**DATA COLLECTION**

**How to Split a Dataset into Training and Testing Sets with Python?**

Exploring three ways of creating train and test samples out of a modelling dataset



Photo by [Siora Photography](https://unsplash.com/@siora18) on [Unsplash](https://unsplash.com/photos/bvfLgWu31t8)

In the context of Machine Learning, the split of our modelling dataset into training and testing samples is probably one of the earliest pre-processing steps that we need to undertake. The creation of different samples for training and testing helps us evaluate model performance.

In this article, we will discuss the purpose of training and testing samples in the context of modelling and model training. Additionally, we are going to explore three easy ways one can use to create such samples using Python and pandas. More specifically, we will showcase how to create training and testing samples:

* Using scikit-learn (aka sklearn) train\_test\_split()
* Using numpy ‘s randn () function
* or with built-in panda’s method called sample ()

**Why do we need train and test samples?**

A very common issue when training a model is **overfitting**. This phenomenon occurs when a model performs really well on the data that we used to train it but it fails to generalise well to new, unseen data points. There are numerous reasons why this can happen — it could be due to the noise in data or it could be that the model learned to predict specific inputs rather than the predictive parameters that could help it make correct predictions. Typically, the higher the complexity of a model the higher the chance that it will be over fitted.

On the other hand, **under fitting** occurs when the model has poor performance even on the data that was used to train it. In most cases, under fitting occurs because the model is not suitable for the problem you are trying to solve. Usually, this means that the model is less complex than required in order to learn those parameters that can be proven to be predictive.

Creating different data samples for training and testing the model is the most common approach that can be used to identify these sort of issues. In this way, we can use the training set for training our model and then treat the testing set as a collection of data points that will help us evaluate whether the model can generalise well to new, unseen data.

The simplest way to split the modelling dataset into training and testing sets is to assign 2/3 data points to the former and the remaining one-third to the latter. Therefore, we train the model using the training set and then apply the model to the test set. In this way, we can evaluate the performance of our model. For instance, if the training accuracy is extremely high while the testing accuracy is poor then this is a good indicator that the model is probably over fitted.

Note that splitting the dataset into training and testing sets is not the only action that could be required in order to avoid phenomenon’s such as overfitting. For instance, if both the training and testing sets contain patterns that do not exist in real world data then the model would still have poor performance even though we wouldn’t be able to observe it from the performance evaluation.

On a second note, you should be aware that there are certain situations you should consider creating an extra set called the **validation** set. The validation set is usually required when apart from model performance we also need to choose among many models and evaluate which model performs better.

**How to split our dataset into train and test sets?**

In this section, we are going to explore three different ways one can use to create training and testing sets. Before jumping into these approaches, let’s create a dummy dataset that will use for demonstration purposes. In the examples below, we will assume that we have a dataset stored in memory as a pandas Data Frame. The iris dataset contains 150 data points, each of which has four features.

1. import pandas as pd
2. from sklearn. datasets import load iris
3. iris\_data = load iris ()
4. df = pd. Data Frame (iris\_data. Data, columns=iris\_data. feature names)
5. print(df)

# sepal length (cm) sepal width (cm) petal length (cm) petal width (cm)

# 0 5.1 3.5 1.4 0.2

# 1 4.9 3.0 1.4 0.2

# 2 4.7 3.2 1.3 0.2

# 3 4.6 3.1 1.5 0.2

# 4 5.0 3.6 1.4 0.2

# .. ... ... ... ...

# 145 6.7 3.0 5.2 2.3

# 146 6.3 2.5 5.0 1.9

# 147 6.5 3.0 5.2 2.0

# 148 6.2 3.4 5.4 2.3

# 149 5.9 3.0 5.1 1.8

#

# [150 rows x 4 columns]

In the examples below, we will assume that we need a 80:20 ratio for training: testing sets.

**Using pandas**

The first option is to use pandas Data Frames’ method [sample()](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.sample.html):

*Return a random sample of items from an axis of object.*

*You can use random state for reproducibility*

We initially create the training set by taking a sample with a fraction of 0.8 from the overall rows in the pandas Data Frame. Note that we also define random state which corresponds to the seed, so that results are reproducible. Subsequently, we create the testing set by simply dropping the corresponding indices from the original Data Frame which are now included in the training set.

As we can see, the training set contains 120 examples, which aligns with the fraction that we requested when sampling the original modelling Data Frame. The remaining 30 examples were packed into the testing set.

**Using scikit-learn**

The second option — and probably the most commonly used — is the use of sklearn ‘s method called [train\_test\_split()](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html):

*Split arrays or matrices into random train and test subsets*

We can create both the training and testing’s sets in a one-liner by passing to train\_test\_split () the modelling Data Frame along with the fraction of the examples that should be included in the testing set. As before, we also set a random state so that the results are reproducible, that is every time we run the code, the same instances will be included in the training and testing sets respectively. The method returns a tuple with two Data Frames containing the training and testing examples.

**Using numpy**

Finally, a less commonly used way of creating testing and training samples is with numpy ‘s method [randn()](https://numpy.org/doc/stable/reference/random/generated/numpy.random.randn.html):

*Return a sample (or samples) from the “standard normal” distribution.*

We first create mask which is a numpy array that contains Boolean values that were computed by comparing a random float numbers in the range between 0 and 1 with the fraction we want to keep for the training set. Subsequently, we create the training and testing samples by filtering the Data Frame accordingly. Note however that this approach will **approximately** give a 80:20 ration meaning that the number of examples included in training and testing samples won’t be necessarily as accurate as the two methods we discussed earlier in this article.

1. import numpy as np
2. mask = np. random. And(Len(df)) =< 0.8
3. training data = df[mask]
4. testing data = df[~mask]
5. print (f"No. of training examples: {training data. shape [0]}")
6. print (f"No. of testing examples: {testing data. shape [0]}")
7. # No. of training examples: 125
8. # No. of testing examples: 25

**What’s next?**

Now that you have created the training and testing sets out of your original modelling dataset, you might also need to undertake further pre-processing steps such as scaling or normalisation. You must be careful when doing so, since you need to avoid introducing future information into your training set. This means that certain actions need to be applied first over the training set and then use the learned parameters from that step in order to apply them on testing set as well. For a more comprehensive explanation on this topic, you can read the article below.

**[Feature Scaling and Normalisation in a nutshell](https://towardsdatascience.com/feature-scaling-and-normalisation-in-a-nutshell-5319af86f89b" \t "_blank)**

[Why, How and When to re-scale your features](https://towardsdatascience.com/feature-scaling-and-normalisation-in-a-nutshell-5319af86f89b" \t "_blank)

[towardsdatascience.com](https://towardsdatascience.com/feature-scaling-and-normalisation-in-a-nutshell-5319af86f89b" \t "_blank)

**Conclusion**

In this article, we explored the importance of splitting our initial modelling dataset into training and testing samples. Furthermore, we discussed how these sets can help us identify whether our model was over fitted or under fitted. Finally, we’ve seen in action how to do this split with Python and pandas in three different ways; using pandas. Sample (), sklearn. traing\_test\_split () and numpy. Randn ().