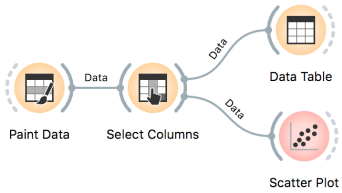
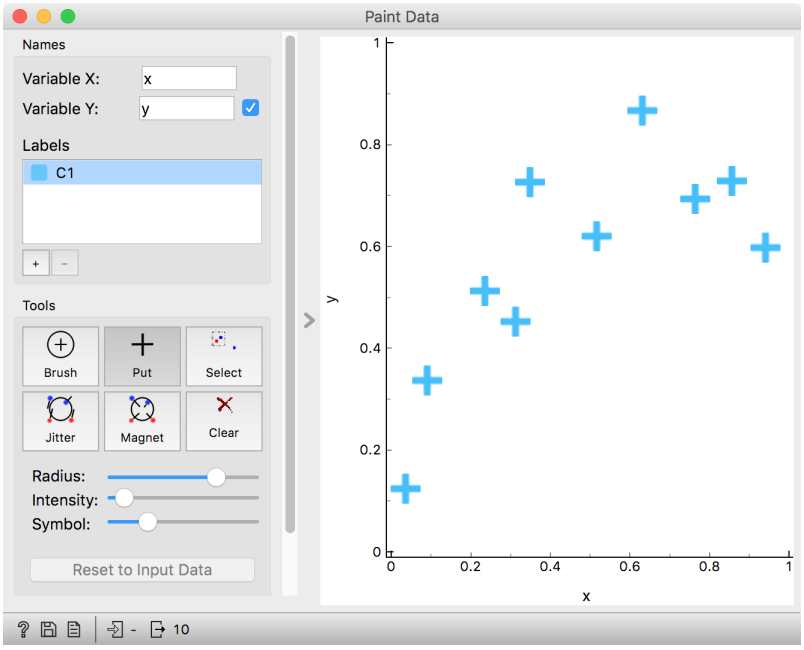
Linear Regression

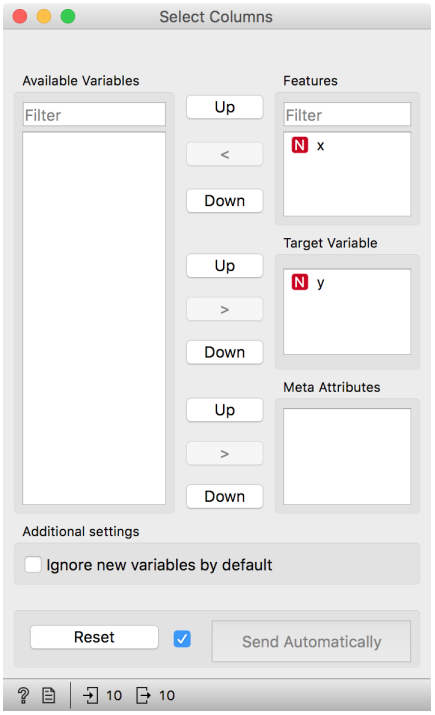
In the *Paint Data* widget, remove the C2 label from the list. If you have accidentally left it while painting, don't despair. The class variable will appear in the *Select Columns* widget, but you can "remove" it by dragging it into the Available Variables list.



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

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

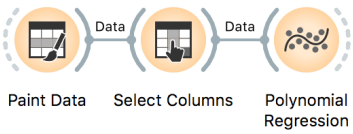


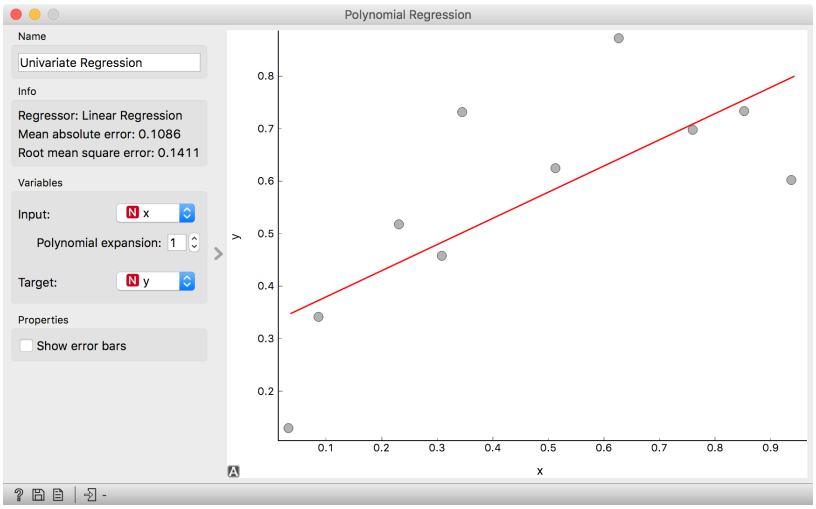


The question above requires us to define what a good fit is. Say, this could be the error the fitted model (the line) makes when it predicts the value of y for a given data point (value of x). The prediction is h(x), so the error is $h(x) - y$. We should treat the negative and positive

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

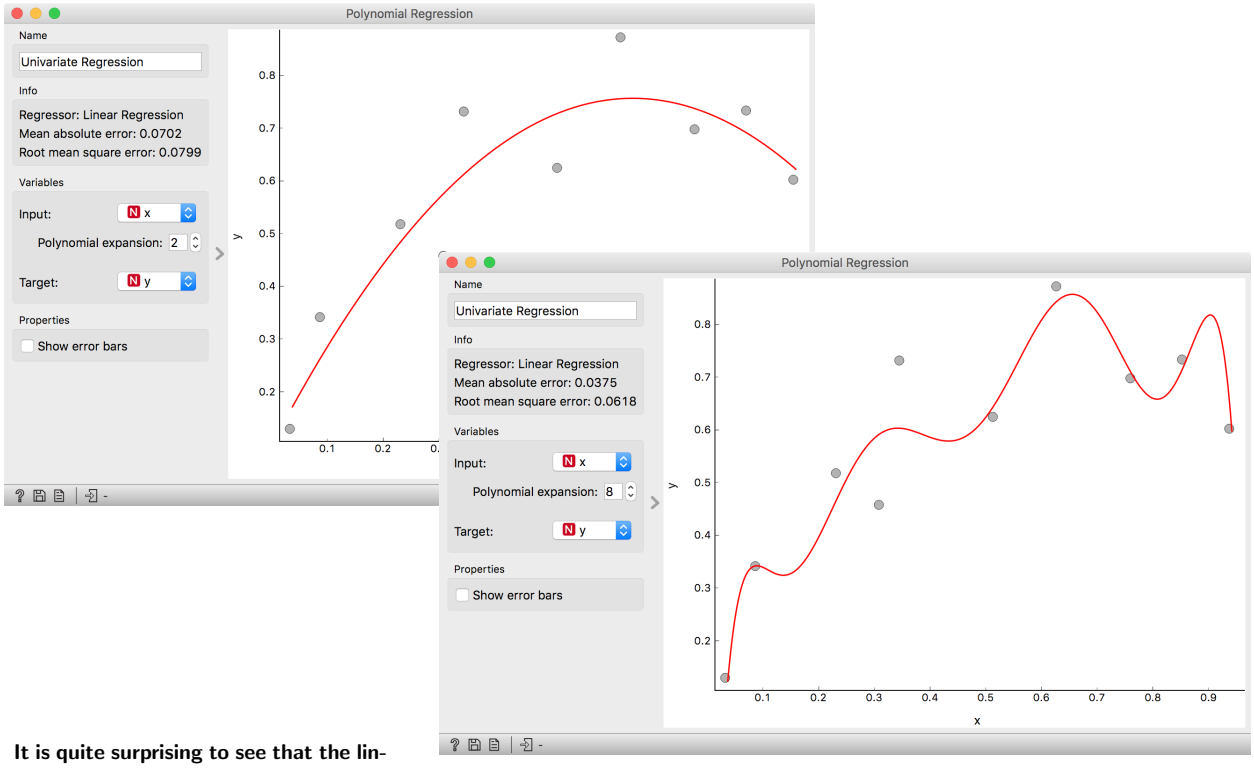
Do not worry about the strange name of the *Polynomial Regression*, we will get there in a moment.





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

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

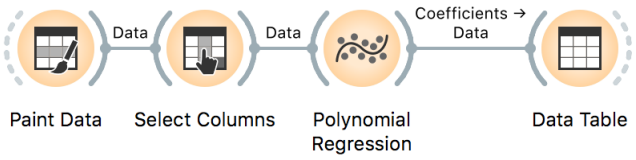


It is quite surprising to see that the linear regression model can fit non-linear (univariate) functions. It can fit the data with curves, such as those on the figures. How is this possible? Notice, though, that the model is a hyperplane (a flat surface) in the space of many features (columns) that are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

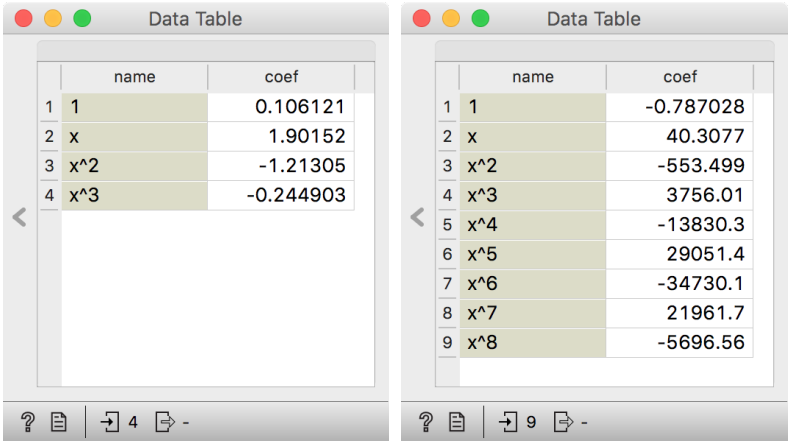
Overfitting is related to the complexity of the model. In polynomial regression, the parameters θ define the model. With the increased number of parameters, the model complexity increases. The simplest model has just one parameter (an intercept), ordinary linear regression has two (an intercept and a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

Regularization

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



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

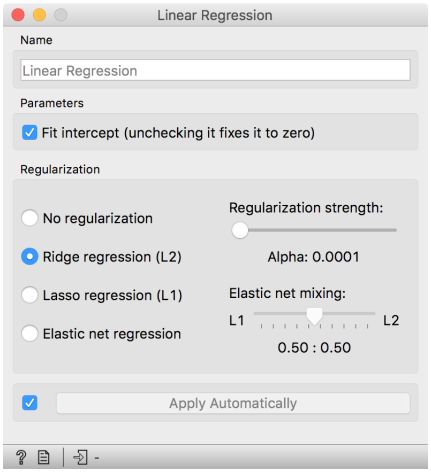


Which inference of linear model would overfit more, the one with high λ or with low λ? What should the value of λ be to cancel regularization? What if the value of λ is high, say 1000?

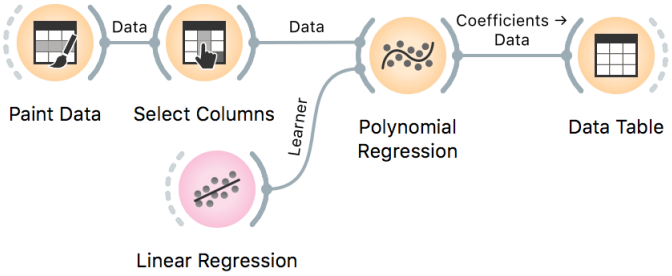
More complex models can fit the training data better. The fitted curve can wiggle sharply. The derivatives of such functions are high, so the coefficients θ need be. If only we could force the linear regression to infer models with a small value of coefficients. Oh, but we can. Remember, we have started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

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



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

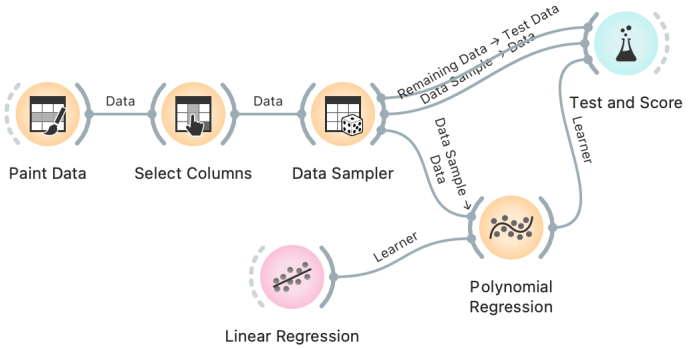


Now for the test. Increase the degree of polynomial to the max. Use Ridge Regression. Does the inferred model overfit the data? How does the degree of overfitting depend on regularization strength?

Regularization and Accuracy on a Test Set

Overfitting hurts. Overfit models fit the training data well but can perform miserably on new data. Let us observe this effect in regression. We will use hand-painted data set, split it into the training (50%) and test (50%) data set, polynomially expand the training data set to enable overfitting and build a model. We will test the model on the (seen) training data and the (unseen) held-out data.

Paint about 20 to 30 data instances. Use the attribute y as the target variable in Select Columns. Split the data 50:50 in Data Sampler. Cycle between test on train or test data in Test and Score. Use ridge regression to build a linear regression model.



Now we can vary the regularization strength in *Linear Regression* and observe the accuracy in *Test and Score*. For accuracy scoring, we will use RMSE, root mean squared error, which is computed by observing the error for each data point, squaring it, averaging this across all the data instances, and taking a square root.

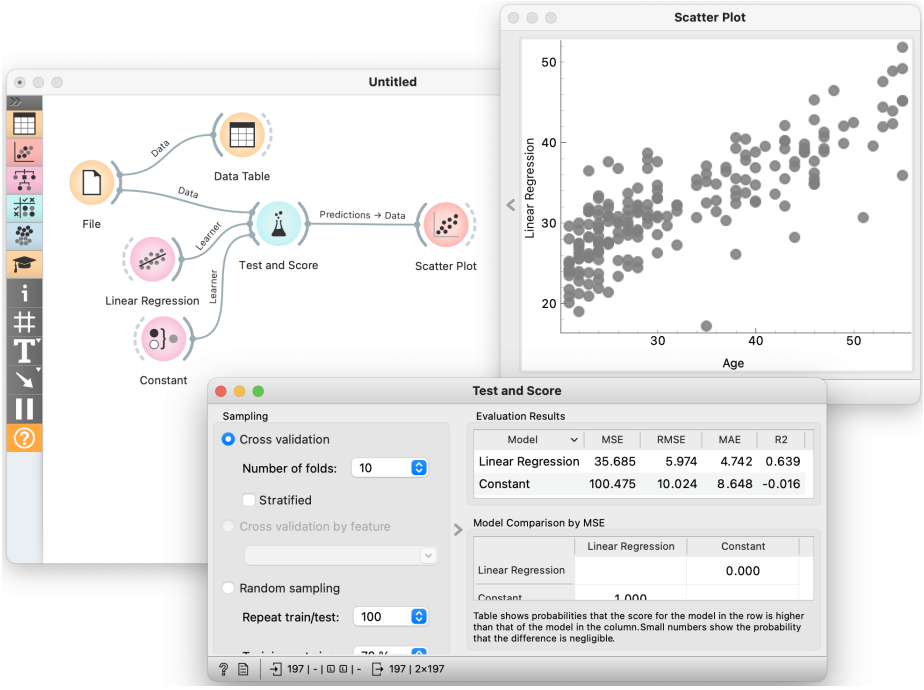
The core of this lesson is to compare the error on the training and test set while varying the level of regularization. Remember that regularization controls overfitting. The more we regularize, the less tightly we fit the model to the training data. So for the training set, we expect the error to drop with less regularization and more overfitting. The error on the training data increases with more regularization and less fitting. We expect no surprises here. But how does this play out on the test set? Which sides minimizes the test-set error? Or is the optimal level of regularization somewhere in between? How do we estimate this level of regularization from the training data alone?

Orange is currently not equipped with the fitting of meta parameters, like the degree of regularization, and we need to find their optimal values manually. At this stage, it suffices to say that we must infer meta parameters from the training data set without touching the test data. If the training data set is sufficiently large, we can split it into a set for training the model and a data set for validation. Again, Orange does not support such optimization yet, but it will sometime in the future. :)

Regularization

Download the methylation data set from `http://file.biolab.si/datasets/methylation.pkl.gz`. **Predictions of age from methylation profile were investigated by Horvath (2013) Genome Biology 14:R115.**

Enough painting. Now for the real data. We will use a data set that includes human tissues from subjects of different ages. The tissues were profiled by measurements of DNA methylation, a mechanism for cells to regulate gene expression. Methylation of DNA is scarce when we are young and gets more abundant as we age. We have prepared a data set where the degree of methylation was expressed per gene. Let us test if we can predict the age from the methylation profile - and if we can do this better than by just predicting the average age of subjects in the training set.



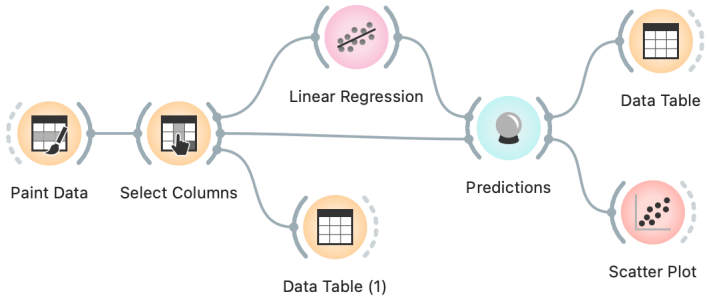
Using other learners, like random forests, takes a while on this data set. But you may try to sample the features, obtain a smaller data set, and try various regression learners.

This workflow looks familiar and is similar to those for classification problems—-the *Test and Score* widget reports on statistics we have not seen before. MAE, for one, is the mean average error. As for classification, we have used cross-validation. Mean average error was computed only on the test data instances and averaged across ten cross-validation runs. The results indicate that our modeling technique misses the age by about five years, which is a much better result than predicted by the mean age in the training set.

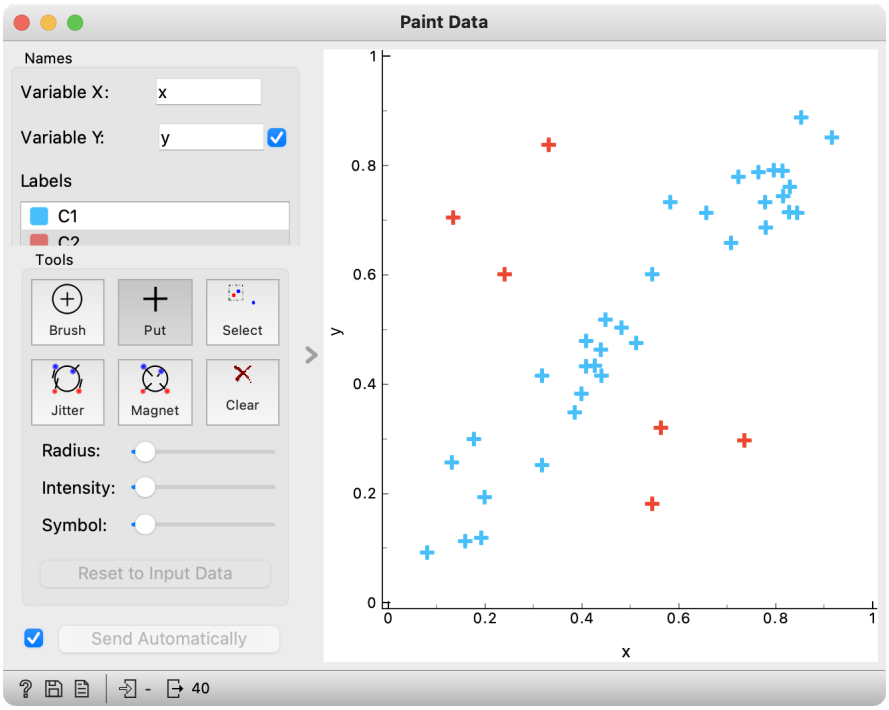
Evaluating Regression

The last lessons quickly introduced scoring for regression and essential measures such as RMSE and MAE. In classification, the confusion matrix was an excellent addition to finding misclassified data instances. But the confusion matrix could only be applied to discrete classes. Before Orange gets some similar for regression, one way to find misclassified data instances is through scatter plot!

This workflow visualizes the predictions that we have constructed on the training data. How would you change the widget to use a separate test set? Hint: The Sample widget can help.

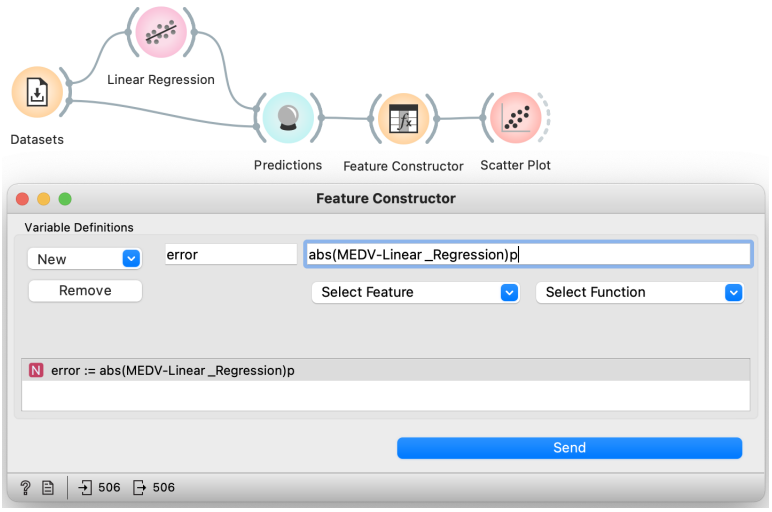


We can play around with this workflow by painting the data such that the regression would perform well on the blue data point and fail on the red outliers. In the scatter plot, we can check if the predicted and true class difference was what we had expected.



A similar workflow would work for any data set—for instance, the

housing data set (from Orange distribution). Say, just like above, we would like to plot the relationship between true and predicted continuous class, but would like to add information on the absolute error the predictor makes. Where is the error coming from? We need a new column. The Feature Constructor widget (albeit a bit geekish) comes to the rescue.

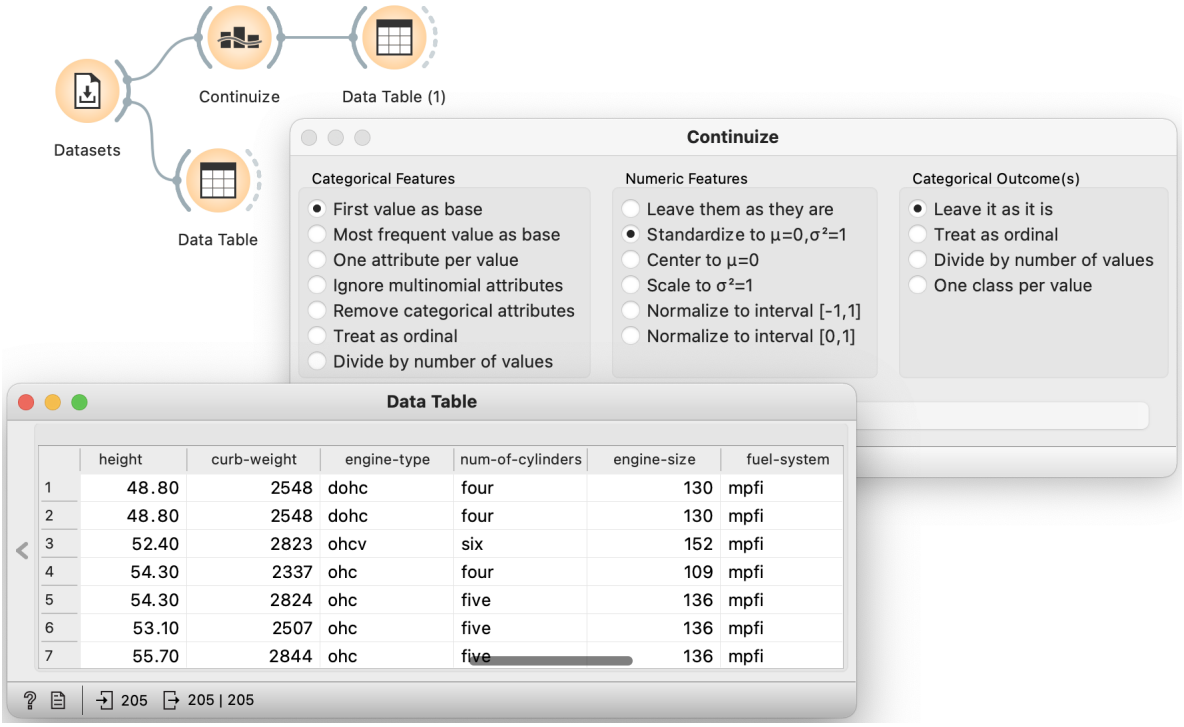


In the *Scatter Plot*, we can now select the data where the predictor erred substantially and explore the results further.

We could, in principle, also mine the errors to see if we can identify data instances for which this was high. But then, if this is so, we could have improved predictions at such regions. Like, construct predictors that predict the error. Creating such predictors looks weird. Could we then also build a predictor that predicts the error of the predictor that predicts the error? Strangely enough, such ideas have recently led to something called Gradient Boosted Trees, which are nowadays among the best regressors. Check them out using the Gradient Boosting widget.

Feature Scoring and Selection

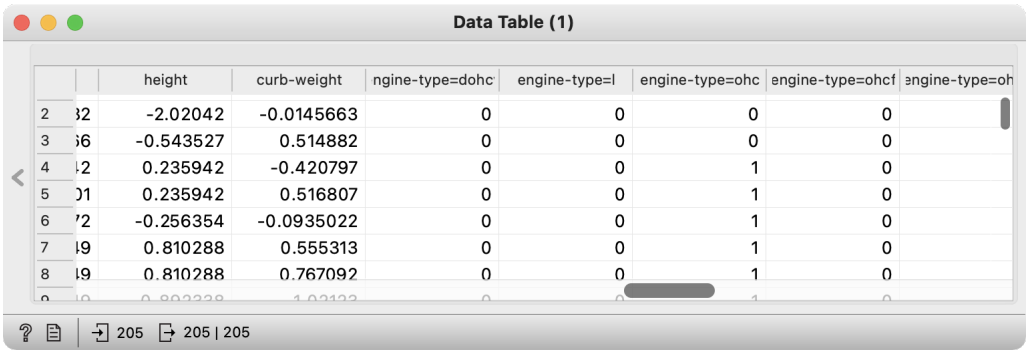
Linear regression infers a model that estimates the class, a real-valued feature, as a sum of products of input features and their weights. Consider the data on prices of imported cars in 1985. Inspecting this data set in a *Data Table* shows that some features, like fuel-system, engine-type, and many others, are discrete. Linear regression only works with numbers. In Orange, linear regression will automatically convert all discrete values to numbers, often using several features to represent a single discrete feature. We also do this conversion manually by using the *Continuize* widget.

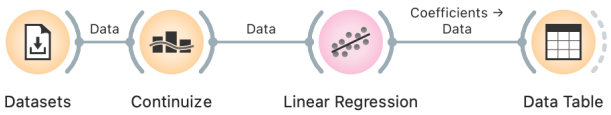


Before continuing, you should check what *Continuize* does and how it converts the nominal features into real-valued features. The table below should provide sufficient illustration.

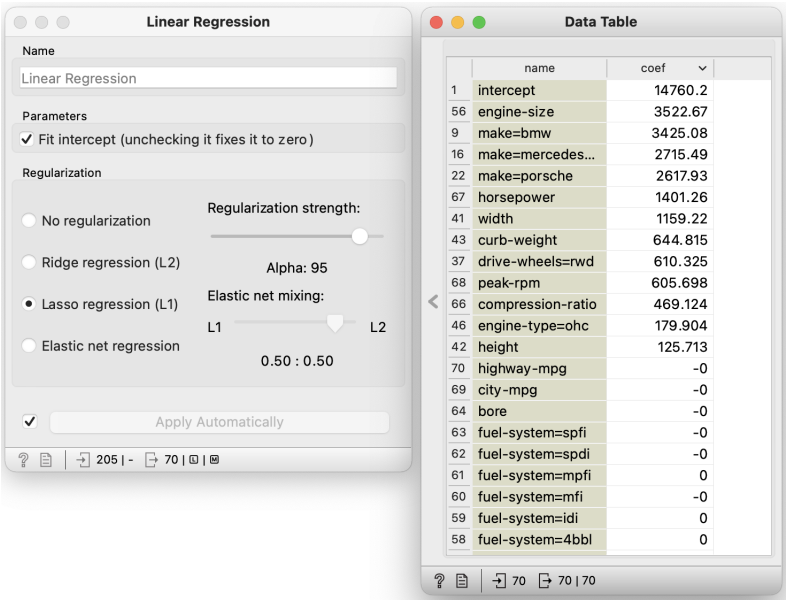
Now to the core of this lesson. Our workflow reads the data continuizes it such that we also normalize all the features to bring them to equal scale. We then load the data into the *Linear Regression* widget and check out the feature coefficients in the *Data Table*.

In *Linear Regression*, we will use L1 regularization. Compared to L2 regularization, which aims to minimize the sum of squared weights, L1 regularization is rougher: it minimizes the sum of absolute values of the weights. The result of this "roughness" is that many of the





features will get zero weights. But this feature elimination may also be exactly what we want. We want to select only the most important features and see how the model that uses only a smaller subset of features behaves. Also, this smaller set of features is ranked. Engine size is a huge factor in the pricing of our cars, and so is the make, where Porsche, Mercedes, and BMW cost more than other cars (ok, no news here).



We care about features with substantial weights, regardless of their sign. Therefore, we should change the workflow to compute and show the data features by their absolute weight. Could you change the workflow accordingly? Hint: use the Feature Construction widget.

We should notice that the number of features with non-zero weights varies with regularization strength. Stronger regularization would result in fewer features with non-zero weights.