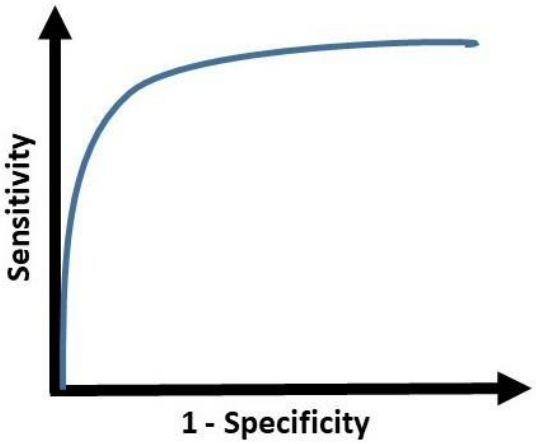
AUC _ROC CURVE

All set? Let's explore it! :D

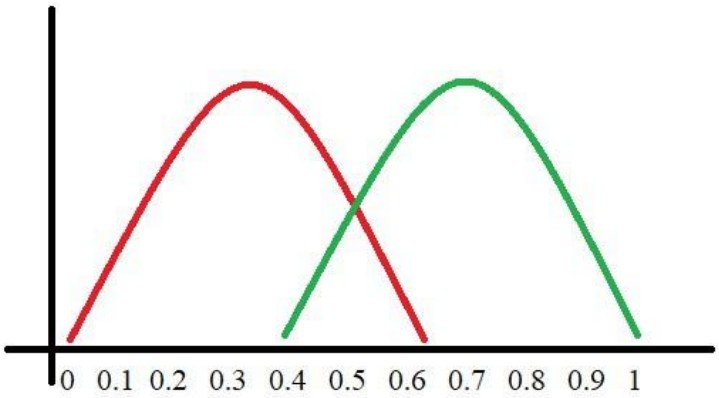


AUC ROC is one of the most important evaluation metrics for any classification model's performance.

What is ROC?

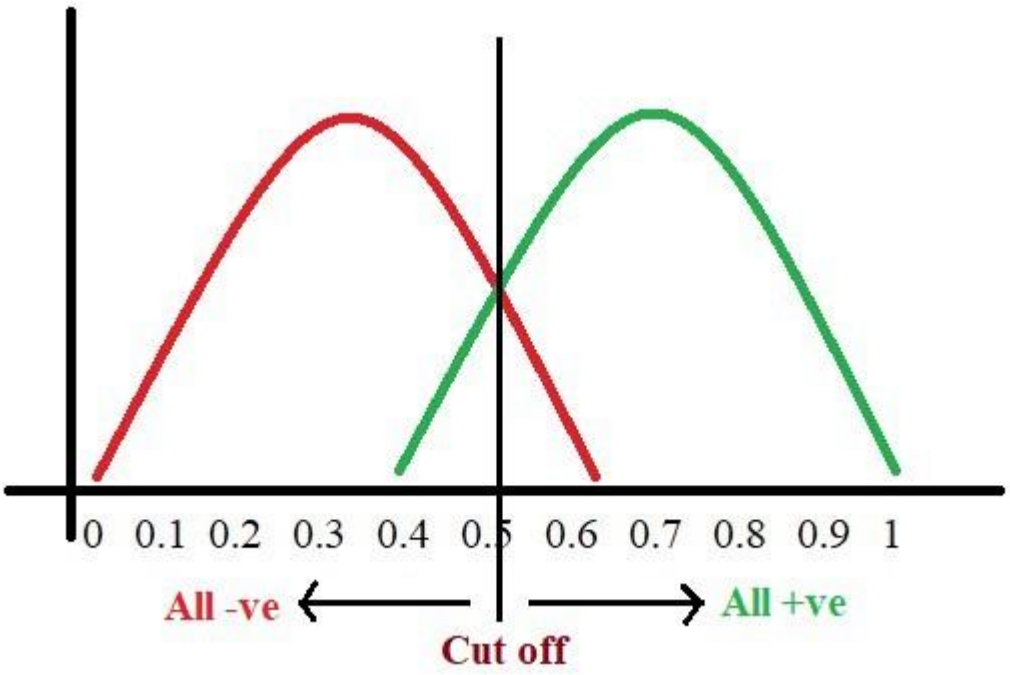
ROC (*Receiver Operating Characteristic*) Curve tells us about how good the model can distinguish between two things (*e.g If a patient has a disease or no*). Better models can accurately distinguish between the two. Whereas, a poor model will have difficulties in distinguishing between the two.

Let's assume we have a model which predicts whether the patient has a particular disease or no. The model predicts probabilities for each patient (*in python we use the" **predict_proba** " function*). Using these probabilities, we plot the distribution as shown below:



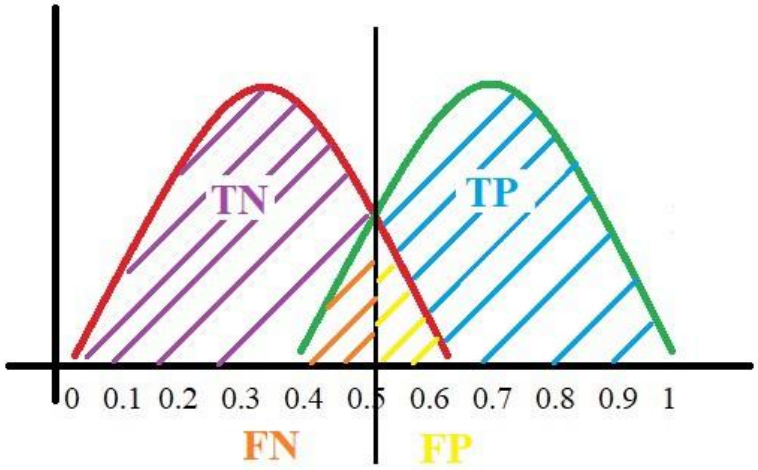
Here, the red distribution represents all the patients who do not have the disease and the green distribution represents all the patients who have the disease.

Now we got to pick a value where we need to set the cut off i.e. a threshold value, above which we will predict everyone as positive (*they have the disease*) and below which will predict as negative (*they do not have the disease*). We will set the threshold at "**0.5**" as shown below:



All the positive values above the threshold will be "**True Positives**" and the negative values above the threshold will be "**False Positives**" as they are predicted incorrectly as positives.

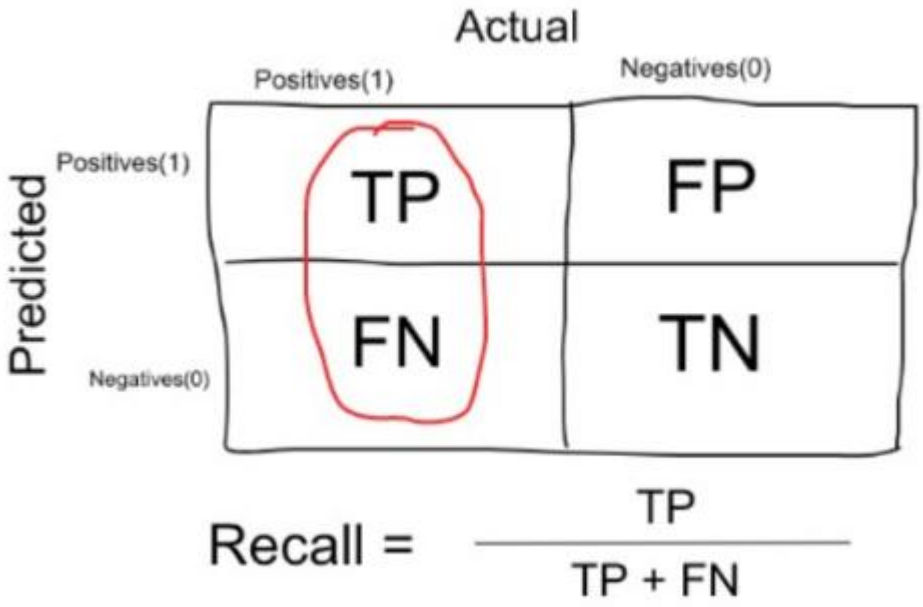
All the negative values below the threshold will be "**True Negatives**" and the positive values below the threshold will be "**False Negative**" as they are predicted incorrectly as negatives.



Here, we have got a basic idea of the model predicting correct and incorrect values with respect to the threshold set. Before we move on, let's go through two important terms: **Sensitivity and Specificity.**

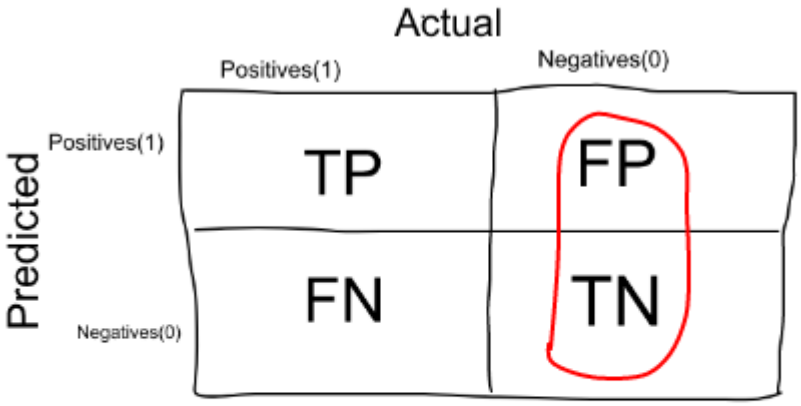
What is Sensitivity and Specificity?

In simple terms, the proportion of patients that were identified correctly to have the disease (*i.e. True Positive*) upon the total number of patients who actually have the disease is called as Sensitivity or Recall.



$$\text{Recall} = \frac{TP}{TP + FN}$$

Similarly, the proportion of patients that were identified correctly to not have the disease (*i.e. True Negative*) upon the total number of patients who do not have the disease is called as Specificity.



$$\text{Specificity} = \frac{TN}{TN + FP}$$

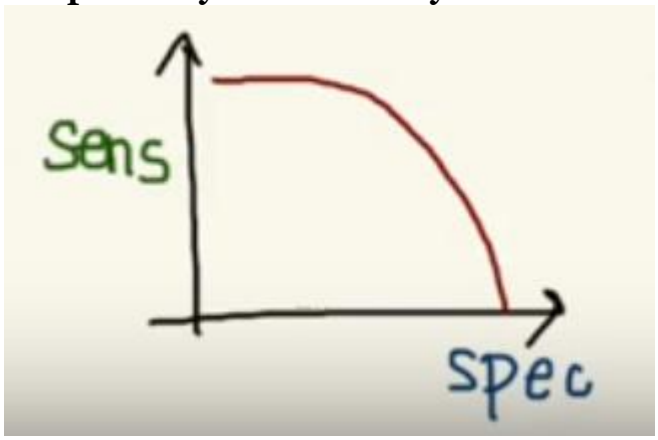
Trade-off between Sensitivity and Specificity

When we decrease the threshold, we get more positive values thus increasing the sensitivity. Meanwhile, this will decrease the specificity.

Similarly, when we increase the threshold, we get more negative values thus increasing the specificity and decreasing sensitivity.

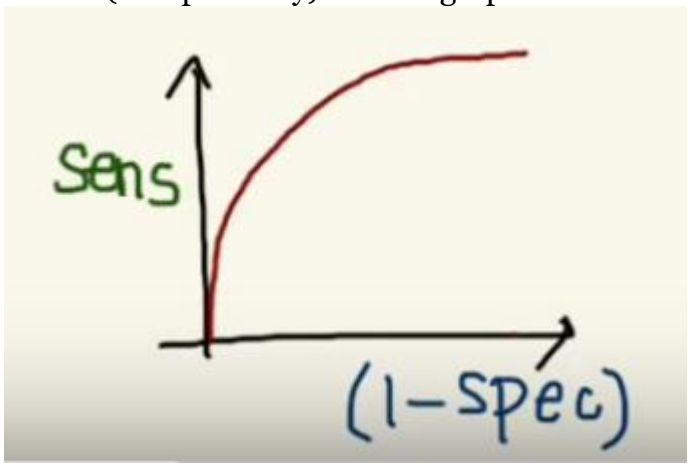
As **Sensitivity ↓ Specificity ↑**

As **Specificity ↓ Sensitivity ↑**

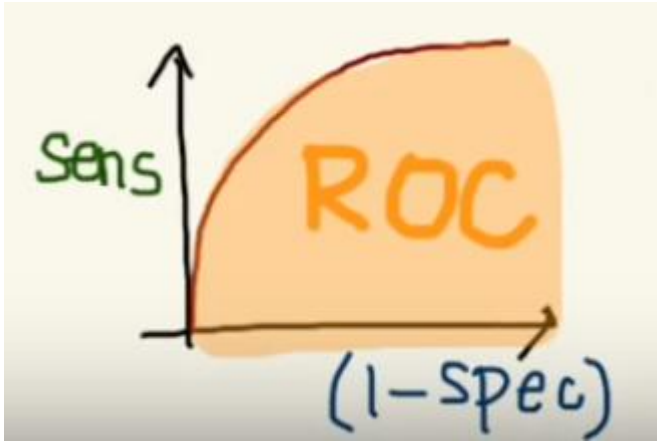


Trade off between Sensitivity & Specificity

But, this is not how we graph the ROC curve. To plot ROC curve, instead of Specificity we use (1—Specificity) and the graph will look something like this:



So now, when the sensitivity increases, (1—specificity) will also increase. This curve is known as the ROC curve.

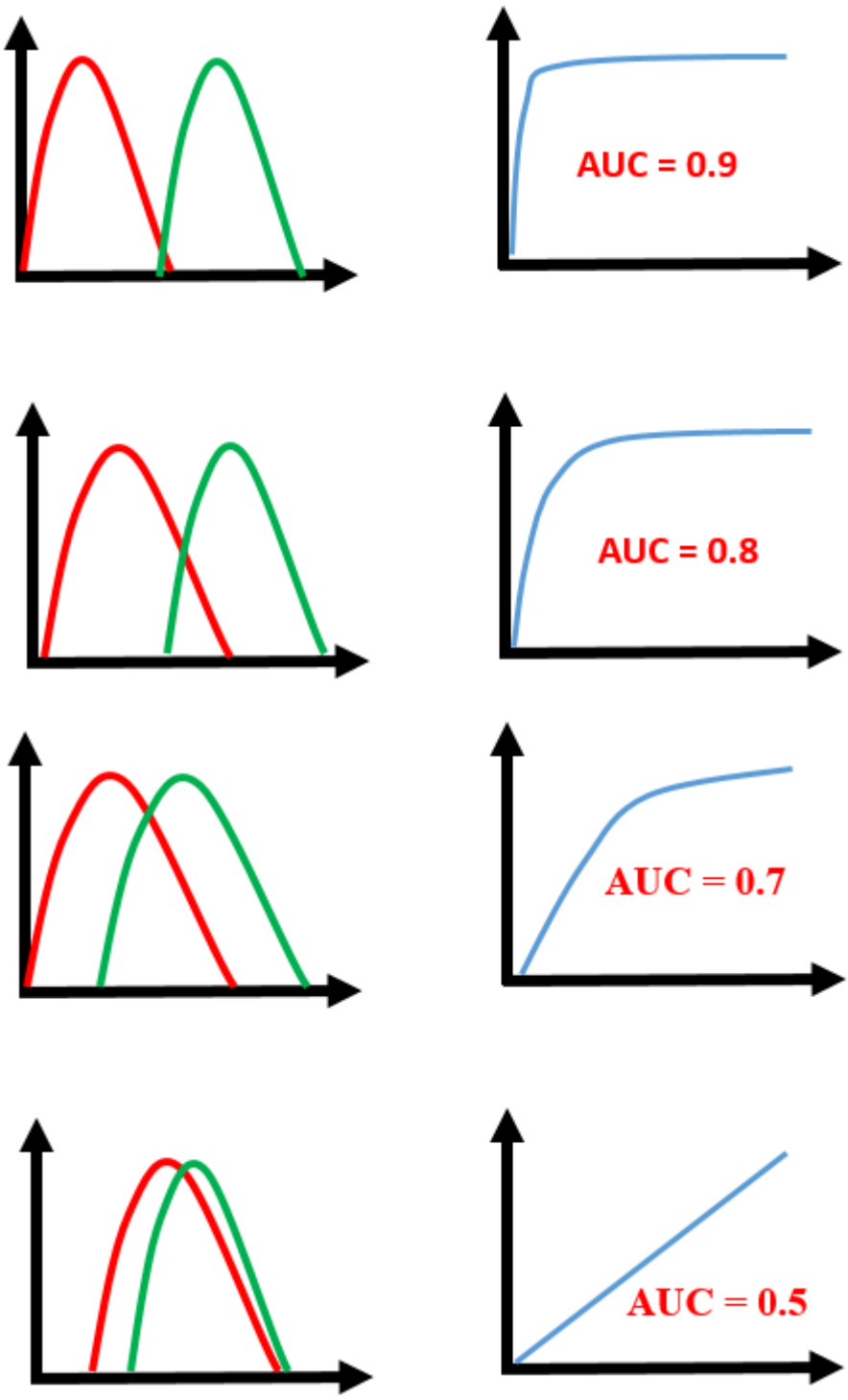


I know you might be wondering why do we use (1—specificity)? Don't worry I'll come back to it soon. :)

Area Under the Curve

The AUC is the area under the ROC curve. This score gives us a good idea of how well the model performances.

Let's take a few examples



As we see, the first model does quite a good job of distinguishing the positive and the negative values. Therefore, there the AUC score is 0.9 as the area under the ROC curve is large.

Whereas, if we see the last model, predictions are completely overlapping each other and we get the AUC score of 0.5. This means that the model is performing poorly and it is predictions are almost random.

Why do we use (1—Specificity)?

Let's derive what exactly is (1—Specificity):

$$Specificity = \frac{TN}{TN + FP}$$

$$1 - Specificity = 1 - \frac{TN}{TN + FP}$$

$$1 - Specificity = \frac{TN + FP - TN}{TN + FP}$$

$$1 - Specificity = \frac{FP}{TN + FP}$$

As we see above, Specificity gives us the True Negative Rate and (1—Specificity) gives us the False Positive Rate.

So the sensitivity can be called as the "***True Positive Rate***" and (1—Specificity) can be called as the "***False Positive Rate***".

So now we are just looking at the positives. As we increase the threshold, we decrease the *TPR* as well as the *FPR* and when we decrease the threshold, we are increasing the *TPR* and *FPR*.

Thus, AUC ROC indicates how well the probabilities from the positive classes are separated from the negative classes.

https://www.youtube.com/watch?v=mUMd_cKU0VM

Pipeline

Pipelines for Automating Machine Learning Workflows

There are standard workflows in applied machine learning. Standard because they overcome common problems like data leakage in your test harness.

Python scikit-learn provides a Pipeline utility to help automate machine learning workflows.

Pipelines work by allowing for a linear sequence of data transforms to be chained together culminating in a modeling process that can be evaluated.

A pipeline is what chains several steps together, once the initial exploration is done. For example, some codes are meant to transform features—normalise numericals, or turn text into vectors, or fill up missing data, they are transformers; other codes are meant to predict variables by fitting an algorithm, such as random forest or support vector machine, they are estimators. Pipeline chains all these together which can then be applied to training data en bloc.

The goal is to ensure that all of the steps in the pipeline are constrained to the data available for the evaluation, such as the training dataset or each fold of the cross validation procedure.

Example of a pipeline that imputes data with most frequent value of each column, and then fit to a decision tree classifier.

```
fromsklearn.pipeline import Pipeline
```

```
steps = [('imputation', Imputer(missing_values='NaN', strategy
= 'most_frequent', axis=0)),       ('clf',
DecisionTreeClassifier())]
```

```
pipeline = Pipeline(steps)
clf = pipeline.fit(X_train,y_train)
```

Instead of fitting to one model, it can be looped over several models to find the best one.

```
classifiers = [



KNeighborsClassifier(5),



RandomForestClassifier(),



GradientBoostingClassifier()]

forclf in classifiers:

steps = [('imputation', Imputer(missing_values='NaN', strategy
= 'most_frequent', axis=0)),

('clf', clf)]

pipeline = Pipeline(steps)
```

I also learnt the pipeline itself can be used as an estimator and passed to cross validation or gridsearch.

```
fromsklearn.model_selection import KFold


fromsklearn.model_selection import cross_val_score


kfold = KFold(n_splits=10, random_state=seed)


results = cross_val_score(pipeline, X_train, y_train,
cv=kfold)


print(results.mean())
```

Cross Validation in Machine Learning

What is Overfitting/Underfitting a Model?

As mentioned, in statistics and machine learning we usually split our data into two subsets: training data and testing data (and sometimes to three: train, validate and test), and fit our model on the train data, in order to make predictions on the test data. When we do that, one of two thing might happen: we overfit our model or we underfit our model. We don't want any of these things to happen, because they affect the predictability of our model—we might be using a model that has lower accuracy and/or is ungeneralized (meaning you can't generalize your predictions on other data). Let's see what under and overfitting actually mean:

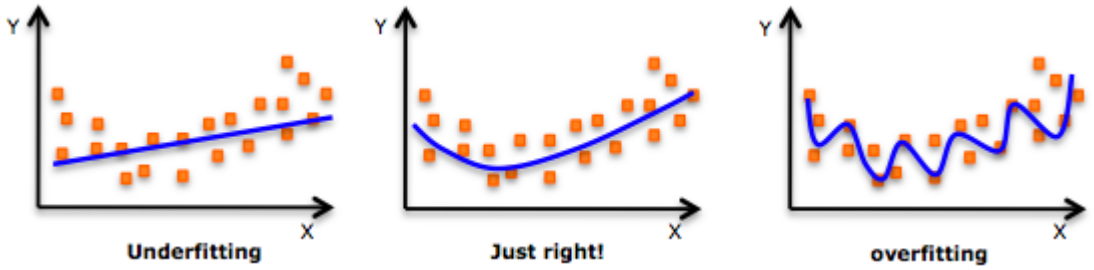
Overfitting

Overfitting means that model we trained has trained "too well" and is now, well, fit too closely to the training dataset. This usually happens when the model is too complex (i.e. too many features/variables compared to the number of

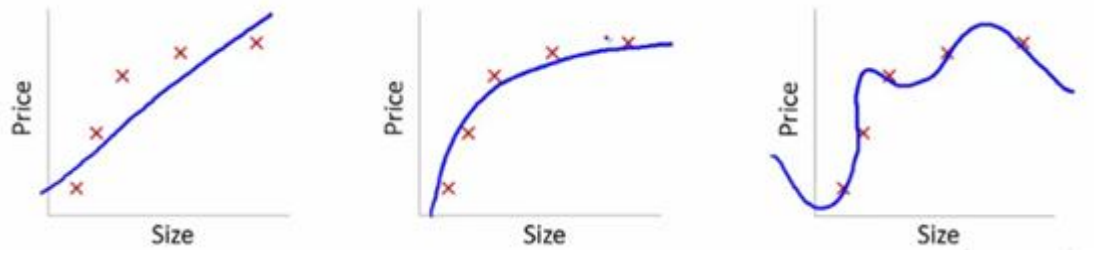
observations). This model will be very accurate on the training data but will probably be very not accurate on untrained or new data. It is because this model is not generalized (or not AS generalized), meaning you can generalize the results and can't make any inferences on other data, which is, ultimately, what you are trying to do. Basically, when this happens, the model learns or describes the "noise" in the training data instead of the actual relationships between variables in the data. This noise, obviously, isn't part in of any new dataset, and cannot be applied to it.

Underfitting

In contrast to overfitting, when a model is underfitted, it means that the model does not fit the training data and therefore misses the trends in the data. It also means the model cannot be generalized to new data. As you probably guessed (or figured out!), this is usually the result of a very simple model (not enough predictors/independent variables). It could also happen when, for example, we fit a linear model (like linear regression) to data that is not linear.It almost goes without saying that this model will have poor predictive ability (on training data and can't be generalized to other data).



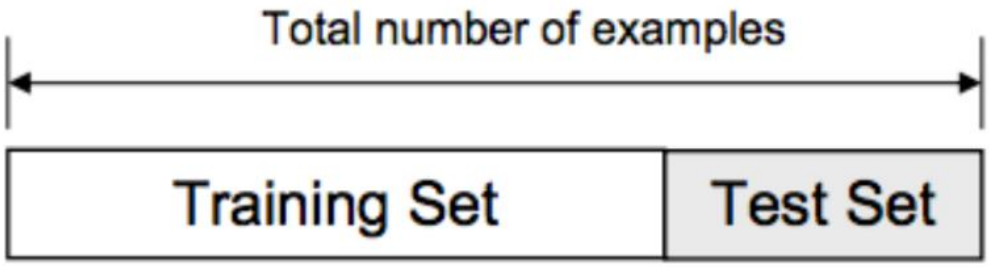
An example of overfitting, underfitting and a model that's "just right!"



It is worth noting the underfitting is not as prevalent as overfitting. Nevertheless, we want to avoid both of those problems in data analysis. You might say we are trying to find the middle ground between under and overfitting our model. As you will see, train/test split and cross validation help to avoid overfitting more than underfitting. Let's dive into both of them!

Train/Test Split

As I said before, the data we use is usually split into training data and test data. The training set contains a known output and the model learns on this data in order to be generalized to other data later on. We have the test dataset (or subset) in order to test our model's prediction on this subset.



Train/Test Split

A common practice in data science competitions is to iterate over various models to find a better performing model. However, it becomes difficult to distinguish whether this improvement in score is coming because we are capturing the relationship better, or we are just over-fitting the data. To find the right answer for this question, we use validation techniques. This method helps us in achieving more generalized relationships.

What is Cross Validation?

Cross Validation is a technique which involves reserving a particular sample of a dataset on which you do not train the model.Later, you test your model on this sample before finalizing it.

Here are the steps involved in cross validation:

1. You *reserve* a sample data set
2. Train the model using the remaining part of the dataset
3. Use the reserve sample of the test (validation) set. This will help you in gauging the effectiveness of your model's performance. If your model delivers a positive result on validation data, go ahead with the current model. It rocks!

k-fold cross validation

K-Fold Cross Validation randomly splits the training data into **K subsets called folds**.

From the above two validation methods, we've learnt:

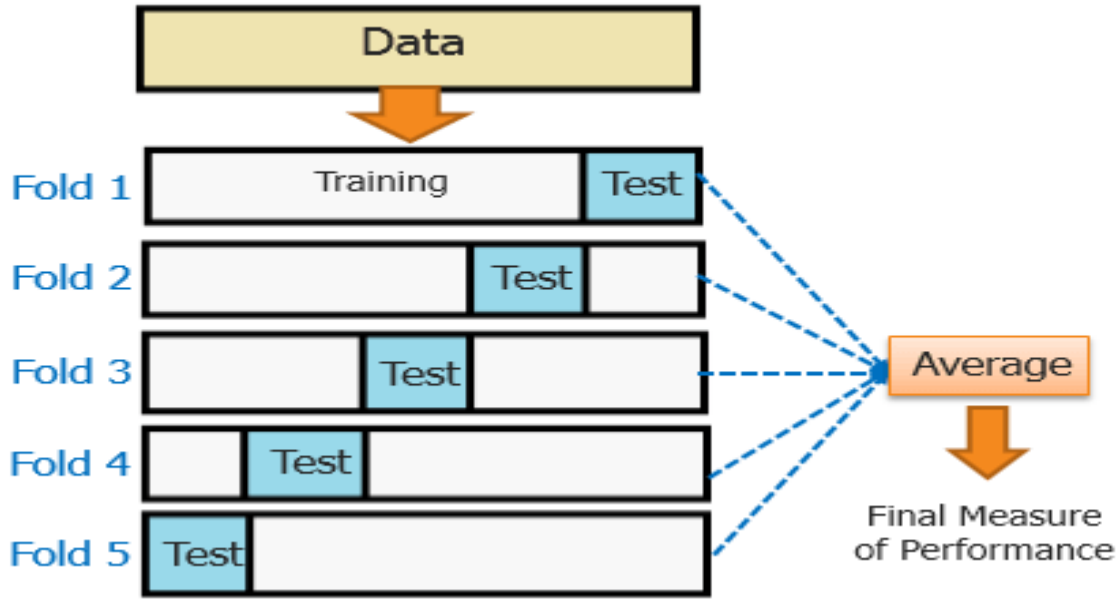
1. We should train the model on a large portion of the dataset. Otherwise we'll fail to read and recognise the underlying trend in the data. This will eventually result in a higher bias
2. We also need a good ratio of testing data points. As we have seen above, less amount of data points can lead to a variance error while testing the effectiveness of the model
3. We should iterate on the training and testing process multiple times. We should change the train and test dataset distribution. This helps in validating the model effectiveness properly

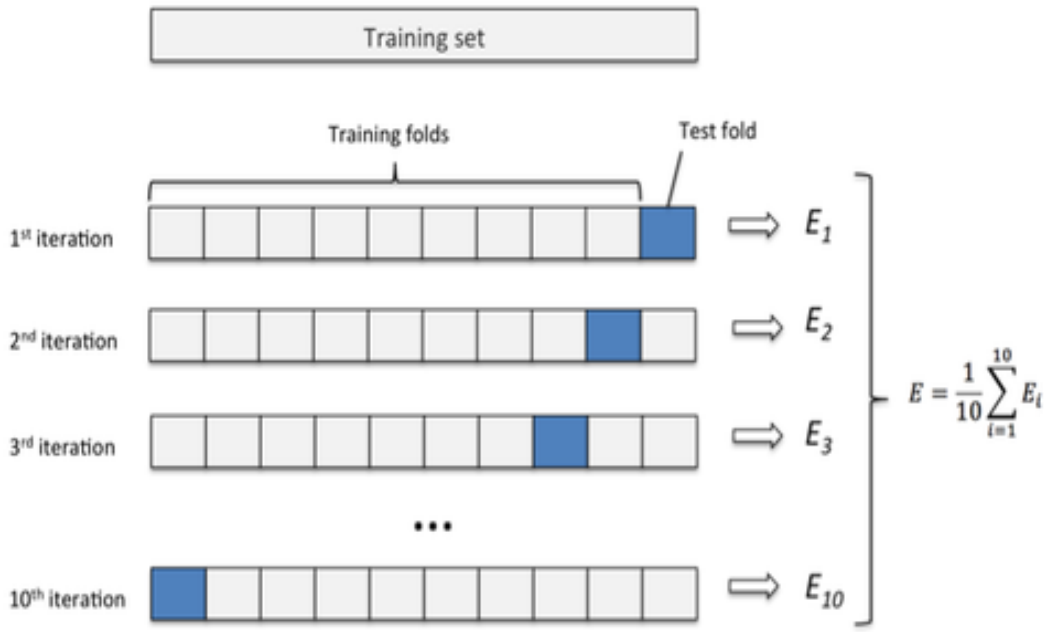
Do we have a method which takes care of all these 3 requirements?

Yes! That method is known as "k-fold cross validation". It's easy to follow and implement. Below are the steps for it:

1. Randomly split your entire dataset into k"folds"
2. For each k-fold in your dataset, build your model on k – 1 folds of the dataset. Then, test the model to check the effectiveness for *kth* fold
3. Record the error you see on each of the predictions
4. Repeat this until each of the k-folds has served as the test set
5. The average of your k recorded errors is called the cross-validation error and will serve as your performance metric for the model

Below is the visualization of a k-fold validation when k=5.





Cross-Validation API

We do not have to implement k-fold cross-validation manually. The scikit-learn library provides an implementation that will split a given data sample up.

The *KFold()* scikit-learn class can be used. It takes as arguments the number of splits, whether or not to shuffle the sample, and the seed for the pseudorandom number generator used prior to the shuffle.
For example, we can create an instance that splits a dataset into 3 folds, shuffles prior to the split, and uses a value of 1 for the pseudorandom number generator;

```
Kfold(n_splits=5,shuffle=False,seed=2)
```

```
1 kfold=KFold(3,True,1)
```

The *split()* function can then be called on the class where the data sample is provided as an argument. Called repeatedly, the split will return each group of train and test sets. Specifically, arrays are returned containing the indexes into the original data sample of observations to use for train and test sets on each iteration.

For example, we can enumerate the splits of the indices for a data sample using the created *KFold* instance as follows:

```
# enumerate splits
fortrain,test inkfold.split(data):
        print('train: %s, test: %s'%(train,test))
```

We can tie all of this together with our small dataset used in the worked example of the prior section

```
# scikit-learn k-fold cross-validation
from numpy import array
from sklearn.model_selection import KFold
# data sample
data=array([0.1,0.2,0.3,0.4,0.5,0.6])
# prepare cross validation
kfold=KFold(3,True,1)
# enumerate splits
fortrain,test inkfold.split(data):
        print('train: %s, test: %s'%(data[train],data[test]))
```

Running the example prints the specific observations chosen for each train and test set. The indices are used directly on the original data array to retrieve the observation values

```
train: [0.1 0.4 0.5 0.6], test: [0.2 0.3]
train: [0.2 0.3 0.4 0.6], test: [0.1 0.5]
train: [0.1 0.2 0.3 0.5], test: [0.4 0.6]
```

Usefully, the k-fold cross validation implementation in scikit-learn is provided as a component operation within broader methods, such as grid-searching model hyperparameters and scoring a model on a dataset.

Nevertheless, the *KFold* class can be used directly in order to split up a dataset prior to modeling such that all models will use the same data splits. This is especially helpful if you are working with very large data samples. The use of the same splits across algorithms can have benefits for statistical tests that you may wish to perform on the data later.

Variations on Cross-Validation

There are a number of variations on the k-fold cross validation procedure.

Three commonly used variations are as follows:

- **Train/Test Split**: Taken to one extreme, k may be set to 1 such that a single train/test split is created to evaluate the model.
- **LOOCV**: Taken to another extreme, k may be set to the total number of observations in the dataset such that each observation is given a chance to be the held out of the dataset. This is called leave-one-out cross-validation, or LOOCV for short.
- **Stratified**: The splitting of data into folds may be governed by criteria such as ensuring that each fold has the same proportion of observations with a given categorical value, such as the class outcome value. This is called stratified cross-validation.
- **Repeated**: This is where the k-fold cross-validation procedure is repeated n times, where importantly, the data sample is shuffled prior to each repetition, which results in a different split of the sample.

```
fromsklearn.model_selectionimportcross_val_score


dtc = DecisionTreeClassifier()
scores = cross_val_score(dtc, X_train, Y_train, cv=5,
scoring = "accuracy")
print("Scores:", scores)
print("Mean:", scores.mean())
print("Standard Deviation:", scores.std())
```