**What is a Confusion Matrix?**

A Confusion matrix is an N x N matrix used for evaluating the performance of a classification model, where N is the number of target classes. The matrix compares the actual target values with those predicted by the machine learning model. This gives us a holistic view of how well our classification model is performing and what kinds of errors it is making.

For a binary classification problem, we would have a 2 x 2 matrix as shown below with 4 values:

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/04/Basic-Confusion-matrix.png)

Let’s decipher the matrix:

* The target variable has two values: **Positive**or **Negative**
* The **columns**represent the **actual values** of the target variable
* The **rows**represent the **predicted values**of the target variable

**Understanding True Positive, True Negative, False Positive and False Negative in a Confusion Matrix**

**True Positive (TP)**

* The predicted value matches the actual value
* The actual value was positive and the model predicted a positive value

**True Negative (TN)**

* The predicted value matches the actual value
* The actual value was negative and the model predicted a negative value

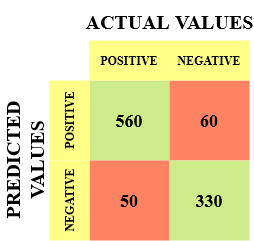
**False Positive (FP) – Type 1 error**

* The predicted value was falsely predicted
* The actual value was negative but the model predicted a positive value

**False Negative (FN) – Type 2 error**

* The predicted value was falsely predicted
* The actual value was positive but the model predicted a negative value

Let me give you an example to better understand this. Suppose we had a classification dataset with 1000 data points. We fit a classifier on it and get the below confusion matrix:

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/04/Confusionmatrix-example.png)

The different values of the Confusion matrix would be as follows:

* True Positive (TP) = 560; meaning 560 positive class data points were correctly classified by the model
* True Negative (TN) = 330; meaning 330 negative class data points were correctly classified by the model
* False Positive (FP) = 60; meaning 60 negative class data points were incorrectly classified as belonging to the positive class by the model
* False Negative (FN) = 50; meaning 50 positive class data points were incorrectly classified as belonging to the negative class by the model

**Why Do We Need a Confusion Matrix?**

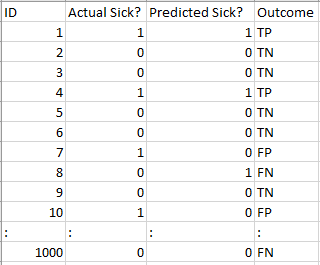
Before we answer this question, let’s think about a hypothetical classification problem.

Let’s say you want to predict how many people are infected with a contagious virus in times before they show the symptoms, and isolate them from the healthy population The two values for our target variable would be: Sick and Not Sick.

Now, you must be wondering – why do we need a confusion matrix when we have our all-weather friend – Accuracy? Well, let’s see where accuracy falters.

Our dataset is an example of an [**imbalanced dataset**](https://www.analyticsvidhya.com/blog/2017/03/imbalanced-data-classification/?utm_source=blog&utm_medium=confusion-matrix-machine-learning). There are 947 data points for the negative class and 3 data points for the positive class. This is how we’ll calculate the accuracy:[Equation_Accuracy](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/04/Equation_Accuracy.png)

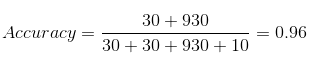
Let’s see how our model performed:

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/04/Dataset.png)

The total outcome values are:

TP = 30, TN = 930, FP = 30, FN = 10

So, the accuracy for our model turns out to be:

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/04/Confusion-matrix_Accuracy.png)

96%! Not bad!

But it is giving the wrong idea about the result. Think about it.

Our model is saying “I can predict sick people 96% of the time”. However, it is doing the opposite. It is predicting the people who will not get sick with 96% accuracy while the sick are spreading the virus!

Do you think this is a correct metric for our model given the seriousness of the issue? Shouldn’t we be measuring how many positive cases we can predict correctly to arrest the spread of the contagious virus? Or maybe, out of the correctly predicted cases, how many are positive cases to check the reliability of our model?

This is where we come across the dual concept of Precision and Recall.

**Precision vs. Recall**

*Precision tells us how many of the correctly predicted cases actually turned out to be positive.*

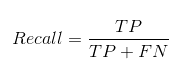
Here’s how to calculate Precision:

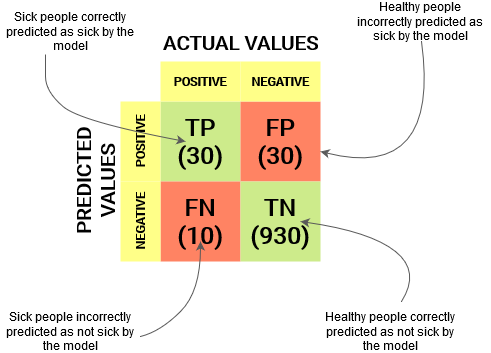
[Confusion Matrix Precision](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/04/Confusion-matrix_Precision.png)

This would determine whether our model is reliable or not.

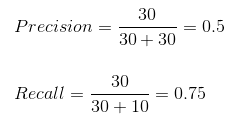
*Recall tells us how many of the actual positive cases we were able to predict correctly with our model.*

And here’s how we can calculate Recall:

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/04/Confusion-matrix_Recall.png)

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/04/Example-Confusion-matrix.png)

We can easily calculate Precision and Recall for our model by plugging in the values into the above questions:

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/04/Confusion_Matrix_Precision_Recall.png)

50% percent of the correctly predicted cases turned out to be positive cases. Whereas 75% of the positives were successfully predicted by our model. Awesome!

*Precision is a useful metric in cases where False Positive is a higher concern than False Negatives.*

Precision is important in music or video recommendation systems, e-commerce websites, etc. Wrong results could lead to customer churn and be harmful to the business.

*Recall is a useful metric in cases where False Negative trumps False Positive.*

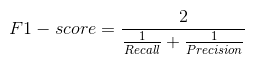
Recall is important in medical cases where it doesn’t matter whether we raise a false alarm but the actual positive cases should not go undetected!

In our example, Recall would be a better metric because we don’t want to accidentally discharge an infected person and let them mix with the healthy population thereby spreading the contagious virus. Now you can understand why accuracy was a bad metric for our model.

But there will be cases where there is no clear distinction between whether Precision is more important or Recall. What should we do in those cases? We combine them!

**F1-Score**

In practice, when we try to increase the precision of our model, the recall goes down, and vice-versa. The F1-score captures both the trends in a single value:

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/04/Confusion-Matrix-F1-score.png)

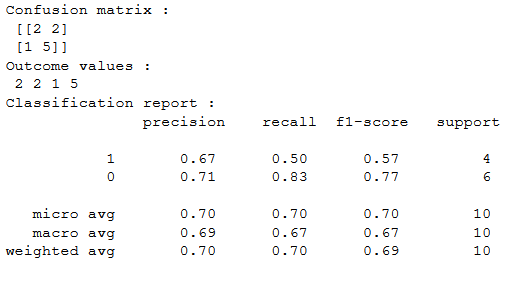
**F1-score is a harmonic mean of Precision and Recall**, and so it gives a combined idea about these two metrics. It is maximum when Precision is equal to Recall.

But there is a catch here. The interpretability of the F1-score is poor. This means that we don’t know what our classifier is maximizing – precision or recall? So, we use it in combination with other evaluation metrics which gives us a complete picture of the result.

**Confusion Matrix using scikit-learn in Python**

You know the theory – now let’s put it into practice. Let’s code a confusion matrix with the [Scikit-learn (sklearn) library](https://www.analyticsvidhya.com/blog/2020/02/everything-you-should-know-scikit-learn/?utm_source=blog&utm_medium=confusion-matrix-machine-learning" \t "_blank) in Python.

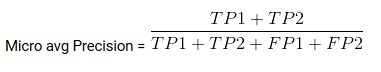
|  |  |
| --- | --- |
|  | # confusion matrix in sklearn |
|  | from sklearn.metrics import confusion\_matrix |
|  | from sklearn.metrics import classification\_report |
|  |  |
|  | # actual values |
|  | actual = [1,0,0,1,0,0,1,0,0,1] |
|  | # predicted values |
|  | predicted = [1,0,0,1,0,0,0,1,0,0] |
|  |  |
|  | # confusion matrix |
|  | matrix = confusion\_matrix(actual,predicted, labels=[1,0]) |
|  | print('Confusion matrix : \n',matrix) |
|  |  |
|  | # outcome values order in sklearn |
|  | tp, fn, fp, tn = confusion\_matrix(actual,predicted,labels=[1,0]).reshape(-1) |
|  | print('Outcome values : \n', tp, fn, fp, tn) |
|  |  |
|  | # classification report for precision, recall f1-score and accuracy |
|  | matrix = classification\_report(actual,predicted,labels=[1,0]) |
|  | print('Classification report : \n',matrix) |



Sklearn has two great functions: **confusion\_matrix()** and **classification\_report()**.

* Sklearn **[confusion\_matrix()](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html)** returns the values of the Confusion matrix. The output is, however, slightly different from what we have studied so far. It takes the rows as Actual values and the columns as Predicted values. The rest of the concept remains the same.
* Sklearn **[classification\_report()](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html)** outputs precision, recall and f1-score for each target class. In addition to this, it also has some extra values: **micro avg**, **macro avg**, and **weighted avg**

**Mirco average** is the precision/recall/f1-score calculated for all the classes.

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/04/Micro-avg-Precision.png)

**Macro average** is the average of precision/recall/f1-score.

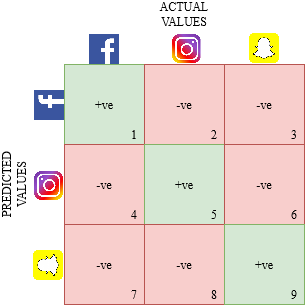
[Confusion matrix: Macro avg Precision](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/04/Macro-avg-Precision.png)

**Weighted average** is just the weighted average of precision/recall/f1-score.

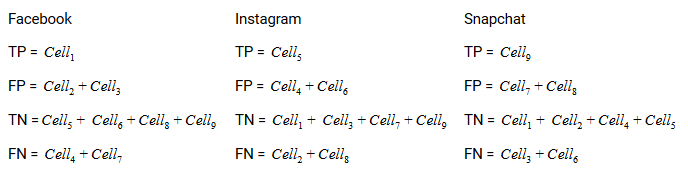
**Confusion Matrix for Multi-Class Classification**

How would a confusion matrix work for a multi-class classification problem? Well, don’t scratch your head! We will have a look at that here.

Let’s draw a confusion matrix for a multiclass problem where we have to predict whether a person loves Facebook, Instagram or Snapchat. The confusion matrix would be a 3 x 3 matrix like this:

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/04/Multiclass-confusion-matrix.png)

The true positive, true negative, false positive and false negative for each class would be calculated by adding the cell values as follows:

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/04/Multiclass-confusion-matrix-result.png)

## What are Sensitivity and Specificity?

This is what a confusion matrix looks like:



 Sensitivity / True Positive Rate / Recall

Sensitivity formula

Sensitivity tells us what proportion of the positive class got correctly classified.

A simple example would be to determine what proportion of the actual sick people were correctly detected by the model.

 False Negative Rate

False Negative Rate

False Negative Rate (FNR) tells us what proportion of the positive class got incorrectly classified by the classifier.

A higher TPR and a lower FNR is desirable since we want to correctly classify the positive class.

 Specificity / True Negative Rate

Specificity formula

Specificity tells us what proportion of the negative class got correctly classified.

Taking the same example as in Sensitivity, Specificity would mean determining the proportion of healthy people who were correctly identified by the model.

### False Positive Rate

False Positive Rate

FPR tells us what proportion of the negative class got incorrectly classified by the classifier.

A higher TNR and a lower FPR is desirable since we want to correctly classify the negative class.

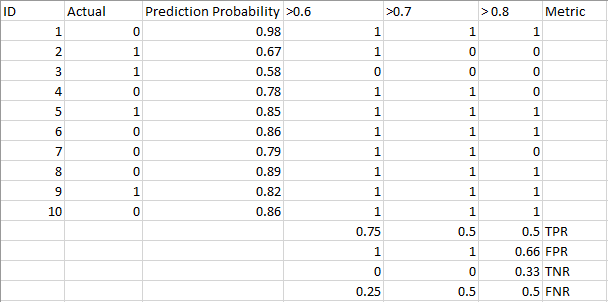
Out of these metrics, **Sensitivity**and **Specificity**are perhaps the most important and we will see later on how these are used to build an evaluation metric. But before that, let’s understand why the probability of prediction is better than predicting the target class directly.

## Probability of Predictions

A machine learning classification model can be used to predict the actual class of the data point directly or predict its probability of belonging to different classes. The latter gives us more control over the result. We can determine our own threshold to interpret the result of the classifier. This is sometimes more prudent than just building a completely new model!

Setting different thresholds for classifying positive class for data points will inadvertently change the Sensitivity and Specificity of the model. And one of these thresholds will probably give a better result than the others, depending on whether we are aiming to lower the number of False Negatives or False Positives.

Have a look at the table below:



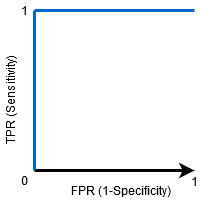
The metrics change with the changing threshold values. We can generate different confusion matrices and compare the various metrics that we discussed in the previous section. But that would not be a prudent thing to do. Instead, what we can do is generate a plot between some of these metrics so that we can easily visualize which threshold is giving us a better result.

The AUC-ROC curve solves just that problem!

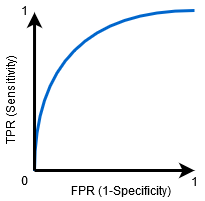
## What is the AUC-ROC curve?

The **Receiver Operator Characteristic (ROC)** curve is an evaluation metric for binary classification problems. It is a probability curve that plots the **TPR**against **FPR**at various threshold values and essentially **separates the ‘signal’ from the ‘noise’**. The **Area Under the Curve (AUC)**is the measure of the ability of a classifier to distinguish between classes and is used as a summary of the ROC curve.

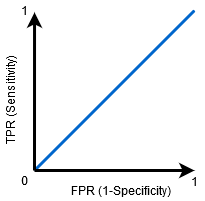
*The higher the AUC, the better the performance of the model at distinguishing between the positive and negative classes.*



When AUC = 1, then the classifier is able to perfectly distinguish between all the Positive and the Negative class points correctly. If, however, the AUC had been 0, then the classifier would be predicting all Negatives as Positives, and all Positives as Negatives.



When 0.5<AUC<1, there is a high chance that the classifier will be able to distinguish the positive class values from the negative class values. This is so because the classifier is able to detect more numbers of True positives and True negatives than False negatives and False positives.



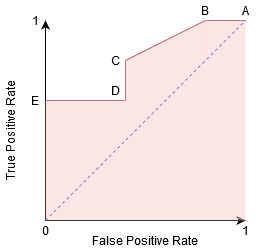
When AUC=0.5, then the classifier is not able to distinguish between Positive and Negative class points. Meaning either the classifier is predicting random class or constant class for all the data points.

So, the higher the AUC value for a classifier, the better its ability to distinguish between positive and negative classes.

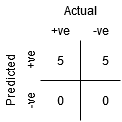
## How Does the AUC-ROC Curve Work?

In a ROC curve, a higher X-axis value indicates a higher number of False positives than True negatives. While a higher Y-axis value indicates a higher number of True positives than False negatives. So, the choice of the threshold depends on the ability to balance between False positives and False negatives.

Let’s dig a bit deeper and understand how our ROC curve would look like for different threshold values and how the specificity and sensitivity would vary.



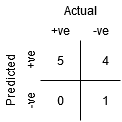
We can try and understand this graph by generating a confusion matrix for each point corresponding to a threshold and talk about the performance of our classifier:



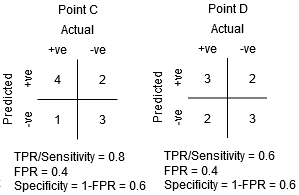
Point A is where the Sensitivity is the highest and Specificity the lowest. This means all the Positive class points are classified correctly and all the Negative class points are classified incorrectly.

*In fact, any point on the blue line corresponds to a situation where True Positive Rate is equal to False Positive Rate.*

*All points above this line correspond to the situation where the proportion of correctly classified points belonging to the Positive class is greater than the proportion of incorrectly classified points belonging to the Negative class.*

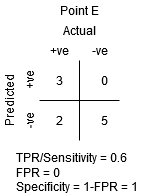


Although Point B has the same Sensitivity as Point A, it has a higher Specificity. Meaning the number of incorrectly Negative class points is lower compared to the previous threshold. This indicates that this threshold is better than the previous one.



Between points C and D, the Sensitivity at point C is higher than point D for the same Specificity. This means, for the same number of incorrectly classified Negative class points, the classifier predicted a higher number of Positive class points. Therefore, the threshold at point C is better than point D.

Now, depending on how many incorrectly classified points we want to tolerate for our classifier, we would choose between point B or C for predicting whether you can defeat me in PUBG or not.



Point E is where the Specificity becomes highest. Meaning there are no False Positives classified by the model. The model can correctly classify all the Negative class points! We would choose this point if our problem was to give perfect song recommendations to our users.

Going by this logic, can you guess where the point corresponding to a perfect classifier would lie on the graph?

Yes! It would be on the top-left corner of the ROC graph corresponding to the coordinate (0, 1) in the cartesian plane. It is here that both, the Sensitivity and Specificity, would be the highest and the classifier would correctly classify all the Positive and Negative class points.

## Understanding the AUC-ROC Curve in Python

Now, either we can manually test the Sensitivity and Specificity for every threshold or let [sklearn](https://www.analyticsvidhya.com/blog/2020/02/everything-you-should-know-scikit-learn/?utm_source=blog&utm_medium=auc-roc-curve-machine-learning" \t "_blank) do the job for us. We’re definitely going with the latter!

Let’s create our arbitrary data using the sklearn make\_classification method:

|  |  |
| --- | --- |
|  | from sklearn.datasets import make\_classification |
|  | from sklearn.model\_selection import train\_test\_split |
|  |  |
|  | # generate two class dataset |
|  | X, y = make\_classification(n\_samples=1000, n\_classes=2, n\_features=20, random\_state=27) |
|  |  |
|  | # split into train-test sets |
|  | X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=27) |

I will test the performance of two classifiers on this dataset:

|  |  |
| --- | --- |
|  | # train models |
|  | from sklearn.linear\_model import LogisticRegression |
|  | from sklearn.neighbors import KNeighborsClassifier |
|  |  |
|  | # logistic regression |
|  | model1 = LogisticRegression() |
|  | # knn |
|  | model2 = KNeighborsClassifier(n\_neighbors=4) |
|  |  |
|  | # fit model |
|  | model1.fit(X\_train, y\_train) |
|  | model2.fit(X\_train, y\_train) |
|  |  |
|  | # predict probabilities |
|  | pred\_prob1 = model1.predict\_proba(X\_test) |
|  | pred\_prob2 = model2.predict\_proba(X\_test) |

Sklearn has a very potent method roc\_curve() which computes the ROC for your classifier in a matter of seconds! It returns the FPR, TPR, and threshold values:

|  |  |
| --- | --- |
|  | from sklearn.metrics import roc\_curve |
|  |  |
|  | # roc curve for models |
|  | fpr1, tpr1, thresh1 = roc\_curve(y\_test, pred\_prob1[:,1], pos\_label=1) |
|  | fpr2, tpr2, thresh2 = roc\_curve(y\_test, pred\_prob2[:,1], pos\_label=1) |
|  |  |
|  | # roc curve for tpr = fpr |
|  | random\_probs = [0 for i in range(len(y\_test))] |
|  | p\_fpr, p\_tpr, \_ = roc\_curve(y\_test, random\_probs, pos\_label=1) |

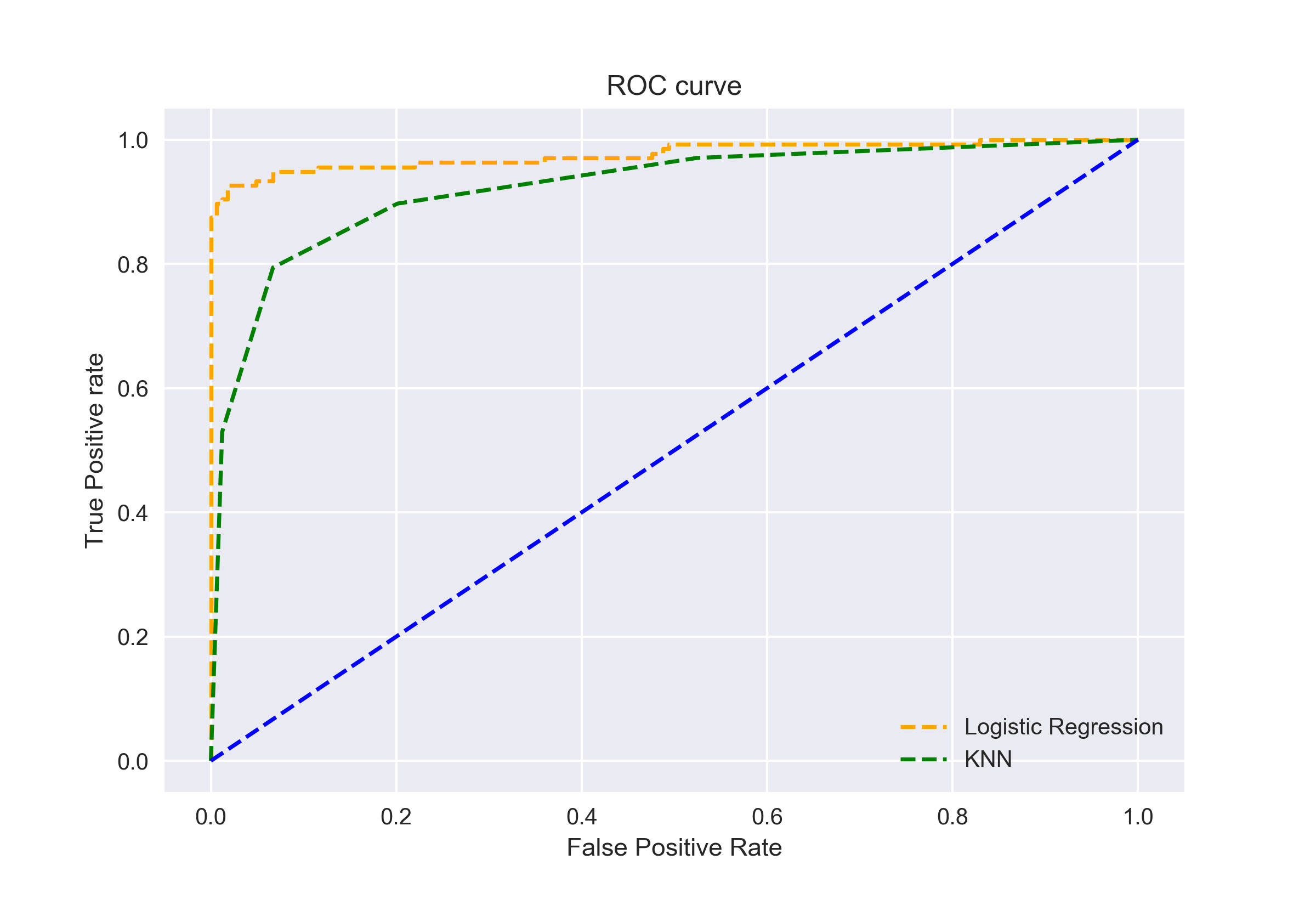
The AUC score can be computed using the roc\_auc\_score() method of sklearn:

|  |  |
| --- | --- |
|  | from sklearn.metrics import roc\_auc\_score |
|  |  |
|  | # auc scores |
|  | auc\_score1 = roc\_auc\_score(y\_test, pred\_prob1[:,1]) |
|  | auc\_score2 = roc\_auc\_score(y\_test, pred\_prob2[:,1]) |
|  |  |
|  | print(auc\_score1, auc\_score2) |

0.9761029411764707 0.9233769727403157

We can also plot the ROC curves for the two algorithms using [matplotlib](https://www.analyticsvidhya.com/blog/2020/02/beginner-guide-matplotlib-data-visualization-exploration-python/?utm_source=blog&utm_medium=auc-roc-curve-machine-learning):

|  |  |
| --- | --- |
|  | # matplotlib |
|  | import matplotlib.pyplot as plt |
|  | plt.style.use('seaborn') |
|  |  |
|  | # plot roc curves |
|  | plt.plot(fpr1, tpr1, linestyle='--',color='orange', label='Logistic Regression') |
|  | plt.plot(fpr2, tpr2, linestyle='--',color='green', label='KNN') |
|  | plt.plot(p\_fpr, p\_tpr, linestyle='--', color='blue') |
|  | # title |
|  | plt.title('ROC curve') |
|  | # x label |
|  | plt.xlabel('False Positive Rate') |
|  | # y label |
|  | plt.ylabel('True Positive rate') |
|  |  |
|  | plt.legend(loc='best') |
|  | plt.savefig('ROC',dpi=300) |
|  | plt.show(); |



It is evident from the plot that the AUC for the Logistic Regression ROC curve is higher than that for the KNN ROC curve. Therefore, we can say that logistic regression did a better job of classifying the positive class in the dataset.

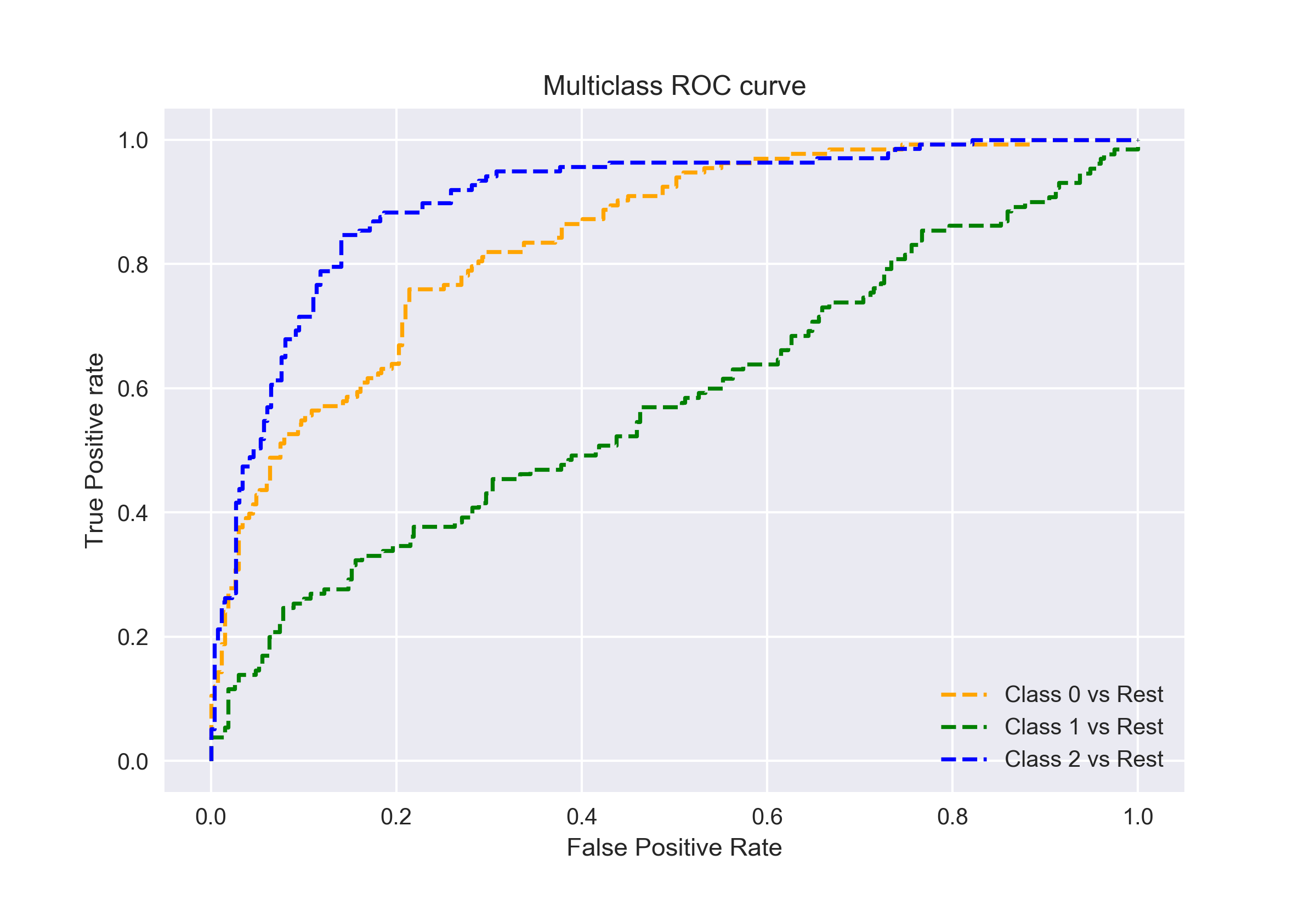
## AUC-ROC for Multi-Class Classification

Like I said before, the AUC-ROC curve is only for binary classification problems. But we can extend it to multiclass classification problems by using the One vs All technique.

So, if we have three classes 0, 1, and 2, the ROC for class 0 will be generated as classifying 0 against not 0, i.e. 1 and 2. The ROC for class 1 will be generated as classifying 1 against not 1, and so on.

The ROC curve for multi-class classification models can be determined as below:

|  |  |
| --- | --- |
|  | # multi-class classification |
|  | from sklearn.multiclass import OneVsRestClassifier |
|  | from sklearn.linear\_model import LogisticRegression |
|  | from sklearn.model\_selection import train\_test\_split |
|  | from sklearn.metrics import roc\_curve |
|  | from sklearn.metrics import roc\_auc\_score |
|  |  |
|  | # generate 2 class dataset |
|  | X, y = make\_classification(n\_samples=1000, n\_classes=3, n\_features=20, n\_informative=3, random\_state=42) |
|  |  |
|  | # split into train/test sets |
|  | X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.4, random\_state=42) |
|  |  |
|  | # fit model |
|  | clf = OneVsRestClassifier(LogisticRegression()) |
|  | clf.fit(X\_train, y\_train) |
|  | pred = clf.predict(X\_test) |
|  | pred\_prob = clf.predict\_proba(X\_test) |
|  |  |
|  | # roc curve for classes |
|  | fpr = {} |
|  | tpr = {} |
|  | thresh ={} |
|  |  |
|  | n\_class = 3 |
|  |  |
|  | for i in range(n\_class): |
|  | fpr[i], tpr[i], thresh[i] = roc\_curve(y\_test, pred\_prob[:,i], pos\_label=i) |
|  |  |
|  | # plotting |
|  | plt.plot(fpr[0], tpr[0], linestyle='--',color='orange', label='Class 0 vs Rest') |
|  | plt.plot(fpr[1], tpr[1], linestyle='--',color='green', label='Class 1 vs Rest') |
|  | plt.plot(fpr[2], tpr[2], linestyle='--',color='blue', label='Class 2 vs Rest') |
|  | plt.title('Multiclass ROC curve') |
|  | plt.xlabel('False Positive Rate') |
|  | plt.ylabel('True Positive rate') |
|  | plt.legend(loc='best') |
|  | plt.savefig('Multiclass ROC',dpi=300); |



ROC Curves are used to see how well your classifier can separate positive and negative examples and to identify the best threshold for separating them.

To be able to use the ROC curve, your classifier has to be ranking - that is, it should be able to rank examples such that the ones with higher rank are more likely to be positive. For example, [Logistic Regression](https://en.wikipedia.org/wiki/Logistic_regression) outputs probabilities, which is a score you can use for ranking.

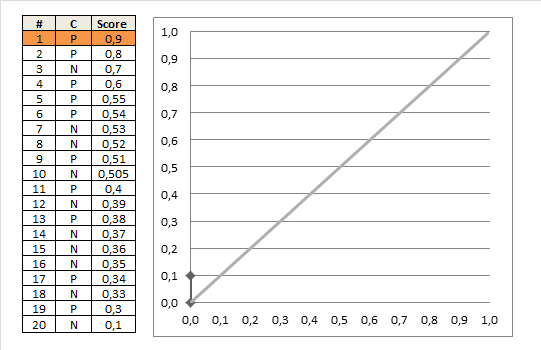
### **Drawing ROC curve**

Given a data set and a ranking classifier:

* order the test examples by the score from the highest to the lowest
* start in (0,0)(0,0)
* for each example xx in the sorted order
  + if x is positive, move 1/pos up
  + if x is negative, move 1/neg right

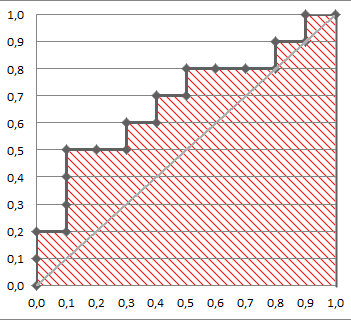
where pos and neg are the fractions of positive and negative examples respectively.

This nice gif-animated picture should illustrate this process clearer



On this graph, the y-axis is true positive rate, and the x-axis is false positive rate. Note the diagonal line - this is the baseline, that can be obtained with a random classifier. The further our ROC curve is above the line, the better.

### **Area Under ROC**



The area under the ROC Curve (shaded) naturally shows how far the curve from the base line. For the baseline it's 0.5, and for the perfect classifier it's 1.

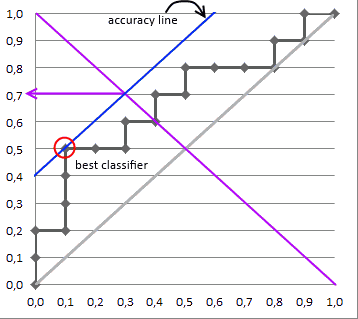
You can read more about AUC ROC in this question: [What does AUC stand for and what is it?](https://stats.stackexchange.com/questions/132777/what-does-auc-stand-for-and-what-is-it?)

### **Selecting the Best Threshold**

To select the best threshold you see each point of your ROC curve as a separate classifier. This mini-classifiers uses the score the point got as a boundary between + and - (i.e. it classifies as + all points above the current one)

Depending on the pos/neg fraction in our data set - parallel to the baseline in case of 50%/50% - you build ISO Accuracy Lines and take the one with the best accuracy.

Here's a picture that illustrates that and for details I again invite you to the reference



### **Reference**

<https://stats.stackexchange.com/questions/105501/understanding-roc-curve>

<http://mlwiki.org/index.php/ROC_Analysis>

## How Gradient Boosting Works

Gradient boosting involves three elements:

1. A loss function to be optimized.
2. A weak learner to make predictions.
3. An additive model to add weak learners to minimize the loss function.

### 1. Loss Function

The loss function used depends on the type of problem being solved.

It must be differentiable, but many standard loss functions are supported and you can define your own.

For example, regression may use a squared error and classification may use logarithmic loss.

A benefit of the gradient boosting framework is that a new boosting algorithm does not have to be derived for each loss function that may want to be used, instead, it is a generic enough framework that any differentiable loss function can be used.

### 2. Weak Learner

Decision trees are used as the weak learner in gradient boosting.

Specifically regression trees are used that output real values for splits and whose output can be added together, allowing subsequent models outputs to be added and “correct” the residuals in the predictions.

Trees are constructed in a greedy manner, choosing the best split points based on purity scores like Gini or to minimize the loss.

Initially, such as in the case of AdaBoost, very short decision trees were used that only had a single split, called a decision stump. Larger trees can be used generally with 4-to-8 levels.

It is common to constrain the weak learners in specific ways, such as a maximum number of layers, nodes, splits or leaf nodes.

This is to ensure that the learners remain weak, but can still be constructed in a greedy manner.

### 3. Additive Model

Trees are added one at a time, and existing trees in the model are not changed.

**A gradient descent procedure is used to minimize the loss when adding trees.**

**Traditionally, gradient descent is used to minimize a set of parameters, such as the coefficients in a regression equation or weights in a neural network. After calculating error or loss, the weights are updated to minimize that error.**

**Instead of parameters, we have weak learner sub-models or more specifically decision trees. After calculating the loss, to perform the gradient descent procedure, we must add a tree to the model that reduces the loss (i.e. follow the gradient). We do this by parameterizing the tree, then modify the parameters of the tree and move in the right direction by (reducing the residual loss.**

Generally this approach is called functional gradient descent or gradient descent with functions.

*One way to produce a weighted combination of classifiers which optimizes [the cost] is by gradient descent in function space*

The output for the new tree is then added to the output of the existing sequence of trees in an effort to correct or improve the final output of the model.

A fixed number of trees are added or training stops once loss reaches an acceptable level or no longer improves on an external validation dataset.

## Improvements to Basic Gradient Boosting

Gradient boosting is a greedy algorithm and can overfit a training dataset quickly.

It can benefit from regularization methods that penalize various parts of the algorithm and generally improve the performance of the algorithm by reducing overfitting.

In this this section we will look at 4 enhancements to basic gradient boosting:

1. Tree Constraints
2. Shrinkage
3. Random sampling
4. Penalized Learning

### 1. Tree Constraints

It is important that the weak learners have skill but remain weak.

There are a number of ways that the trees can be constrained.

A good general heuristic is that the more constrained tree creation is, the more trees you will need in the model, and the reverse, where less constrained individual trees, the fewer trees that will be required.

Below are some constraints that can be imposed on the construction of decision trees:

* **Number of trees**, generally adding more trees to the model can be very slow to overfit. The advice is to keep adding trees until no further improvement is observed.
* **Tree depth**, deeper trees are more complex trees and shorter trees are preferred. Generally, better results are seen with 4-8 levels.
* **Number of nodes or number of leaves**, like depth, this can constrain the size of the tree, but is not constrained to a symmetrical structure if other constraints are used.
* **Number of observations per split** imposes a minimum constraint on the amount of training data at a training node before a split can be considered
* **Minimim improvement to loss** is a constraint on the improvement of any split added to a tree.

### 2. Weighted Updates

The predictions of each tree are added together sequentially.

The contribution of each tree to this sum can be weighted to slow down the learning by the algorithm. This weighting is called a shrinkage or a learning rate.

*Each update is simply scaled by the value of the “learning rate parameter v”*

The effect is that learning is slowed down, in turn require more trees to be added to the model, in turn taking longer to train, providing a configuration trade-off between the number of trees and learning rate.

*Decreasing the value of v [the learning rate] increases the best value for M [the number of trees].*

It is common to have small values in the range of 0.1 to 0.3, as well as values less than 0.1.

*Similar to a learning rate in stochastic optimization, shrinkage reduces the influence of each individual tree and leaves space for future trees to improve the model.*

### 3. Stochastic Gradient Boosting

A big insight into bagging ensembles and random forest was allowing trees to be greedily created from subsamples of the training dataset.

This same benefit can be used to reduce the correlation between the trees in the sequence in gradient boosting models.

This variation of boosting is called stochastic gradient boosting.

*at each iteration a subsample of the training data is drawn at random (without replacement) from the full training dataset. The randomly selected subsample is then used, instead of the full sample, to fit the base learner.*

A few variants of stochastic boosting that can be used:

* Subsample rows before creating each tree.
* Subsample columns before creating each tree
* Subsample columns before considering each split.

Generally, aggressive sub-sampling such as selecting only 50% of the data has shown to be beneficial.

*According to user feedback, using column sub-sampling prevents over-fitting even more so than the traditional row sub-sampling*

### 4. Penalized Gradient Boosting

Additional constraints can be imposed on the parameterized trees in addition to their structure.

Classical decision trees like CART are not used as weak learners, instead a modified form called a regression tree is used that has numeric values in the leaf nodes (also called terminal nodes). The values in the leaves of the trees can be called weights in some literature.

As such, the leaf weight values of the trees can be regularized using popular regularization functions, such as:

* L1 regularization of weights.
* L2 regularization of weights.

*The additional regularization term helps to smooth the final learnt weights to avoid over-fitting. Intuitively, the regularized objective will tend to select a model employing simple and predictive functions.*