WHITEPAPER

# How to Correctly Validate Machine Learning Models

Calculating model accuracy is a critical part of any machine learning project, yet many data science tools make it difficult or impossible to assess the true accuracy of a model. Often tools only validate the model selection itself, not what happens around the selection. Or worse, they don't support tried and true techniques like cross-validation.

This whitepaper discusses the four mandatory components for the correct validation of machine learning models, and how correct model validation works inside RapidMiner Studio.

# Introduction

Did you ever look forward to a watching a new movie, invite all your friends over to watch it with you, and then it turned out to be awful? Or did you ever buy a new vacuum cleaner based on some good reviews on Amazon only to find it breaks after a few uses? This is a situation where things look promising at first but then quickly collapse. Like a pair of cheap sneakers. Or Anakin Skywalker.

All data scientists have been in a situation where you think a machine learning model will do a great job of predicting something, but once it's in production, it doesn't perform as well as expected. In the best case, this is only an annoying waste of your time. But in the worst case, a model performing unexpectedly can cost millions of dollars – or potentially even human lives!

So was the predictive model wrong in those cases? Possibly. But often it is not the model which is wrong, **but how the model was validated.** A wrong validation leads to over-optimistic expectations of what will happen in production.

Since the consequences are often dire, I'm going to discuss how to prevent mistakes in model validation and the necessary components of a correct validation.

To kick off the discussion, let's get grounded in some of the basic concepts of validating machine learning models: predictive modeling, training error, test error and cross validation.

# Basics of Predictive Models and Validation

Before we dive deeper into the components of a correct validation without any surprises, this section describes the basic concepts of the validation of machine learning models. If you are familiar with the concepts of predictive modeling, training and test error, as well as cross validation, you might skip to the next section called "Accidental Contamination".

## What Models Can Be Validated?

Let's make sure that we are on the same page and quickly define what we mean by a "predictive model". We start with a data table consisting of multiple columns x1, x2, x3,... as well as one special column y. Below is a small example for such a table:

| **$_x1$** | **$x_2$** | **$x_3$** | **...** | **y** |
|---|---|---|---|---|
| 8 | 6 | 9 | ... | positive |
| 2 | 1 | 4 | ... | negative |
| ... | ... | ... | ... | ... |

**Table 1: A data table for predictive modeling. The goal is to find a function mapping the x-values to the correct value of y. Number of logical processors supported by the different editions of RapidMiner Studio.**

A ***predictive model*** is a function which maps a given set of values for the x-columns to the correct corresponding value for the y-column. Finding such a function for a given data set is called training the model.

Good models are not only avoiding errors for x-values they already know, but, in particular, they are also able to create predictions for situations which are only somewhat similar to the situations which are already stored in the existing data table. For example, such a model can predict that the y value for the x-values of (1, 2, 5,...) should be "negative," since those values are closer to the values in the second row of our table. This ability to generalize from known situations to unknown future situations is the reason we call this particular type of model predictive.

In this blog series, we'll focus only on predictive models for a column y with categorical values.  In principle, the validation concepts also hold if we want to predict numerical values (called regression), or if there is no such column at all (we call this ***unsupervised learning***).

# Training vs. Test Error

The one thing true for all machine learning methods, whether it is a decision tree or deep learning: you want to know how well your model will perform. You do this by measuring its accuracy.

Why? First of all, because measuring a model's accuracy can guide you to select the best-performing algorithm for it and fine-tune its parameters so that your model becomes more accurate. But most importantly, you will need to know how well the model performs before you use it in production. If your application requires the model to be correct for more than 90% of all predictions but it only delivers correct predictions 80% of the time, you might not want the model to go into production at all.

So how can we calculate the accuracy of a model? The basic idea is that you can train a predictive model on a given dataset and then use that underlying function on data points where you already know the value for ***y***. This results in two values of ***y***:  the actual one, as well as the prediction from the model, which we will call ***p***. The following table shows a dataset where we applied the trained model on the training data itself, leading to a situation where there is a new prediction ***p*** for each row:

| $x_1$ | $x_2$ | $x_3$ | … | y | p |
|---|---|---|---|---|---|
| 8 | 6 | 9 | … | positive | positive |
| 2 | 1 | 4 | … | negative | negative |
| … | … | … | … | … | |

Table 2: A table with training data. We created a predictive model and applied it on the same data which leads to a prediction for each row stored in column p. Now we can easily compare how often our predictions are wrong.

It is now relatively easy to calculate how often our predictions are wrong by comparing the predictions in **p** to the true values in **y** – this is called the classification error. To get the classification error, all we need to do is count how often the values for **y** and **p** differ in this table, and then divide this count by the number of rows in the table.

There are two important concepts used in machine learning: the training error and the test error.

- Training error. We get this by calculating the classification error of a model on the same data the model was trained on (just like the example above)
- Test error. We get this by using two completely disjoint datasets: one to train the model and the other to calculate the classification error. Both datasets need to have values for y. The first dataset is called training data and the second, test data.

Let's walk through an example of each. We will use the data science platform RapidMiner Studio to illustrate how the calculations and validations are actually performed. You can download RapidMiner Studio for free at http://www.rapidminer.com and follow along with these examples if you like.

Let's get started with the basic process you can use to calculate the training error for any given dataset and predictive model:

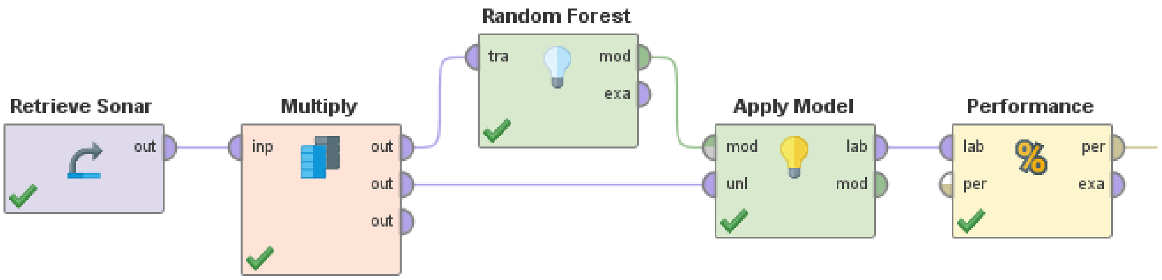


Figure 1: Creating a Random Forest model in RapidMiner Studio and applying it to the training data. The last operator called "Performance" then calculates the training error.

First we load a dataset ("Retrieve Sonar") and deliver this training data into a "Random Forest" operator and an "Apply Model" operator which creates the predictions and adds them to the input training data. The last operator on the right, called "Performance," then calculates the training error based on both the true values for y as well as the predictions in p.

Next let's look at the process to calculate the test error. It will soon be apparent why it is so important that the datasets to calculate the test error are completely disjoint (i.e., no data point used in the training data should be part of the test data and vice versa).

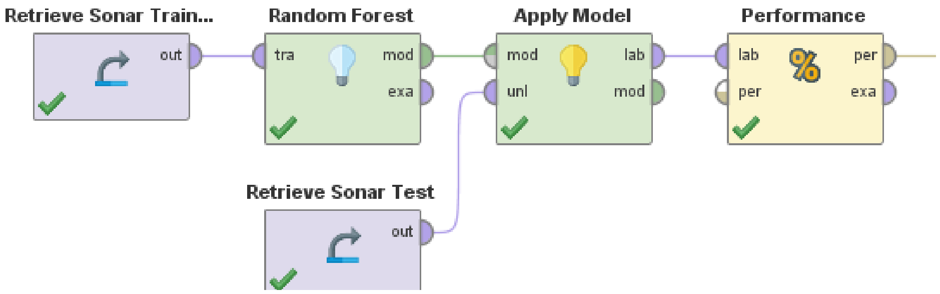


**Figure 2: Using two disjoint data sets for training the model and calculating the classification error leads to the so-called test error.**

Calculating any form of error rate for a predictive model is called model validation. As we discussed, you need to validate your models before they go into production in order to decide if the expected model performance will be good enough for production. But the same model performance is also often used to guide your efforts to optimize the model parameters or select the best model type. It is very important to understand the difference between a training error and a test error. ***Remember that the training error is calculated by using the same data for training the model and calculating its error rate. For calculating the test error, you are using completely disjoint data sets for both tasks.***

# Why You Should Ignore the Training Error

You will – unfortunately – find a lot of references in machine learning literature to training errors. This is a bad practice and should be avoided altogether. Training errors can be dangerously misleading! Let's demonstrate.

| $x_1$ | $x_2$ | y |
|---|---|---|
| -0.6 | 0.3 | positive |
| 0.7 | -0.1 | negative |
| (...random...) | (...random...) | (...random...) |

Table 3: A table with random data for the x-columns as well as for the classes in y.

The image below shows a chart of this data with the two x-columns on the axes of the chart and the random values for y used as color:

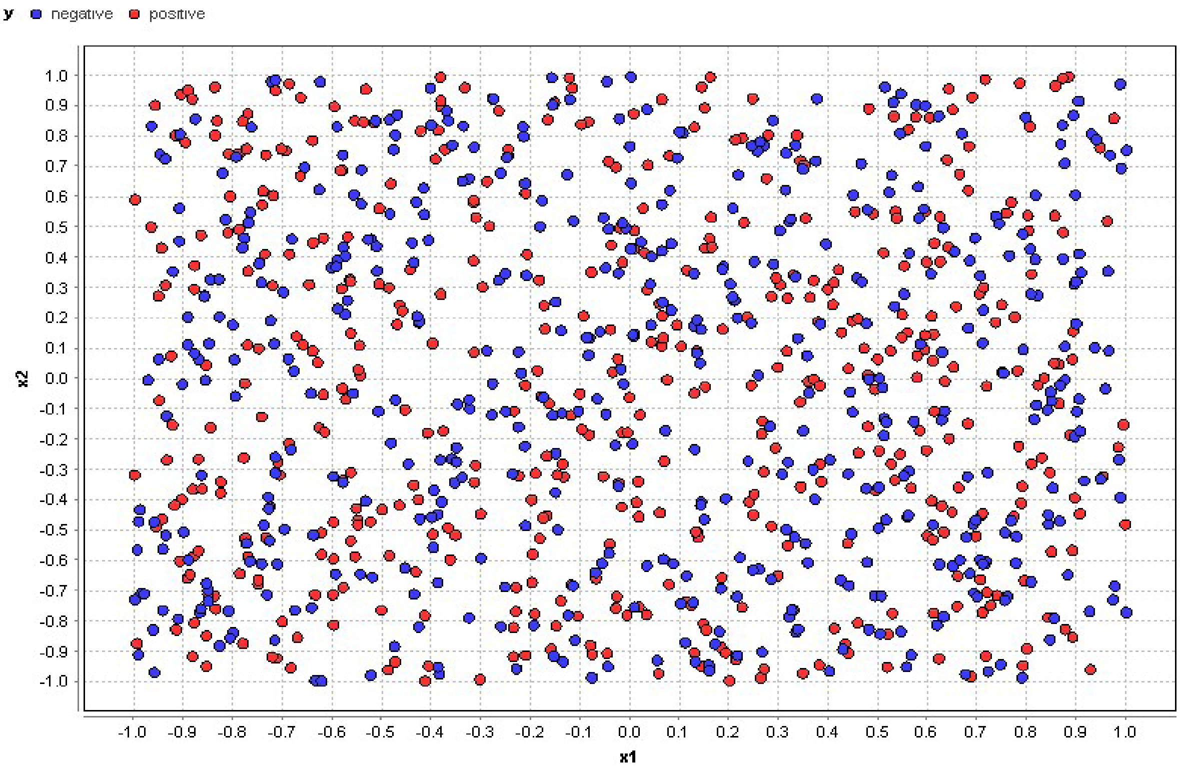


Figure 3: This is a data set in 2 dimensions where all values are completely random, including the class y which is depicted with the colors red (positive) and blue (negative).

The data set we have used for this image is 1,000 rows large and is also equally distributed, i.e.there are the same amount of 500 positive and 500 negative rows in this data. Now think about this: this data is completely random for both the positions of the points and the color of the points. Do you think it is possible to build a predictive model for such a random data set?

In fact, it is not. If the data is truly random, there is no pattern to be found and hence no function can be trained. The best you can achieve is to either produce random predictions or just go with one of the two classes "positive" or "negative" in all cases.

And what kind of classification error should such a predictive model have? Well, this is easy. Both classes are equally distributed (500 out of 1,000 rows for each class) and if we always predict one class in all cases, we should be correct in exactly half of all the cases and wrong for the other half, i.e. we should end up with a classification error of 50%.

So let's do a little experiment and calculate the training error and test error for a widely used machine learning method, k-Nearest Neighbors. Here's how this algorithm works: all data points from the training set are simply stored during the training phase. When the model is applied to a new data point, the predictive function looks for the k most similar data points in the stored training data and uses their classes (or the majority of the classes in case they are different.

And here is the result: the training error of the 1-Nearest Neighbor classifier is 0% (!) – a perfect classifier! And much better than the 50% error we have expected. How is this possible?

The reason is simple: if you apply a 1-Nearest Neighbor classifier on the same data you trained it on, the nearest neighbor for each point will always be the point itself. Which, of course, will lead to correct predictions in all cases, even though this is random data. But for data points which are not part of the training data, the randomness will kick in and the 0% classification error you hoped for will quickly turn into the 50% we expected. And this is exactly the kind of negative surprise you can get if you make the decision to go into production based on validating the model's performance on the results of the training error. Using the training error gives you an inaccurate model validation.

Below is the RapidMiner Studio process we have used to calculate the training error for the 1-Nearest Neighbors classifier on the random data set. It delivers a 0% classification error.

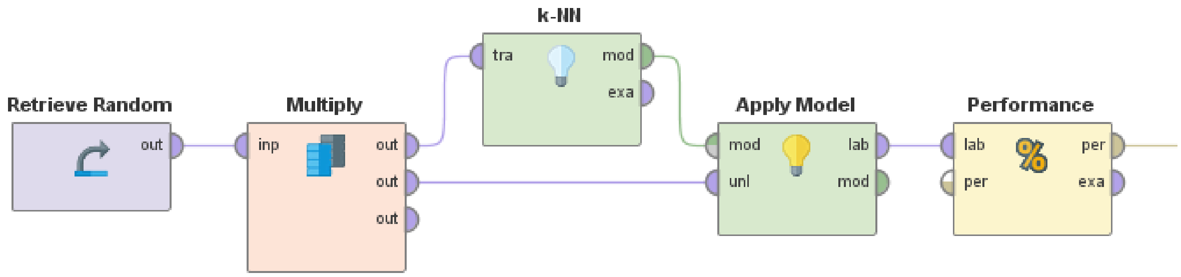


**Figure 4: Calculating the training error of a 1-Nearest Neighbor classifier always leads to a 0% training error - even on complete random data.**

We can compare this 0% error of the 1-Nearest Neighbors classifier now to the test error calculated on a completely independent test set but following the same random data generation principles. The process below is using two disjoint data sets of random data, both with a size of 1,000 rows:

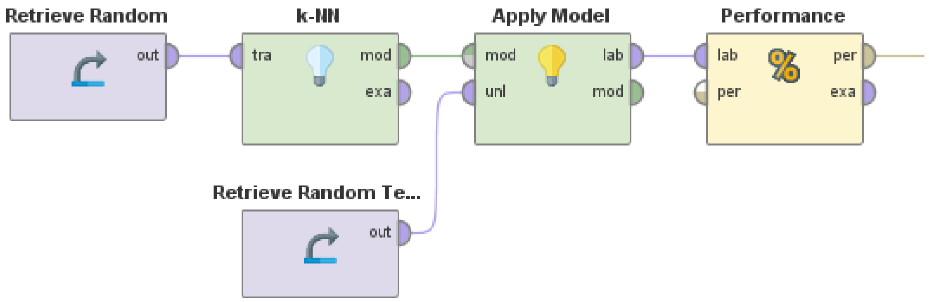


**Figure 5: Calculating the test error of 1-Nearest Neighbors for random data shows a completely different picture: we get a test error of 51.5% - close to our expectation of 50%.complete random data.**

This process actually delivers a test error of exactly 51.5% which is very close to our expected error of 50%. You can see that the test error is a much better estimation about how your model will perform on unseen data than the training error which would give you the false believe of a perfect model.

Now, you might argue that a 1-Nearest Neighbor classifier is not a realistic approach (which I would argue against!) and that data typically is not random (which I would accept for many cases). So how do the training error and the test error look different for more realistic scenarios? The table below shows the difference between both error types for three different machine learning methods on four different datasets from the UCI data set repository for machine learning:

| Data Set | Default | Random Forest | | Logistic Regression | | 5-NN | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| Diabetes | 34.9% | 32.6% | 27.0% | 32.6% | 23.2% | 28.7% | 19.7% |
| Ionosphere | 35.9% | 17.1% | 4.6% | 18.1% | 12.3% | 21.9% | 12.5% |
| Sonar | 46.6% | 32.3% | 14.4% | 29.0% | 18.3% | 25.8% | 13.5% |
| Wine | 22.6% | 18.4% | 14.5% | 11.8% | 10.1% | 21.9% | 10.2% |
| Avg. Dev. | | | 9.98% | | 6.90% | | 10.60% |

**Table 4: The table shows the test and the training errors for three different machine learning methods on four different data sets. The differences are sometimes dramatic (up to 18%) with an average of almost 10%.**

The column "default" is the error rate you would get if you went with just the majority class as prediction in all cases. You can clearly see that all machine learning methods have been able to improve the test error from the default error in all cases, so the model has clearly learned something useful. But you can also see how far off the training errors are – in fact, none of them would turn out to be true if you went into production. The test errors on the other hand will at least be very close to what you will observe in production. On average, the deviation between the two error types is about 10% - in one case, namely Random Forest on the Sonar data set, the difference went as high as 18%!Table 2: ***The table shows the test and the training errors for three different machine learning methods on four different data sets. The differences are sometimes dramatic (up to 18%) with an average of almost 10%.***

One thing is very important: the best thing we can do is **deliver an *estimation*** about how well the model will perform in the future. If done in the right way, this estimation will be close to what can be achieved but there is no guarantee that the estimated performance will be exactly what can be expected.

In any case, however, the ***test error*** is a much better estimation about how well the model will perform for new and unseen cases in the future. The ***training error*** is not helpful at all, as we have clearly seen above. That's why I recommend that you ***don't use training errors at all.*** They are misleading because they always deliver an overly optimistic estimation about model accuracy.

# Validating Using Hold-Out Data Sets

Now that we've established that using the training error is a terrible idea, let's dive deeper into different ways how you can calculate test errors. The first thing to notice is that it is often very difficult or expensive to get more data where you have values for y.

For example, if you create a model to predict which customers are more likely to churn, then you build the model on data about the people who have churned in the past. Generating more churn data is exactly what you try to prevent so this is not a good idea. Another example is a predictive maintenance use case where you predict if and when a machine needs maintenance or will fail. Nobody in the right mind will purposely break more machines just to create more training data!

So training data is expensive and generating more of it for testing is difficult.

It is a good practice in such cases to use a part of the available data for training and a different part for testing the model. This part of the data used for testing is also called a hold-out data set. Practically all data science platforms have functions for performing this data split. In fact, below is the RapidMiner Studio process we have used to calculate the test errors for the data sets in the previous section:

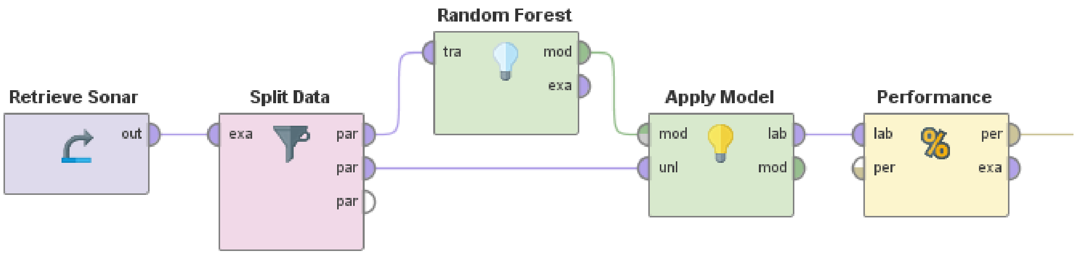


**Figure 6: The available training data is split into two disjoint parts, one is used for training and the other one for the testing model.**

Of course you have a conflict here.  Typically, a predictive model is better the more data it gets for training. So this would suggest to use as much data as possible for training. At the same time, you want to use as much data as possible for testing in order to get reliable test errors. Often a good practice is to use 70% of your data for training and the remaining 30% for testing.

# Cross-Validation as the Gold Standard

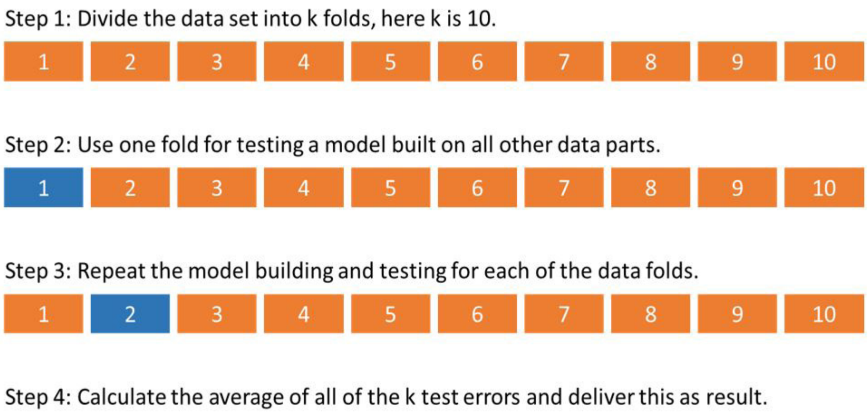
Using a hold-out data set from your training data in order to calculate the test data is an excellent way to get a much more reliable estimation on the future accuracy of a model. But still there is a problem: how do we know that the hold-out set was not particularly easy for the model? It could be that the random sample you selected is not so random after all, especially if you only have small training data sets available. You might end up with all the tough data rows for building the model and the easy ones for testing – or the other way round. In both cases your test error might be less representative of the model accuracy than you think.

One idea might be to just repeat the sampling of a hold-out set multiple times and use different samples each time for the hold-out set. For example, you might create 10 different hold-out sets and 10 different models on the remaining training data sets. And in the end you can just average those 10 different test errors and will end up with a better estimate which is less dependent on the actual sample of the test set. This procedure has a name – repeated hold-out testing. It was the standard way of validating models for some time, but nowadays it has been replaced by a different approach.

Although in principle the averaged test errors on the repeated hold-out sets are superior to a single test error on any particular test set, it still has one drawback: we will end up with some data rows being used in multiple of the test sets while other rows have not been used for testing at all. As a consequence, the errors you make on those repeated rows have a higher impact on the test error which is just another form of a bad selection bias. Hmm... what's a good data scientist to do?

The answer: k-fold cross-validation.

With k-fold cross-validation you aren't just creating multiple test samples repeatedly, but are dividing the complete dataset you have into k disjoint parts of the same size. You then train k different models on k-1 parts each while you test those models always on the remaining part of data. If you do this for all k parts exactly once, you ensure that you use every data row equally often for training and exactly once for testing. And you still end up with k test errors similar to the repeated holdout set discussed above. The picture below shows how cross-validation works in principle:



Step 1: Divide the data set into k folds, here k is 10.

Step 2: Use one fold for testing a model built on all other data parts.

Step 3: Repeat the model building and testing for each of the data folds.

Step 4: Calculate the average of all of the k test errors and deliver this as result.

**Figure 7: Principle of a k-fold cross-validation. Divide the data into k disjoint parts and use each part exactly once for testing a model built on the remaining parts.**

For the reasons discussed above, a k-fold cross-validation is the go-to method whenever you want to validate the future accuracy of a predictive model. It is a simple method which guarantees you that there is no overlap between the training and test sets (which would be bad as we have seen above!) It also guarantees that there is no overlap between the k test sets which is good since it does not introduce any form of negative selection bias. And last but not least, the fact that you get multiple test errors for different test sets allows you to build an average and standard deviation for these test errors. This means that instead of getting a test error like 15% you will end up with an error average like 14.5% +/- 2% giving you a better idea about the range the actual model accuracy will likely be in when put into production.

Despite all its obvious advantages, a proper cross-validation is still not implemented in all data science platforms on the market which is a shame and part of the reason why many data scientists fall into the traps we are discussing in this white paper. The images below show how to perform a cross validation in RapidMiner Studio:

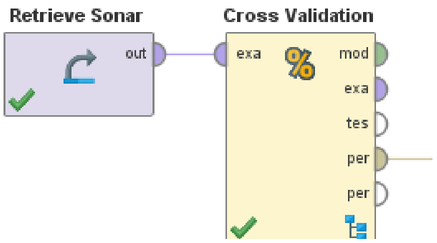


Figure 8: A cross-validation operator in RapidMiner Studio. The operator takes care of creating the necessary data splits into k folds, training, testing, and the average building at the end.

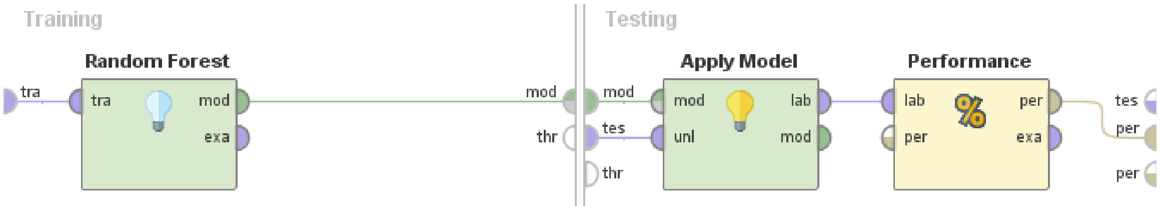


Figure 9: : The modular approach of RapidMiner Studio allows you to go inside of the cross-validation to change the model type, parameters, or even perform additional operations.

Before we move on to the next section, let's now also perform a cross validation on our four data sets from above using our three machine learning models Random Forest, Logistic Regression, and a k-Nearest Neighbors learner with k=5. Here are the results:

| Data Sets | Random Forest | Logistic Regression | 5-NN |
|---|---|---|---|
| Diabetes | 27.6% +/- 4.5% | 24.4% +/- 4.2% | 28.3% +/- 3.0% |
| Ionosphere | 8.8% +/- 5.2% | 14.2% +/- 5.8% | 15.7% +/- 4.8% |
| Sonar | 33.3% +/- 9.3% | 25.9% +/- 8.9% | 19.3% +/- 12.6% |
| Wine | 16.1% +/- 1.8% | 10.9% +/- 3.4% | 16.3% +/- 3.3% |

**Table 5: Test errors of three machine learning methods on four data sets. Calculated with a 10-fold cross-validation. Each cell shows the average test error of all folds plus the standard deviation.**

If you compare the averages test errors above with the single fold test errors we calculated before, you can see that the differences are sometimes quite high. In general, the single test errors are in the range of one standard deviation away from the average value delivered by the cross validation but the differences can still be dramatic (see for example Random Forest on Ionosphere). This is exactly the result of the selection bias and the reason you should always go with a cross-validation instead of a single calculation on only one hold-out data set.

But there is also a drawback which is the higher runtime. Performing a 10-fold cross-validation on your data means that you now need to build 10 models instead of one, which dramatically increases the computation time. If this becomes an issue, you will see the number of folds being decreased to values as little as 3 to 5 folds instead.

## KEY TAKEAWAYS

- In machine learning, training a predictive model means finding a function which maps a set of values x to a value y.
- We can calculate how well such a model is doing by comparing the predicted values with the true values for y.
- If we apply the model on the data it was trained on, we can calculate the training error.
- If we calculate the error on data which was unknown in the training phase, we can calculate the test error.
- You should never use the training error for estimating how well a model will perform. In fact, it is better to not care at all about the training error.
- You can always build a hold-out set of your data not used for training in order to calculate the much more reliable test error Cross-validation is a perfect way to make full use of your data without leaking information into the training phase. It should be your standard approach for validating any predictive model.

# Accidental Contamination

You learned in the previous section to calculate the predictive accuracy of a model by applying the model on data points it has not previously been trained on, and compare the model's predictions with the known true result for this test set. If you repeat this multiple times for non-overlapping test cases, just like you do in case of the cross-validation, you end up with a reliable estimation about how well your model will perform on new cases. This approach in general is a good basis for doing model selection, i.e. answering the question which type of model (think: "random forest or linear regression?") is going to perform best on my data in the future.

So all is good then, right? Wrong! The problem is that it is still very easy to leak information about the testing data into the training data if you perform a cross-validation in the wrong way. We call this phenomenon contamination of the training data. Contamination provides access to information the machine learning method should not have access to during training.  With contamination, the model will perform better than you expect it to perform when compared to situations when this information is not available.  This is exactly why accidental contamination leads to an over-optimistic estimation about how well the model will perform in the future.

This effect can be as drastic in the example of the k-Nearest Neighbor classifier above, where the training error (all information is available in this case) was 0% while the testing error was 50% and hence no better than randomly guessing the class of the data points.  As we will see below, the same effect happens if you leak information about the test data into the training phase.

The test error becomes effectively a training error then and is lower than you would expect without the contamination. Therefore, all the efforts you did by using a cross-validation to avoid leakage of information is pointless if you are doing this wrong. And unfortunately, most data science platforms do not even support the correct way of performing a cross-validation in the proper way. This is the main reason why so many data scientists make this mistake.

Let's talk show a few typical situations where accidental contamination of training data by leaking information about the test data can happen. This is by no means a complete list of examples, but it should be enough to give you the general idea about the problem and what to look for if you want to avoid this.

# Example 1: Contamination through Normalization

Let's start with a very common situation: you would like to normalize data so that all columns have a similar range and no column overshadows others. This is particularly important before using any similarity-based models like, for example, k-Nearest Neighbors.

This seems to be a simple task: just normalize the data and then train the model and validate it with a cross-validation. This is how this would look like in RapidMiner:

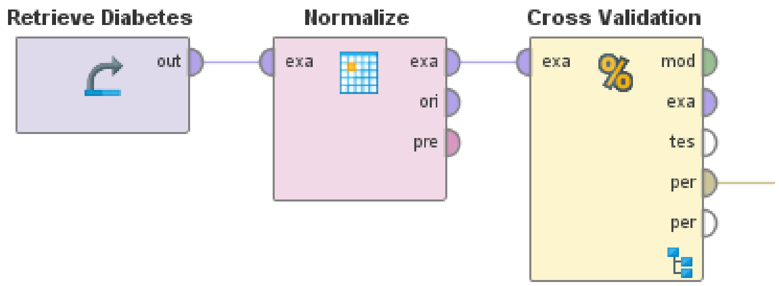


**Figure 10: This RapidMiner Studio process simply performs a normalization (z-transformation) on the data before the model is validated. The inner part of the cross-validation looks exactly like in Figure 9.**

Looks good you say? Wrong! Because the approach shown in Figure 10 will lead to a wrong error estimation. If you normalize the data before the cross-validation, you actually leak information about the distribution of the test data into the way you normalize the training data. Although you are not using the test data for training the model, you nevertheless instill some information about it into the training data. This is exactly why we call this effect contamination.

What you should do instead is to perform the normalization inside of the cross-validation and only on the training data. You then take the information you gathered about the training data and use it for the transformation of the test data. This is where the RapidMiner Studio visual interface comes in handy since you can easily see the correct setup in the image below:

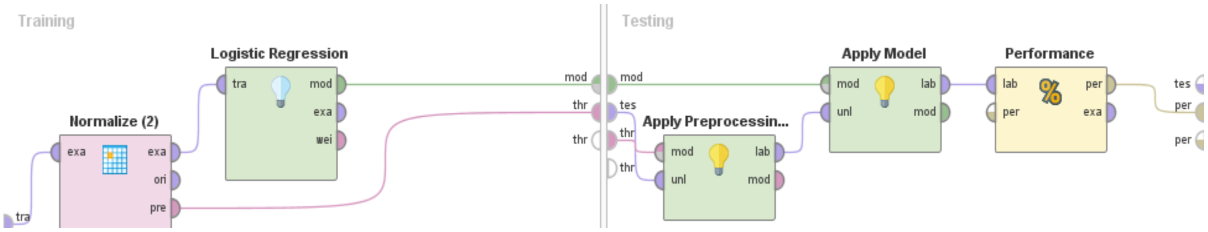


**Figure 11: This is how you properly evaluate the impact of the normalization on the model accuracy. The parameters derived from the normalization on the training data are delivered into the test phase and used there instead of the other way around.**

Please note that we are feeding data directly into the cross-validation operator and performing all necessary steps inside. The first thing you do in the training phase on the left is to normalize only the training data. The transformed data is then delivered to the machine learning method (Logistic Regression in this example). Both the predictive model and the transformation parameters from the normalization are then delivered into the testing phase on the right. Here we first transform the test data based on the distributions found in the training data before we apply the predictive model on the transformed test data. Finally, we calculate the error as usual.

The table below summarizes the cross-validated test errors, one time with the normalization performed before the cross-validation like in Figure 10 and one time with the normalization inside of the cross-validation like in Figure 11. It clearly shows the effect on the error if you validate the model without measuring the impact of the data pre-processing itself.

| Data Sets | Random Forest | | Logistic Regression | | 5-NN | |
|---|---|---|---|---|---|---|
| | | | | | | |
| Diabetes | 27.5% | 27.5% | 24.1% | 29.6% | 26.6% | 28.1% |
| Ionosphere | 11.4% | 11.4% | 15.7% | 16.8% | 15.4% | 19.4% |
| Sonar | 30.7% | 30.7% | 24.0% | 32.2% | 18.7% | 26.0% |
| Wine | 16.5% | 16.5% | 11.1% | 11.9% | 11.6% | 12.2% |
| Avg. Dev. | | **0%** | | **3.9%** | | **3.4%** |

**Table 6: The cross-validated test errors with a normalization before the cross-validation ("Wrong") or inside. You can clearly see that the contamination of training data in the "Wrong" cases leads to overly optimistic error estimations.**

If the normalization is done before the cross-validation, the calculated error is too optimistic in all cases and the data scientist would run into a negative surprise when going into production. On average, the actual error is 3.9% higher for Logistic Regression and 3.4% higher for a k-Nearest Neighbors with k=5. The difference is 0% for Random Forest simply because the models do not change at all if the data is normalized or not. In general, you can see that the effect is not as drastic as just using the training error, but still there are differences of more than 8% in some of the cases, higher than most people would expect. This is caused by only validating the model itself and not the impact of the data pre-processing.

A final thing which is important to notice: the model will actually perform roughly the same way in production, no matter if you perform the normalization outside or inside the cross-validation. When building a production model, you would typically use the complete dataset anyway and hence also apply the normalization on the complete data. But the performance of the model will be the lower one, pretty much in the range of the one shown in the column "Nested" in the table above. So the correct validation is not helping you create better models, instead it is telling you the truth about how well (or poorly) the model will work in production without letting you run into a catastrophic failure later on.

# Example 2: Contamination through Parameter Optimization

Next let's look at the impact of optimizing parameters of the model, e.g., in case of Random Forest the number of trees in the ensemble or in the case of k-Nearest Neighbors the parameter k determining how many neighbors are used for finding the prediction. Data scientists often search for optimal parameters to improve a model's performance. They can do this either manually by changing parameter values and measuring the change of the test error by cross-validating the model using the specified parameters. Or they can use automatic approaches like grid searches or evolutionary algorithms. The goal is to always find the best performing model for the data set at hand.

As above in the case of normalization, using the cross-validated test errors found during this optimization sounds like a good plan to most data scientists – but unfortunately, it's not. This should become immediately clear if you think of a (automatic) parameter optimization method just as another machine learning method which tries to adapt a prediction function to your data. If you only validate errors inside of this "bigger" machine learning method and deliver those errors to the outside, you effectively turn the inner test error into a kind of overly optimistic training error.

In case of parameter optimization, this is a well-known phenomenon in machine learning. If you select parameters so that the error on a given test set is minimized, you are optimizing this parameter setting for this particular test set only – hence this error is no longer representative for your model. Many data scientists suggest to use a so-called validation set in addition to a test set. So you can first use your test set for optimizing your parameter settings, and then you can use your validation set to validate the error of the parameter optimization plus the model on this set.

The problem with this validation set approach is that it comes with the same kind of single-test-set-problems we discussed before. You won't get all the advantages of a proper cross-validation if you use validation sets only.

But you can do this correctly by nesting multiple levels of cross-validation into each other. The inner cross-validation is used to determine the error for a specific parameter setting, and can be used inside of an automatic parameter optimization method. This parameter optimization becomes your new machine learning algorithm. Now you can simply wrap another, outer cross-validation around this automatic parameter optimization method to properly calculate the error which can be expected in production scenarios.

This sounds complex in writing but again a visual representation can help us to understand this correct setup. Figures 12 and 13 show the outer cross validation with the parameter optimization inside:

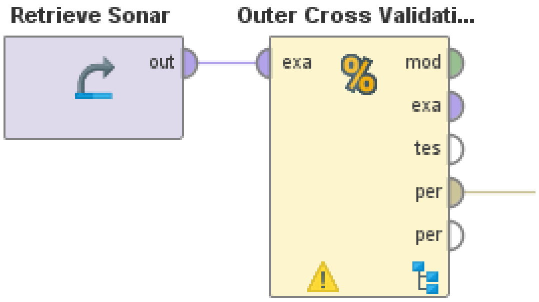


Figure 12: Properly validating a parameter optimization requires two cross-validations, an inner one to guide the search for optimal parameters and an outer one (shown here) to validate those parameters on an independent validation set.

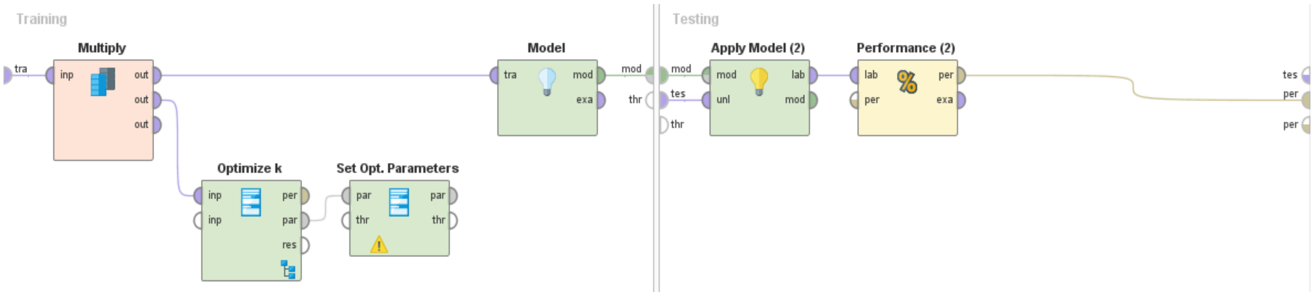


Figure 13: Inside of the outer cross validation. In the training phase on the left is an automated search for an optimal parameter set which uses another, inner cross-validation (not shown). Then the optimal parameters are used for evaluating the final model.

Since the need for an independent validation set is somewhat known in machine learning, many people believe that the impact of correct validation on the errors must be quite high. But it turns out that while there definitely is an impact on the test errors, it is smaller than the impact on the correct test error in the normalization case above. The following table shows the results:

| Data Sets | Random Forest | | Logistic Regression | | k-NN | |
|---|---|---|---|---|---|---|
| | | | | | | |
| Diabetes | 27.1% | 27.5% | 23.2% | 23.6% | 24.5% | 25.5% |
| Ionosphere | 8.3% | 9.4% | 12.0% | 13.4% | 16.0% | 16.5% |
| Sonar | 25.5% | 25.5% | 22.1% | 22.1% | 19.8% | 19.8% |
| Wine | 15.5% | 15.7% | 10.2% | 10.7% | 16.4% | 16.5% |
| Avg. Dev. | | 0.4% | | 0.6% | | 0.4% |

**Table 7: We used an automatic grid search for parameter optimization. The column “Wrong” shows the naive approach of showing the cross-validated error which was delivered from the validation guiding the search. “Nested” shows the correct estimations.**

As expected, the properly validated errors using the “Nested” approach as described above are higher than the errors taking directly out of the validation used for guiding the search (“Wrong” in the table above). But the differences are rather small with roughly 0.5% difference on average and a maximum deviation of 1%. But still, even a difference of 1% between your expectation and how well the model performs in production can be a huge and costly surprise.

# Example 3: Contamination through Feature Selection

As a last example, let's look at another pre-processing step which is frequently performed for optimizing the accuracy of machine learning models, namely feature selection. The goal is to select an optimal subset of features or data columns used by the model. In theory, many models can make this selection themselves, but in reality noisy columns can throw models off and it's better to keep only those columns which provide meaningful information.

Similar to the parameter optimization, data scientists could manually select a subset of features and evaluate the model's accuracy by calculating a cross-validated test error using only the data of the reduced feature set. Since the number of combinations grows exponentially with the number of columns in the input data, this isn't often a feasible approach; the reason why automated optimization techniques are widely used. But just as in the case of parameter optimization, picking the optimal feature set is basically an extension of training the predictive model, and it needs to be properly validated as well.

Of course you could use an extra hold-out set for evaluating any built model at the end. But as before, you end up with the disadvantage of only using a single validation set. The better alternative is to nest the automatic feature selection (using an inner cross-validation to guide the search for the optimal set of features) into an outer cross-validation again, just as we did in case of the parameter optimization.

The correct setup starts with an outer cross-validation just as described above where we used an outer cross-validation to evaluate the performance of the parameter optimization. The inner setup of this cross-validation looks slightly different though, as follows:

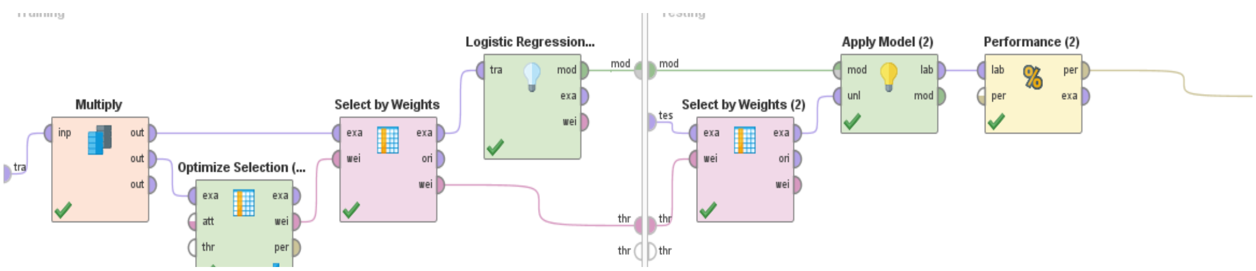


Figure 14: An automatic evolutionary feature selection is used on the training data of the outer cross validation. The optimal feature set is selected before the model is trained and delivered to the test phase where it is applied again before prediction.

And here are the results for the wrong version of the cross-validation where you don't properly validate the effect of the feature selection and only use the performance value of the optimal feature set as found during the optimization runs. The value in the columns "Nested" are determined with the correct approach using an outer cross-validation as depicted above:

| Data Sets | Random Forest | | Logistic Regression | | 5-NN | |
|---|---|---|---|---|---|---|
| | | | | | | |
| Diabetes | 25.5% | 26.3% | 22.5% | 23.6% | 25.0% | 26.2% |
| Ionosphere | 7.1% | 13.7% | 16.5% | 17.7% | 8.8% | 13.1% |
| Sonar | 22.2% | 26.7% | 22.5% | 23.7% | 11.6% | 22.6% |
| Wine | 12.3% | 12.3% | 13.0% | 13.0% | 10.0% | 10.1% |
| Avg. Dev. | | 3.0% | | 0.9% | | 4.2% |

**Table 8: We used an automatic evolutionary feature selection. The column "Wrong" shows the naive approach of showing the cross-validated error which was delivered from the validation guiding the search while "Nested" delivers the proper estimations.**

Here you see that the amount by which your expectations would be off is on average up to 4.2%. The largest deviation was even as high as 11% (5-NN on Sonar) – which is even higher than the maximum of 8% we saw in the normalization case above. Think about this: your error in production would be double (22% vs. 11%) what you expected; just because you did not the right kind of validation which takes into account the effect of the data pre-processing. You can very easily avoid this type of negative surprise!

It is worth pointing out that the calculated average and maximum deviations greatly depend on the data sets and model types you use. We have seen above that the effect of the wrong validation for normalization was zero for Random Forest since this model type does not care about the different scales of the data columns. On the other hand, the effect was larger for feature selection using a Random Forest. And this is the problem: you never know in advance how large the effect will really be. The only way to be prepared about what is coming in the future is to properly validate the effects of the model itself as well as all other optimizations with a correct form of modular, nested cross-validations.

# KEY TAKEAWAYS

- You can contaminate your training data set by applying data transformations before the cross-validation which leads to information leakage about the test data into the complete data set.
- Think about pre-processing, parameter optimizations, or other (automatic) optimization schemes just as another form of machine learning trying to find the best predictive model.
- This “bigger” machine learning needs to be evaluated just like the actual machine learning method itself.
- Most data science products do not allow you to perform the model validation in a correct way, i.e. taking into account the effect of pre-processing or model optimizations.
- We have seen three examples showing how large the effect on model validation can be when done incorrectly.
- For an un-validated normalization, the effect is on average up to 3.9% with a maximum of 8%.
- For an un-validated parameter optimization, the effect is on average up to 0.6% with a maximum of 1%.
- For an un-validated feature selection, the effect is on average up to 4.2% with a maximum of 11%.
- It is important to understand that these effects can be even higher for different data sets or different forms of un-validated pre-processing.
- The only way to avoid surprises when going into production is to properly validate before.

# Consequences of Accidental Contamination

Just to be 100% clear: we are not saying that you cannot build a good model if you are not properly validating your models. You absolutely can, but without a proper validation you won't be aware if your supposedly good model will be only mediocre before you go into production use – you will only realize this after you already are in production! And this can be a costly error as this section describes. The decision to go into production with a model should be based on a correct validation taking into account the effect of not just the model selection, but also of all things you do around the selection of the right model.

Let me start with two examples. Let's first consider a machine learning model which is taking images from an MRI scan to detect malign tumor cells. It is amazing how machine learning can support doctors in identifying risk areas in the massive amounts of data those machines produce today. Although the final decision about treatment lies with the doctor, the identification of risk areas is heavily supported by algorithms. There are two types of mistakes the predictive model can make: the patient actually has cancer but it remains undetected, or the patient has no cancer but the model causes a false alarm. Now let's assume that the data scientist made a cross-validation but accidentally contaminated the training data like we have described above. With this wrong validation, the machine learning model is found to only make errors in 0.1% of all cases. This is better than the performance of even the best doctors, so this model goes into production. The expectation is that only one out of a thousand patients will suffer from undetected cancer cells or from a false alarm (which of course still is the better case). But as it turns out, some contamination as described above led to an over-optimistic estimation, and a check with the first treated 1,000 patients showed that not one but one hundred patients actually have been misclassified. The error rate of this model was actually 10%, which would have been detected with a correct form of validation. Now a hundred people won't receive the proper treatment because of undetected cancer cells or might suffer from a treatment they didn't need.

Besides life-threatening outcomes, an over-optimistic validation might cause costly errors also in a business context. Let's imagine a company is losing $200 Million per year due to customer churn. A machine learning model has been created and – with an improper validation – has shown to reduce this churn rate by 20%, i.e. by $40 Million if the right measurements are put into place for select customers. Those measurements incur costs of $20 Million, but given the reduction in churn volume of $40 Million, it's still a very good investment. But as it turns out, the model was not properly validated and after spending the $20 Million, the reduction in churn was only 5%, i.e. $10 Million revenue savings from reduced churn instead of the expected $40 Million. As a consequence, the expected $20 Million gain has turned into a real $10 Million loss. The decision for this unsuccessful investment would never have been made if the correct model accuracy would have been known beforehand. This is why doing the validation correctly, not just building great models, is so incredibly important.

## KEY TAKEAWAYS

- Not doing a proper validation does not mean that your models are not performing.
- It means that your models look better than they will actually perform in practice.
- The negative consequences of this can be huge.

# How to Avoid Accidental Contamination

As we have seen before, the problem is that most people only validate the model selection itself, not what happens around this selection. A common pattern is to perform data or model changes before you validate the model on the complete data set.  The examples above demonstrated this clearly, for example in the simple case of performing a data normalization before the model is evaluated in a cross-validation.

If you perform the validation in programming languages like R or Python, you need to make sure that your code validates your model the correct way so that calculated errors are correct and not too optimistic. This is difficult and error-prone, and a common reason why visual products for data science like RapidMiner Studio are perfect for building the correct workflows as we have seen this in the Figures above. But it turns out that many visual products fall somewhat short in supporting proper model validation.

Any visual data science product needs four main components to perform correct validations:

**1. Proper validation schemes like cross-validation:** This should be table-stakes for all data science products by now, but it unfortunately is not.  Many visual products (like Alteryx) do not support cross-validation for all the machine learning models they offer.  Products not offering this functionality should be avoided in all serious data science scenarios.

**2. Modular validation schemes:** This means that the cross-validated test error is not just delivered automatically or if a parameter of the machine learning method is switched on, but the validation method is explicitly modeled.  If the error is just delivered automatically (or requested by the user with a parameter setting), then this accuracy can only be used for the model selection itself, and is dangerous to be used for guiding any optimization around the model selection. New entrants in the data science market like Microsoft Azure Machine Learning or Dataiku fall short, and do not properly measure the impact of parameter changes or pre-processing methods like discussed above. Although this over-simplified approach might be sufficient for dipping your toes into data science, it certainly leads to high risks for production usage.

**3. Multi-level nesting:** To measure the effect of activities outside model selection, you need to be able to nest multiple cross-validations into each other.  This is especially true if you want to measure the impact on multiple optimizations simultaneously. You can perform a feature selection and a parameter optimization for the same model, using one inner cross-validation to guide the parameter optimization and feature selection, and an outer cross-validation to measure the impact of both optimizations.  Even long-time vendors in the data science market like SAS or IBM SPSS do not offer this functionality in their visual environments.  Others, like Datarobot, claim that they do the right thing “automagically”, but unfortunately in a completely hidden way so that you need to trust their black-box approach. Our experiences showed that betting on data science black boxes is rarely a good idea. Offering this functionality is what truly separates the most powerful products with more accurate (and honest) results from the mediocre ones. If you want the full confidence that your models will perform as expected, there is simply no way around a multi-level nesting approach for modular cross-validations.

**4. Pre-processing models:** The visual platform needs enough capabilities to apply exactly the same preprocessing you applied on the training data also on a different test data set. This is often relatively well-supported for simple preprocessing tasks like selecting a subset of features, but often there is no support for applying the same normalization factors (like mean and standard deviation values from the training data to apply the same z-Transformation) or other more complex data space transformations. Storing the necessary information for later application is what we call a pre-processing model. Most visual data science products do not support pre-processing models at all, while some products (like Knime) support at least the most basic transformations but not all the necessary transformations occurring in non-scientific scenarios.

As pointed out above, most visual data science products do unfortunately not support all four components – some do not even support the necessary basics like a cross-validation scheme for all machine learning methods. We have mentioned above specific products as examples, and explained where they fall short on at least one of these requirements.

This then forces you to either accept wrong and over-optimistic performance validations or alternatively start the error-prone process of coding the validation part yourself.  But this of course renders the benefits of the visual environment pretty much moot and is a very inefficient approach.
RapidMiner Studio is the only visual data science and machine learning product which is well-known for supporting all four components above. They are all a must to create correct validations that give you the necessary level of confidence in your models before you go into production usage with them. Don't settle for less – millions of dollars (or even human lives) depend on the correct validation of your models and the predictions they deliver.

## KEY TAKEAWAYS

- Pay attention if you code the validation with scripting languages like R or Python – you need to take care of the correct, non-contaminating validation yourself which can be complex.
- Many visual data science products fall short on the four necessary components for correct model validations.
- You need support for proper validation schemes like cross-validation.
- The validation methods need to be modular i.e. you can change what happens inside.
- The validation methods need to support multiple levels i.e. you can nest one validation into another.
- You need pre-processing models for all data transformations and other model optimizations.
- RapidMiner Studio is the only visual product supporting all four necessary elements for correct model validations.

rapidminer

10 Milk Street, 11th Floor
Boston, MA 02108