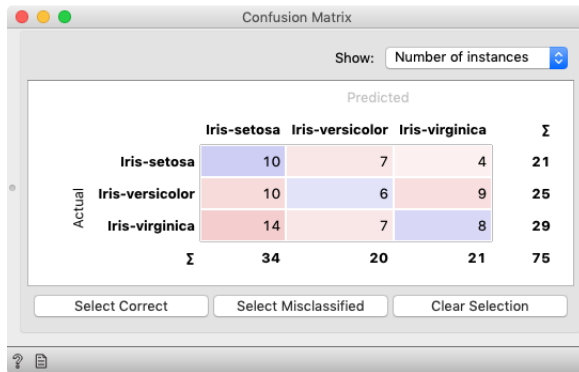
Connect the *Data Sampler* widget carefully. The *Data Sampler* splits the data to a sample and out-of-sample (so called remaining data). The sample was given to the *Tree* widget, while the remaining data was handed to the *Predictions* widget. Set the *Data Sampler* so that the size of these two data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



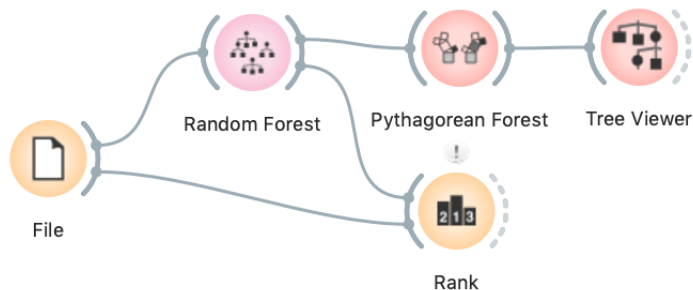
		Predicted			Σ
		Iris-setosa	Iris-versicolor	Iris-virginica	
Actual	Iris-setosa	10	7	4	21
	Iris-versicolor	10	6	9	25
	Iris-virginica	14	7	8	29
Σ		34	20	21	75

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

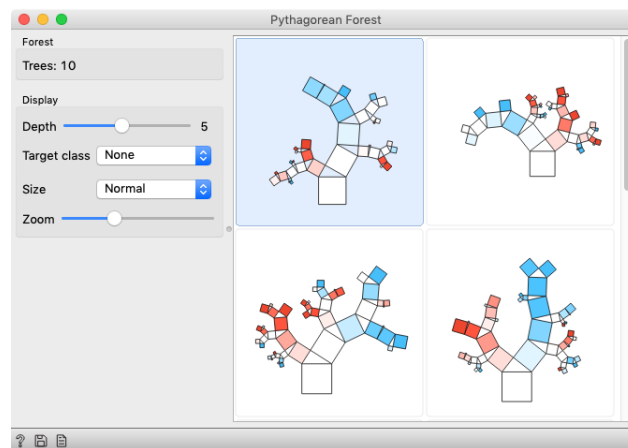
We have just learned that we need to train the classifiers on the training set and then test it on a separate test set to really measure performance of a classification technique. With this test, we can distinguish between those classifiers that just memorize the training data and those that actually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

Random Forests



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.

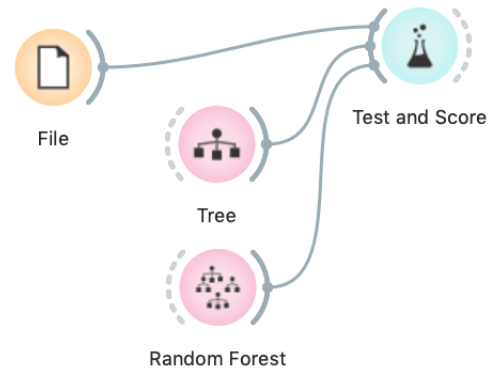
Scoring Methods	Select Attributes	#	Gai...tio	Gini	Rand...rest
<input type="checkbox"/> Information Gain	<input type="radio"/> None	3	0.168	0.137	0.070
<input checked="" type="checkbox"/> Information Gain Ratio	<input type="radio"/> All	2	0.163	0.093	0.054
<input checked="" type="checkbox"/> Gini Decrease	<input type="radio"/> Manual	4	0.118	0.146	0.115
<input type="checkbox"/> ANOVA	<input checked="" type="radio"/> Best ranked: 5		0.116	0.119	0.128
<input type="checkbox"/> X ²	<input type="checkbox"/> Send Automatically	3	0.087	0.075	0.056
<input type="checkbox"/> ReliefF			0.074	0.095	0.094
<input type="checkbox"/> FCBF		2	0.063	0.038	0.027
			0.062	0.081	0.075
			0.029	0.039	0.053
		3	0.022	0.016	0.011
			0.008	0.011	0.049
			0.008	0.010	0.055

Cross-Validation

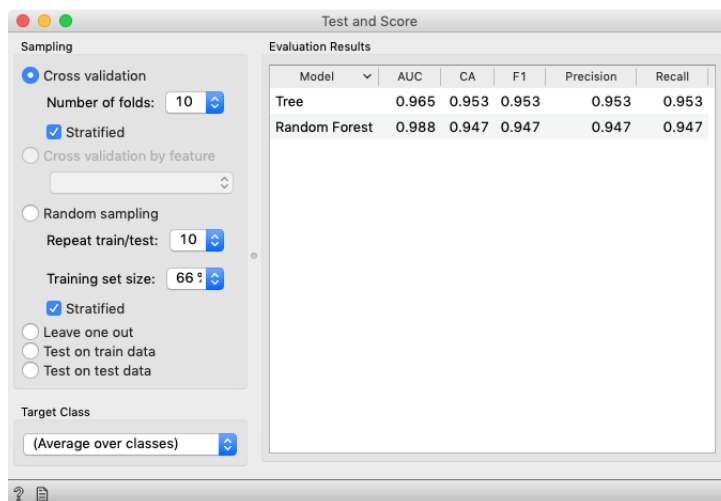
Estimating the accuracy may depend on a particular split of the data set. To increase robustness, we can repeat the measurement several times, each time choosing a different subset of the data for training. One such method is cross-validation. It is available in Orange in the *Test and Score* widget.

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.



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

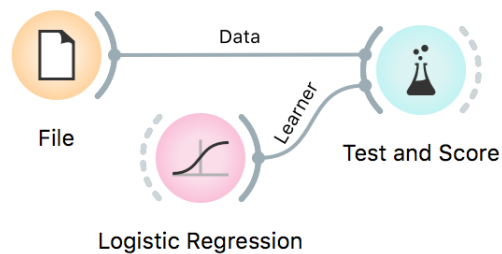
In the *Test and Score* widget, the second column, CA, stands for classification accuracy, and this is what we really care for for now.

Assignment: Overfitting

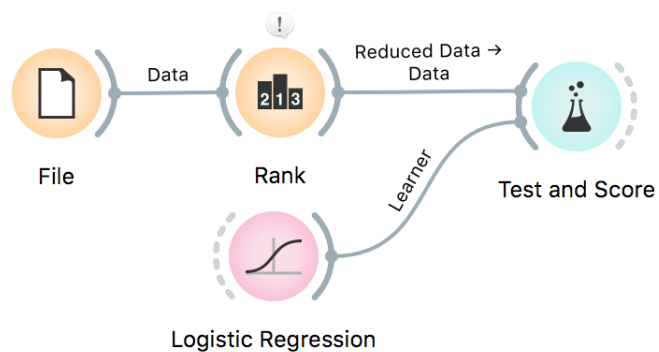
OVERFITTING IS SOMETHING WE TRY TO AVOID AT ALL TIMES. But overfitting comes in many shapes and sizes. For this exercise we will use a *blood-loneliness* data set with the File widget. This data set relates gene expressions in blood with a measure of loneliness obtained from a sample of elderly persons. Let's try to model loneliness with logistic regression and see how well the model performs.

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

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



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



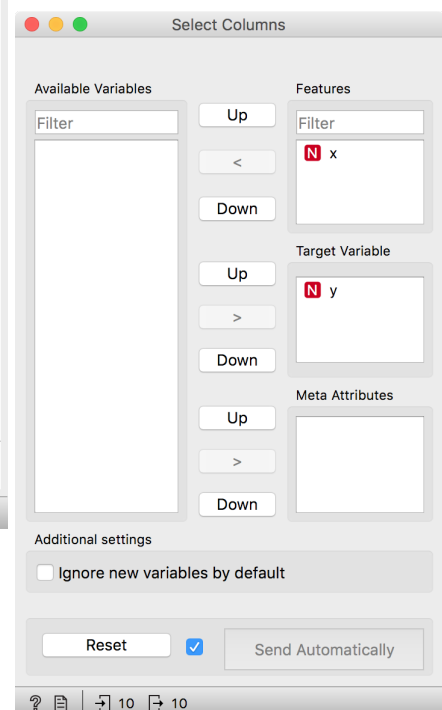
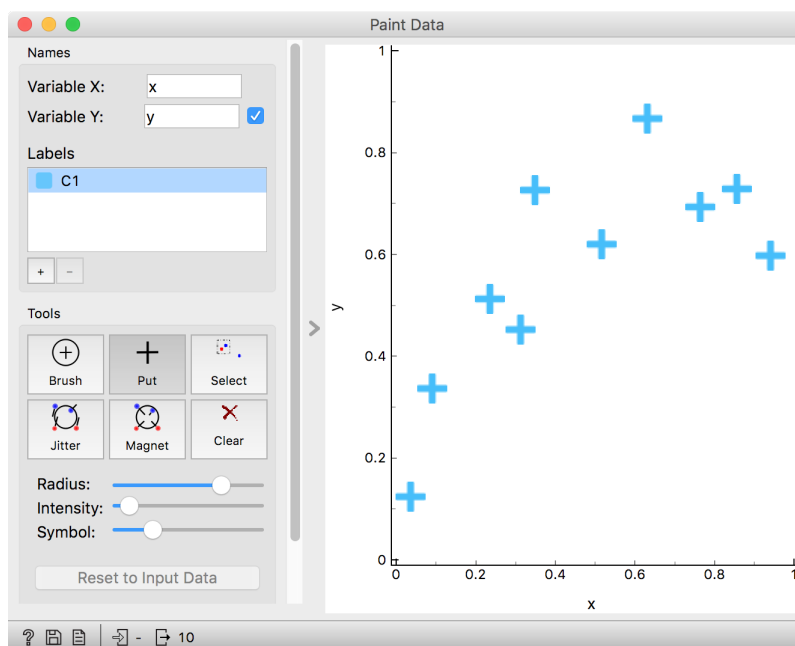
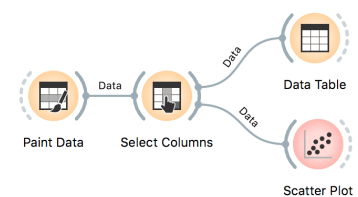
3. How are the results now? What happened? Is there a better way of performing feature selection?

Linear Regression

For a start, let us construct a very simple data set. It will contain just one continuous input feature (let's call it x) and a continuous class (let's call it y). We will use *Paint Data*, and then reassign one of the features to be a class using *Select Columns* and moving the feature y from "Features" to "Target Variable". It is always good to check the results, so we are including *Data Table* and *Scatter Plot* in the workflow at this stage. We will be modest this time and only paint 10 points and use Put instead of the Brush tool.

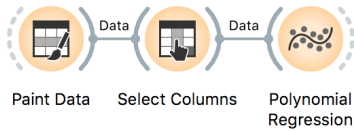
We want to build a model that predicts the value of the target variable y from the feature x . Say that we would like our model to be linear, to mathematically express it as $h(x) = \theta_0 + \theta_1 x$. Oh, this is the equation of a line. So we would like to draw a line through our data points. The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

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

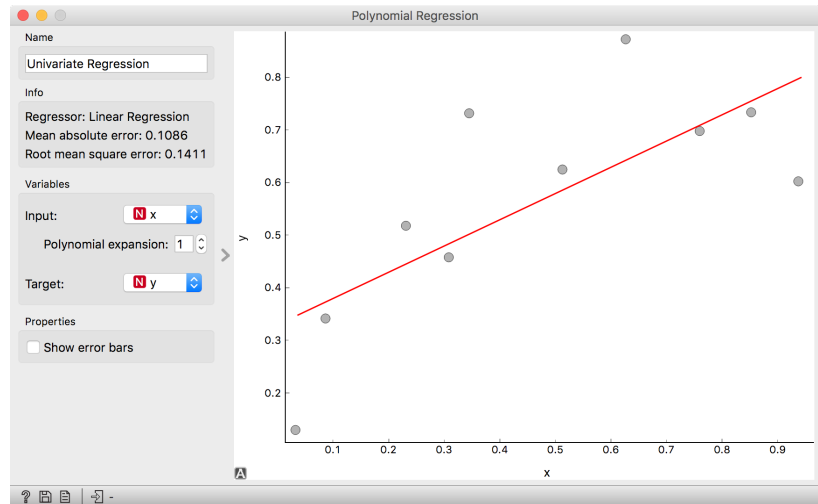


The question above requires us to define what a good fit is. Say, this could be the error the fitted model (the line) makes when it predicts the value of y for a given data point (value of x). The prediction is $h(x)$, so the error is $h(x) - y$. We should treat the negative and

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

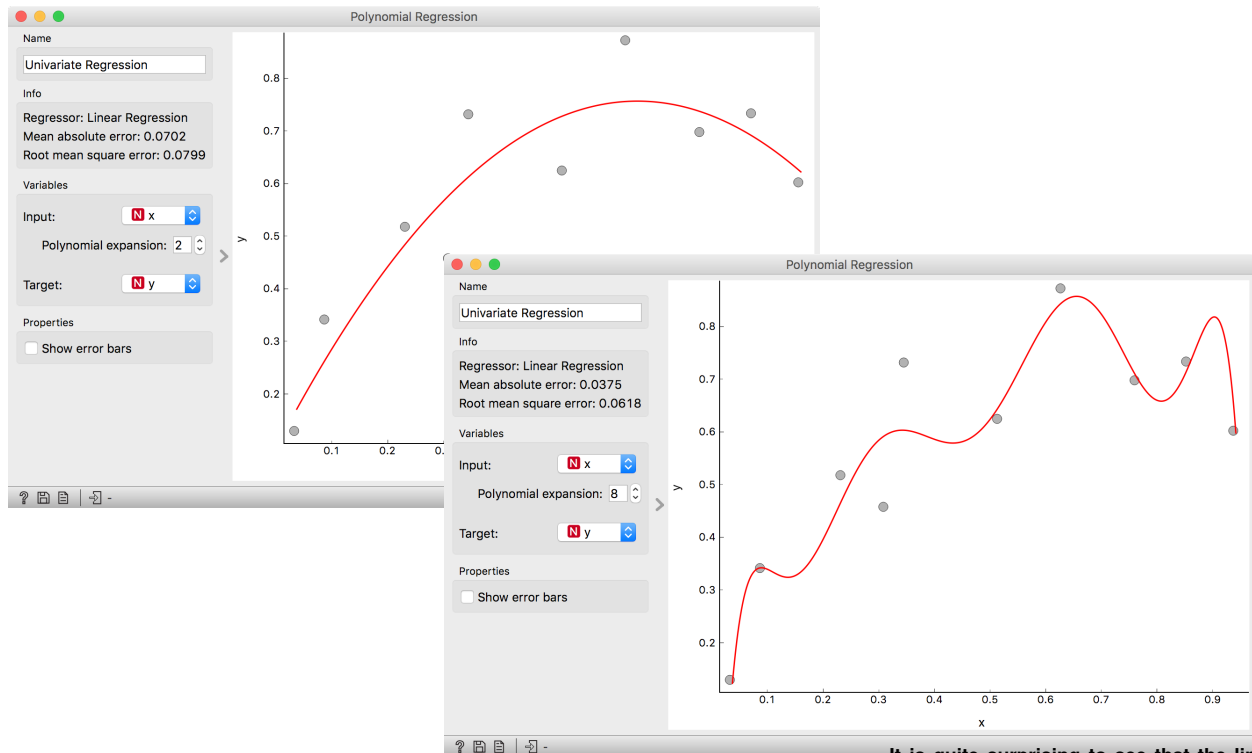


positive errors equally, plus – let us agree – we would prefer punishing larger errors more severely than smaller ones. Therefore, we should square the errors for each data point and sum them up. We got our objective function! It turns out that there is only one line that minimizes this function. The procedure that finds it is called linear regression. For cases where we have only one input feature, Orange has a special widget in the Educational add-on called *Polynomial Regression*.



Looks ok, except that these data points do not appear exactly on the line. We could say that the linear model is perhaps too simple for our data set. Here is a trick: besides the column x , the widget *Polynomial Regression* can add columns x^2 , x^3 , ..., x^n to our data set. The number n is a degree of polynomial expansion the widget performs. Try setting this number to higher values, say to 2, and then 3, and then, say, to 8. With the degree of 3, we are then fitting the data to a linear function $h(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$.

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

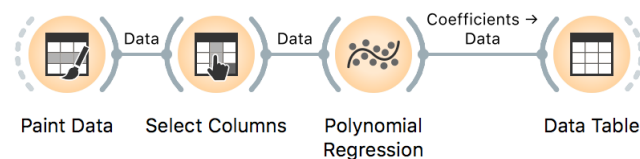


Overfitting is related to the complexity of the model. In polynomial regression, the parameters θ define the model. With the increased number of parameters, the model complexity increases. The simplest model has just one parameter (an intercept), ordinary linear regression has two (an intercept and a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

It is quite surprising to see that the linear regression model can fit non-linear (univariate) functions. It can fit the data with curves, such as those on the figures. How is this possible? Notice, though, that the model is a hyperplane (a flat surface) in the space of many features (columns) that are the powers of x . So for the degree 2, $h(x) = \theta_0 + \theta_1 x + \theta_2 x^2$ is a (flat) hyperplane. The visualization gets curvy only once we plot $h(x)$ as a function of x .

Regularization

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



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

	name	coef
1	1	0.106121
2	x	1.90152
3	x^2	-1.21305
4	x^3	-0.244903

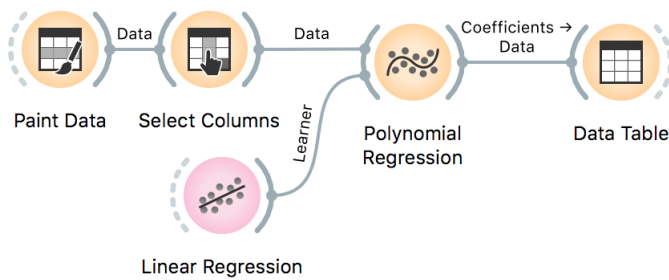
	name	coef
1	1	-0.787028
2	x	40.3077
3	x^2	-553.499
4	x^3	3756.01
5	x^4	-13830.3
6	x^5	29051.4
7	x^6	-34730.1
8	x^7	21961.7
9	x^8	-5696.56

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

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

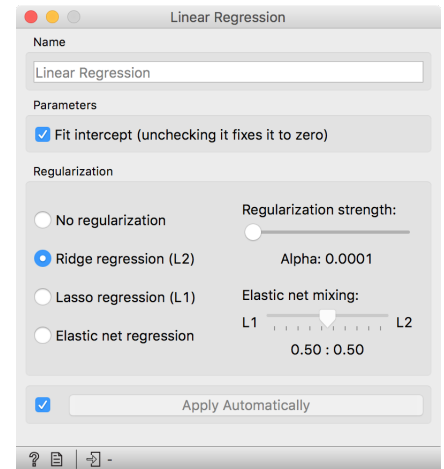
Here we go: we just reinvented regularization, which helps machine learning models not overfit the training data. To observe the effects of regularization, we can give *Polynomial Regression* to our linear regression learner, which supports these settings.

The Linear Regression widget provides two types of regularization. Ridge regression is the one we have talked about and minimizes the sum of squared coefficients θ . Lasso regression minimizes the sum of the absolute value of coefficients. Although the difference may seem negligible, the consequences are that lasso regression may result in a large proportion of coefficients θ being zero, in this way performing feature subset selection.



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

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

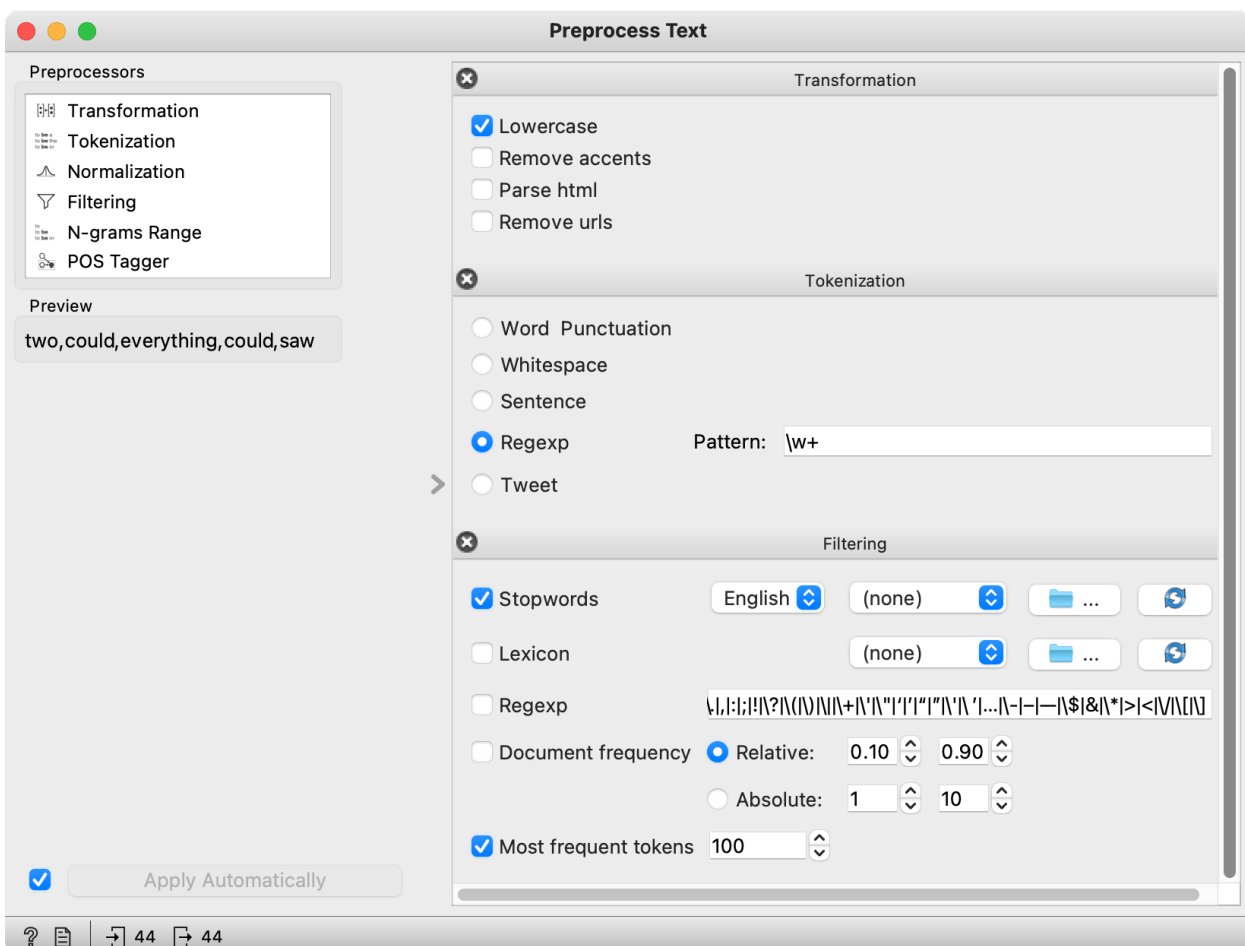


Network from Text

Now, let us try to generate networks from text. What do we mean by that? How can a text be transformed to a network? Well, documents are nodes, while edges can be the number of shared words. Again, we are back to similarity, but applied to text.

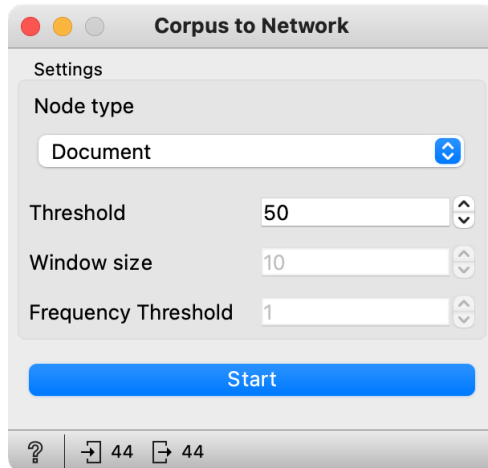
For this task, we will need the *Text add-on*. Load *grimm-tales-selected* with the *Corpus* widget. This data set contains 44 Grimm's tales, some of which are tales of magic and some are animal tales.

Every text needs to be preprocessed, that is we have to split the text to words and remove those words that have no meaning (such as stopwords). To speed up the analysis, we will keep only 100 most frequent tokens.

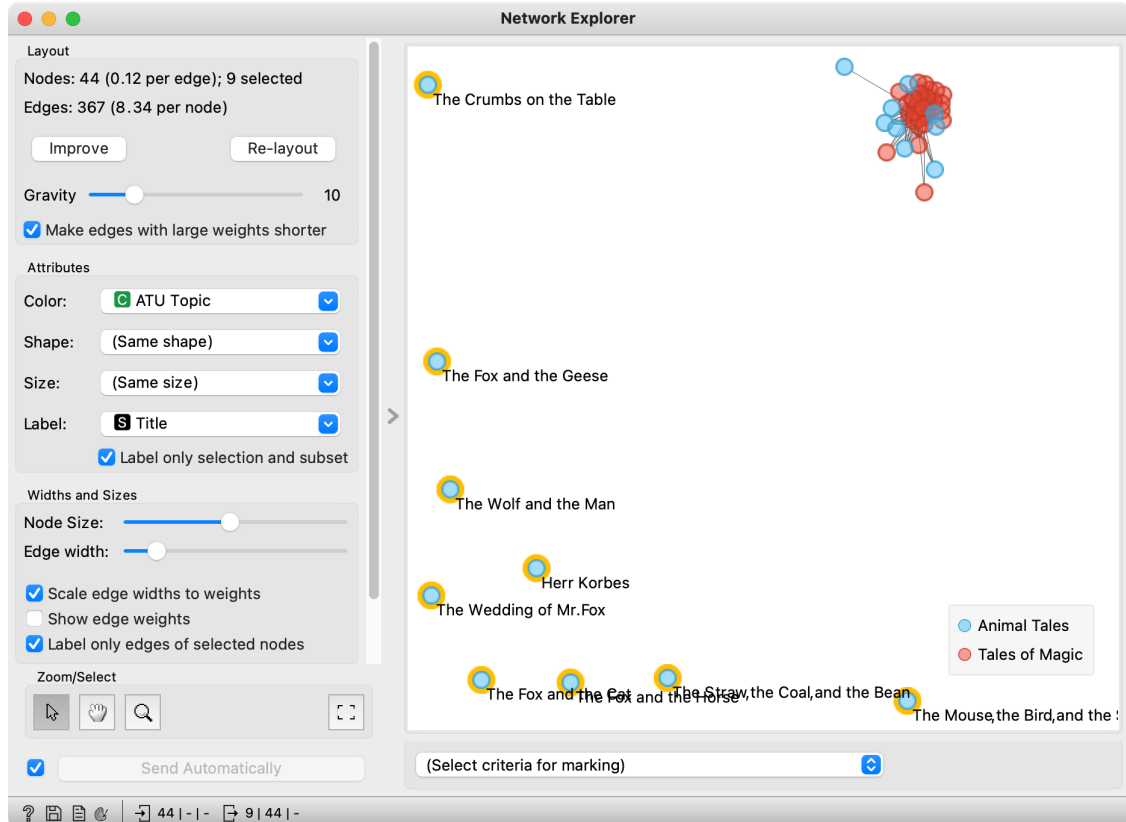


Now pass the data to *Corpus to Network*. With this widget, we will generate the graph. If we are generating network where nodes are documents, then we need to set a single parameter, namely the

threshold. This is similar to the similarity threshold in *Network from Distances*. *Threshold* will define how many words the documents have to share for them to have a connecting edge. In our case, we will set the threshold quite high - two documents have to share at least 50 words to be connected with an edge.

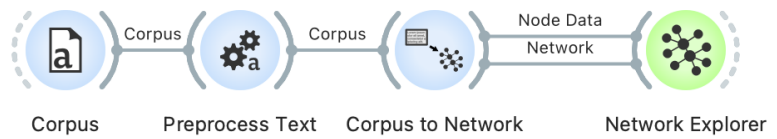


Let us observe the end result in *Network Explorer*. Seem like Tales of Magic are well-connected, even with some Animal Tale, while certain Animal Tales are quite distinct and don't share as many words with the other tales.



A task for the reader: play with the threshold and observe how

the graph changes. Does a lower threshold results in a more or less connected graph? What happens if words are used as Node types in *Corpus to Network*? What does such a graph show?



Bibliography

<https://github.com/biolab/orange3>, a.

<https://github.com/quasars>, b.

<https://quasar.codes>.

<https://quasar.codes>.

Robert Bringhurst. *The Elements of Typography*. Hartley & Marks, 3.1 edition, 2005. ISBN 0-88179-205-5.

Frank Mittelbach and Michel Goossens. *The L^AT_EX Companion*. Addison–Wesley, second edition, 2004. ISBN 0-201-36299-6.

Edward R. Tufte. *Envisioning Information*. Graphics Press, Cheshire, Connecticut, 1990. ISBN 0-9613921-1-8.

Edward R. Tufte. *Visual Explanations*. Graphics Press, Cheshire, Connecticut, 1997. ISBN 0-9613921-2-6.

Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, Connecticut, 2001. ISBN 0-9613921-4-2.

Edward R. Tufte. *Beautiful Evidence*. Graphics Press, LLC, first edition, May 2006. ISBN 0-9613921-7-7.

Hideo Umeki. The geometry package. <http://ctan.org/pkg/geometry>, December 2008.

Index

license, [2](#)