Table of Contents

[Section – Adding it to your local RapidMiner Studio (10 minutes) 1](#_Toc7592837)

[Section - Why Ignore Training Error? (15-20 mins) 3](#_Toc7592838)

[Group Task 1: Training error and the test error look different for more realistic scenarios 6](#_Toc7592839)

[Section – Methods to calculate Test error (15-20 mins) 7](#_Toc7592840)

[A. Validate Using Holdout Datasets 7](#_Toc7592841)

[B. Repeated Hold-Out Testing 8](#_Toc7592842)

[C. k-fold cross-validation. 8](#_Toc7592843)

[Group Task 2: 10 fold CV error estimates 10](#_Toc7592844)

# Section – Adding it to your local RapidMiner Studio (10 minutes)

To add the repository, you need to unzip the folder provided over Moodle.

Go back to RapidMiner Studio and click on Create Repositories:

Graphical user interface, text, application, Word

Description automatically generated

You need to choose "New Local Repository" in the opening dialogue:

Graphical user interface, text, application

Description automatically generated

and choose the path to the unzipped directory:

Graphical user interface, text, application

Description automatically generated

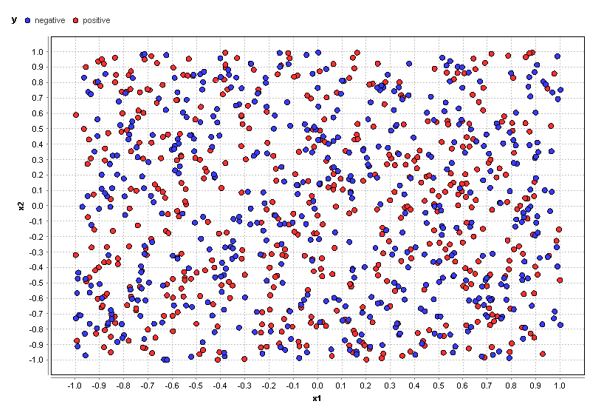
Note that the Alias can be anything you like.

# Section - Why Ignore Training Error? (15-20 mins)

Requirements: You will be using the following:

1. Processes:
   * 01 Training Error for 1-NN on Random Data, and
   * 02 Test Error for 1-NN on Random Data
2. Datasets (csv files):
   * Random, and Random Test
   * Sonar Training and Sonar Test

**Objective:** 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. Here our objective is to fully understand the underpinnings of the problem.



**Figure 1:** The data set Random in 2 dimensions where all values are completely random, including the class y which is depicted with the colors red (positive) and blue (negative).

**Overview:** Considering the data set “Random” 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.

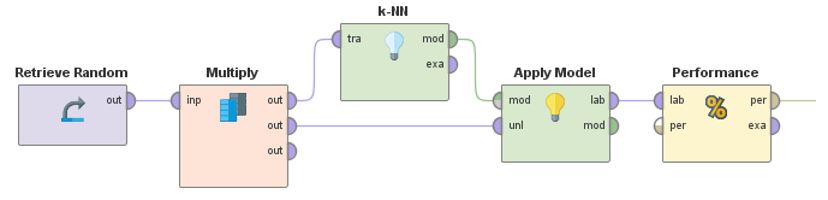
Do you think it is possible to build a predictive model for such a random data set?

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, that’s 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%.

**Experiment:** to calculate the training error and test error using the machine learning algorithm **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)



**Do the following:**

**Load the process:** 01 Training Error for 1-NN on Random Data

**Run the process, and**

**Observe the performance** of the model in the results tab

**The result:** the training error of the 1-Nearest Neighbor classifier is 0% (!) – a perfect classifier!

Much better than the anticipated 50% error. 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.

**Do the following:**

**Load the process:** 02 Test Error for 1-NN on Random Data

**Run the process, and**

**Observe the performance**: Classification error \_\_\_\_\_51.5\_\_\_\_\_\_ %

Discuss:

* Do the results represent a more realistic estimate of classification error?

## Task 1: Training error and the test error look different for more realistic scenarios

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dataset | Random Forest | | 5-NN | |
| Training | Testing | Training | Testing |
| Sonar | 0.00% | 29.03% | 12.33% | 25.81% |

**Conclusion:** 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 it is recommend that you don’t use training errors at all. They are misleading because they always deliver an overly optimistic estimation about model accuracy.

**Key Takeaway:** You should “never” use the training error for estimating how well a model will perform. In fact, it’s better to ignore the training error all together.

# Section – Methods to calculate Test error (15-20 mins)

Requirements: You will be using the following:

1. Processes:
   * 03 Hold-out data set split, and
   * 04 Cross-validation
2. Datasets:
   * Sonar, and
   * Diabetes, and Ionosphere.

Now that we’ve established that using the training error is a terrible idea, let’s dive deeper into different ways to calculate test errors.

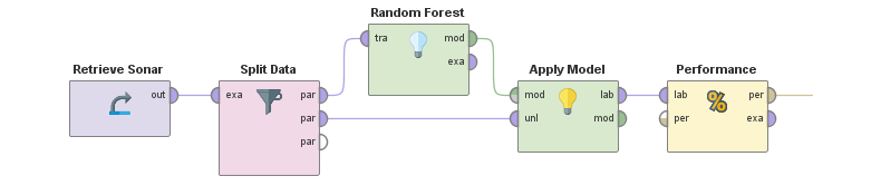
## Validate Using Holdout Datasets

The first thing to note is that it is often very difficult or expensive to get labeled data. 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.
* In predictive maintenance case you want to 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 **holdout dataset**. Practically all data science platforms have functions for performing this data split. In fact, below is the RapidMiner Studio process we used to calculate the test errors for the datasets in the previous section:

****

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

**Do the following:**

**Load the process:** 03 Hold-out data set split

**Run the process, and**

**Observe the performance** of the model in the results tab

Of course, you have a conflict here. Typically, a predictive model is better the more data it gets for training. So, this would suggest that you use as much data as possible for training. At the same time, you want to use as much data as possible to test 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.

## Task 2: Hold-out data set split error estimates

|  |  |  |
| --- | --- | --- |
| Dataset | Random Forest | 5-NN |
| Hold-out error and standard deviation | Hold-out error and standard deviation |
| Diabetes | 32.61% | 28.70% |
| Ionosphere | 8.57% | 21.90% |
| Sonar | 24.19% | 25.81% |

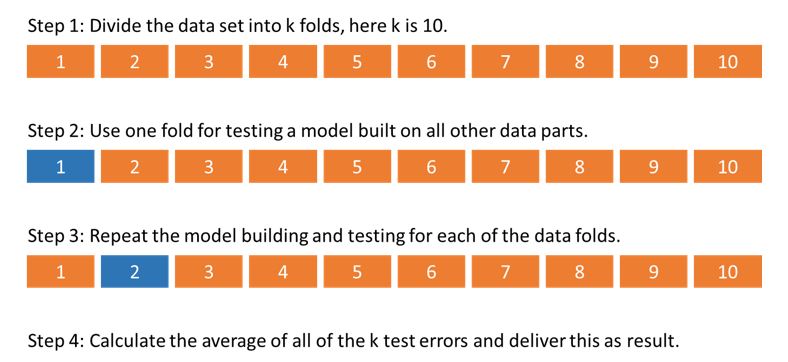
## k-fold cross-validation.

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**:**

****

**Figure 2:** 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:**

* 1. **that there is no overlap between the training and test sets.**
  2. that there is **no overlap between the k test sets which is good since it does not introduce any form of negative selection bias**.
  3. 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.

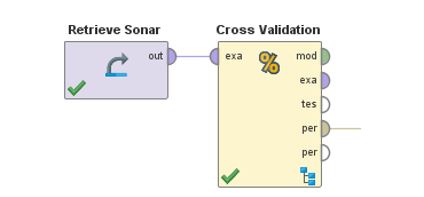
This means that instead of getting a test error like 15% you will end up with an error average like 14.5% +/- 2% giving you a better idea about the range the actual model accuracy will likely be when you put into production.

**Do the following:**

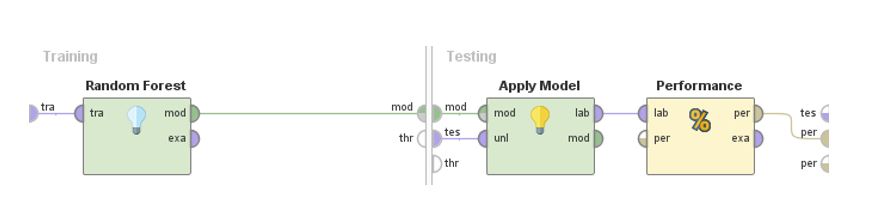
**Load the process:** 04 Cross-validation

**Run the process, and**

**Observe the performance** of the model in the results tab

****

**Figure 3:** 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 4:** 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 conclude this section, using the two machine learning models Random Forest, and k-Nearest Neighbors learner with k=5. Carry out the following group task on the three datasets

## Task 3: 10 fold CV error estimates

|  |  |  |
| --- | --- | --- |
| Dataset | Random Forest | 5-NN |
| Cross validation error and standard deviation | Cross validation error and standard deviation |
| Diabetes | 25.92% +/- 5.34% | 28.25% +/- 3.13% |
| Ionosphere | 5.7% +/- 5.04% | 15.66% +/- 5.04% |
| Sonar | 24.98% +/- 8.62% | 19.29% +/- 13.32% |

**Observations:**

* + - 1. Variance in error: If you compare the average test errors above with the single fold test error we calculated before, you can see that the differences are sometimes high. 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.
      2. 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**

* You can always build a holdout 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.