Best Practice for working with train-test splits

Don't worry if you don't know yet what train-test splits are. However, it is a fundamental concept in the field of ML.

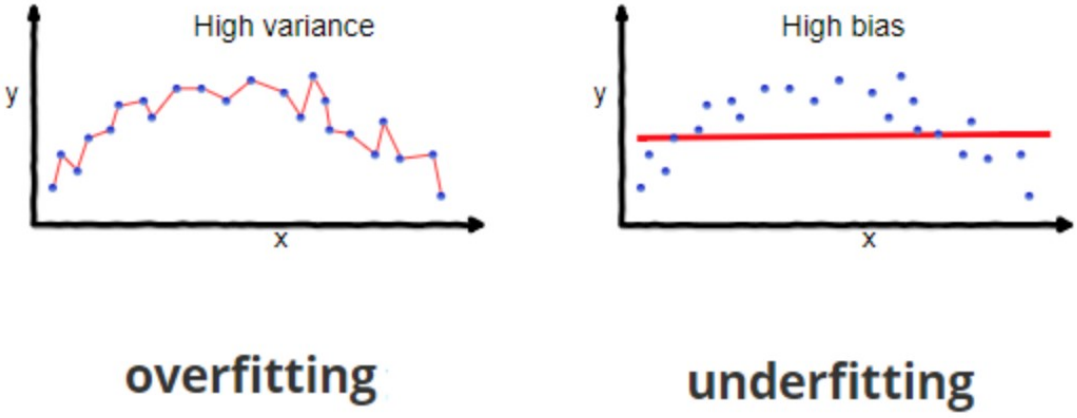
What is a train-test split and why do we need it?

To get into what a train-test split is, we need to first understand what is a 'test group' and why the hell do we need one when doing ML modeling.

In short, when we're trying to predict any kind of output, we will be 'training' a machine learning model using a dataset. This model will try to learn as much from the data as possible in order to make accurate predictions. As much, that perhaps our model will learn too much from it, up to a point where it will be only useful to predict that bunch of data we gave him to work with, but failing to make predictions with any other.

This potential problem or risk is the kick-starter point to several tools and concepts we usually apply in machine learning:

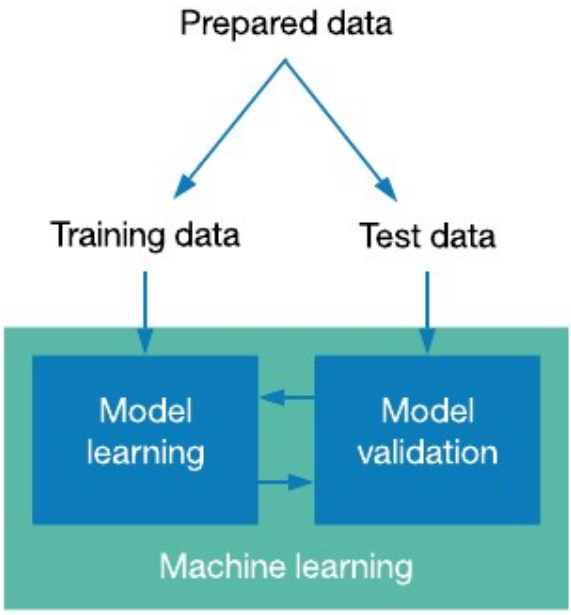
- The bias-variance trade-off, that, in a nutshell, it's just about how much our model should learn from our training data. If it learns too much, we'd say it has '**High Variance**' and it would be '**Over fitting**' our data. On the contrary, if it learns too little, it would have '**High Bias**' and the model would be '**Under fitting**' our training data.



- The concept of regularization also comes in handy when talking about all this; since it's a technique that allows us to control how much do we allow our model to learn from our data. The concept is not that simple but can be easily applied with Python.

- Apart from any kind of technique to control how much is our model learning from the data, a well-established practice is to split our data to evaluate our model, so we can be sure it performs well on different elements. Here's where the concept of test split finally appears! The idea is that we'll like to divide our original data into two groups: the training group will be used, not surprisingly, to train our model. While we'll leave apart a bunch of data, so once we have already trained our model and we are happy with its performance, we can evaluate it with a completely new bunch of data, to check if the model is consistent. This will be our test group. And if we obtain a much worse score on our test set, than in our train

group, then probably we're over fitting our training data. Usually **80–20** or **70–30** % train-test split is considered reasonable.



- Finally, we could also talk about the concept of doing cross-validation when evaluating the performance of our model.

How do we do a proper train-test split with Python?

As usually, Sklearn makes it all so easy for us, and it has a beautiful life-saving library, that comes really in handy to perform a train-test split:

from sklearn.model_selection import train_test_split

Let's go over a simple example anyway:

X_train, X_test, y_train, y_test = train_test_split(your_data, y, test_size=0.2, random_state=123)

There's really not much about how to implement this library and now you'll have 4 different bunches of data:

X_train: this will be your training group

X_test: this will be your test group

Y_train: this will be your target for your training group

Y_test: as you can imagine, this will be your target for your test group

6 common mistakes

1. **Write in disorder your train-test split code**

Yes, as silly as it sounds, it has the potential to be a huge headache. Picture this, it's early in the morning and you have been all night working on a dataset. Cleaning, merging data, doing some

feature engineering…the usual. It's time for you to make the train-test split so you can try a simple model before going to bed. You write your code, but instead of writing:

X_train, X_test, y_train, y_test

You write, for example:

X_test, X_train, y_test, y_train

One of the most important rules in Data Science is that you shouldn't reveal your test score until you're satisfied with your training/cross-validated score. But remember, you wrote in disorder your train-test split code, so the score you're getting is not your training, but instead it's your test score. You may spend hours trying to understand why you're getting such low values on your training data. Or even worse, you may discard your project because of having such a bad performance. In any case, it's a mistake that once it's done, it's hard to find and it may lead to hours of deep diving into your code trying to solve this silly mistake.

2. Mistype the size of the test group

One of the parameters you should specify is either 'train_size' or 'test_size'. You should use only one of them, but even more important, be sure not to confuse them. Otherwise, you could be setting a train set with only 20–30%. This could lead to several problems. From not having enough data to train a proper model, to obtaining too good, or too bad results that may lead you into some time-consuming further analysis.

3. Normalize your test group apart from your train data

Normalization is the process of adjusting values measured on different scales to a common scale. Suppose you're trying to predict whether a person is male or female, given a set of features such as height, weight and heart rate. In this case, all features are in different scales. For example, height could be in centimeters, while weight may be in kilos. In cases like this, is highly recommended to normalize the data to express it all in a common scale.

Sklearn offers a very friendly library to do it calling:

from sklearn.preprocessing import StandardScaler

The process of normalizing takes the Mean and Standard Deviation of each feature and adjusted their scale in order to be in between -1 and 1 with a Mean of 0. Once we imported the library, we can create an object StandardScaler, to proceed with the normalization:

scaler = StandardScaler()

However, if we are splitting our data into train and test groups, we should fit our StandardScaler object first using our train group and then transform our test group using that same object. For example:

scaler.fit(X_train)

X_train = scaler.transform(X_train)

X_test = scaler.transform(X_test)

Why do we have to normalize data this way? Remember that we'll use our data to train our model, so we'll want our StandardScaler object to register and proceed with the Mean and Standard

Deviation of our train set and transform our test group using it. Otherwise, we would be doing two different transformations, taking two different Means and two different Standard Deviations. Treating as different data that's supposed to be the same.

4. **Not shuffle your data when needed or vice-versa**

Another parameter from our Sklearn train_test_split is 'shuffle'. Let's keep the previous example and let's suppose that our dataset is composed of 1000 elements, of which the first 500 correspond to males, and the last 500 correspond to females. The default value for this parameter is 'True', but if by mistake or ignorance we set it to 'False' and we split our data 80–20, we'll end up training our model with a dataset with 500 males and 300 females, and testing it with a dataset only containing 200 females within it.

Take into account that the default value is 'True', so if it comes a time when you don't want to shuffle your data, don't forget to specify it.

5. **Not wisely use the 'stratify' parameter**

The 'stratify' parameter comes into handy so that the proportion of values in the sample produced in our test group will be the same as the proportion of values provided to parameter stratify. These results especially useful when working around classification problems, since if we don't provide this parameter with an array-like object, we may end with a non-representative distribution of our target classes in our test group.

Usually, this parameter is used by passing the target variable like this:

X_train, X_test, y_train, y_test = train_test_split(your_data, y, test_size=0.2, stratify=y, random_state=123, shuffle=True)

6. **Forget of setting the'random_state' parameter**

If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.

In other words: if you don't specify a number or RandomState object instance, each iteration of the train_test_split will give you different groups, since the seed used to proceed with the randomness around the split would be different. This may lead to confusion if for any reason we have to run our code, and we start obtaining different results.