**Model Performance Metric and Ensembles**

**Performance measurement** is an indispensable process in any machine learning project. Our machine learning process may not always result in an optimum model with the expected accuracy. Hence, performance measurement is needed here to evaluate the effectiveness of a trained model in prediction.

There are a great variety of metrics which are used in performance measurement but in general, they can be categorized based on the model type, 1) Classifier or 2) Regressor. In this article, we will only focus on the classifier type measurement by introducing seven common performance metrics used in a classification project. The seven metrics are as below:

1. **Accuracy score**
2. **Confusion matrix**
3. **Precision**
4. **Recall**
5. **F1 Score**
6. **ROC Curve**
7. **AUROC**

While it may take a while to understand the underlying concept of some performance metrics above, the good news is that the implementation of those metrics has never been easier with [**Scikit-Learn**](https://scikit-learn.org/)**, a Python Machine Learning Library.**Scikit-Learn can simplify the performance measurement in just a few lines of Python code.

In the sections below, the concept of each performance metric will be explained by walking through a simple binary classification project based on the classic Wisconsin breast cancer dataset. The aim of this classification is to predict if a breast cancer tumour is malignant or benign.

To ensure you can follow the material, you are recommended to update your Python package with **Scikit-learn Version 0.23 or above**. Some functionalities presented here may not be supported in an earlier version of Scikit-learn.

1. Data Preparation

1.1 Data loading

We are going to use the Wisconsin breast cancer dataset for our classification task. To get the data, we can use a helper function from Scikit-Learn.

The *load\_breast\_cancer* is a Scikit-Learn helper function that enables us to fetch and load the desired breast cancer dataset into our Python environment. Here we call the helper function and assign the loaded breast cancer data into a variable, *br\_cancer*.

The loaded dataset has a Python dictionary structure which includes:

* A “*data”* key containing an array with one row per instance and one column per feature
* A “target” key containing an array of labels
* A “DESCR” key containing the description of the dataset

1.2 Data exploration

Now, let’s take a quick preview on our loaded dataset by showing the first three rows of records.

from sklearn.datasets import load\_breast\_cancer

br\_cancer = load\_breast\_cancer()

br\_cancer.target[0:3]

The records show that the breast cancer dataset consists of 30 numerical features (1 feature per column). If you are interested to know more about the details of the features, you may refer to the resource [here](https://scikit-learn.org/stable/datasets/toy_dataset.html#breast-cancer-dataset).

Next, let’s look at the first three target values.

class values

The first three target values

The first three values are *0* which signifies a “*malignant*” class. There is another possible value which is 1 that signifies a “*benign*” class.

Now, we can assign the feature values and the target value to variable *X* and *y,* respectively.

To explore the data dictionary further find out all the available keys:

print(br\_cancer.keys())

Output: The keys in that data dictionary are:

dict\_keys(['data', 'target', 'frame', 'target\_names', 'DESCR', 'feature\_names', 'filename', 'data\_module'])

You can find out the names of all features in the dataset:

list(br\_cancer.feature\_names)

**The output is a list of all possible features’ names:**

['mean radius',

'mean texture',

'mean perimeter',

'mean area',

'mean smoothness',

'mean compactness',

'mean concavity',

'mean concave points',

'mean symmetry',

'mean fractal dimension',

'radius error',

'texture error',

'perimeter error',

'area error',

'smoothness error',

'compactness error',

'concavity error',

'concave points error',

'symmetry error',

'fractal dimension error',

'worst radius',

'worst texture',

'worst perimeter',

'worst area',

'worst smoothness',

'worst compactness',

'worst concavity',

'worst concave points',

'worst symmetry',

'worst fractal dimension']

To find out the name of the class labels you can use:

list(br\_cancer.target\_names)

The output is two class labels:

['malignant', 'benign']

Or simply to find the class for specific instances (patients)

br\_cancer.target[[10, 50, 85]]

The output is:

array([0, 1, 0])

If you want further information regarding the dataset you could try:

print(br\_cancer.DESCR)

1.3 Standardization of data

At this point, you might have noticed that the value ranges of the 30 features are not consistent and this may affect the accuracy of our model prediction. To circumvent this issue, we can use the Scikit-Learn *StandardScaler* to re-scale the feature values to their unit variance.

We create a *StandardScaler*object and use its*fit\_transform* method to re-scale our *X*values and assign the transformed values to a variable, *X\_scaled*.

X, y = br\_cancer['data'], br\_cancer['target']

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

1.4 Splitting data into training and test set

Prior to training our classifiers, we need to split our dataset into training and test set. To do so, we can make use of Scikit-Learn function, *train\_test\_split.*

The *train\_test\_split* can automatically randomize our dataset and split it into training and test set based on the *test\_size* that we assign. In this case, we set the *test\_size*to 0.3 in order to set apart 30% of our dataset as the test set and the rest will be used as the training set. The training and test set are assigned to variable *X\_train*,*X\_test*, *y\_train* and*y\_test*, respectively.

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.3, random\_state=42)

2. Training classifiers

We are going to train three classifiers based on the k-nearest neighbours algorithm (KNN), Stochastic Gradient Descent (SGD) and logistic regression, respectively. The reason we build three classifiers is that we can compare their performance using the metrics we are going to cover in this article.

Scikit-Learn offers KNN, SGD and logistic regression classifier for us. All that we need to do is just to import those classifiers (*KNeighboursClassifier, SGDClassifier*and*LogisticRegression*) into our Python environment. Next, we create an object for each classifier and then use the *fit* method to start training the respective classifier using the same training set (*X\_train* & *y\_train*) as its input.

*(The entire training process should only take a few seconds as we are dealing with a very small dataset. )*

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.3, random\_state=42)

from sklearn.neighbors import KNeighborsClassifier

from sklearn.linear\_model import SGDClassifier

from sklearn.linear\_model import LogisticRegression

knn\_model = KNeighborsClassifier(n\_neighbors=3)

knn\_model.fit(X\_train, y\_train)

sgd\_model = SGDClassifier(random\_state=42)

sgd\_model.fit(X\_train, y\_train)

log\_model = LogisticRegression()

log\_model.fit(X\_train, y\_train)

Now, let us try to use our trained classifiers to make predictions by using the model in-built method, *predict*.

y\_pred\_knn = knn\_model.predict(X\_test)

y\_pred\_sgd = sgd\_model.predict(X\_test)

y\_pred\_log = log\_model.predict(X\_test)

We fetch the test set to the *predict*methodand the prediction outcomes of each classifier are stored in three variables, *y\_pred\_knn,* *y\_pred\_sgd*, and *y\_pred\_log*, respectively.

3. Performance Measurement

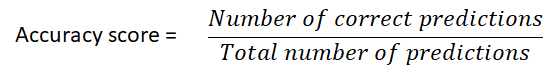
This is the main topic of this article. Here we are going to cover seven performance metrics for classifier by delving into their underlying concept and their implementation using Scikit-Learn.

Let’s start with the simplest one, *accuracy score*.

3.1 Accuracy Score

**3.1.1 Concept**

Accuracy score is probably the most straightforward metric in measuring a classifier’s performance. It is a metric that shows us a fraction of the correct prediction. The formula is as below:



The formula of accuracy score

Let’s say if there are 100 records in our test set and our classifier manages to make an accurate prediction for 92 of them, the accuracy score would be 0.92.

**3.1.2 Implementation in Scikit-Learn**

Scikit-Learn provides a function, *accuracy\_score*, which accepts the true value and predicted value as its input to calculate the accuracy score of a model.

Firstly, we import the *accuracy\_score* function from the *metrics* module of Scikit-Learn. Next, we input the *y\_test* as true value and *y\_pred\_knn*, *y\_pred\_sgd* & *y\_pred\_log* as the predicted values to the *accuracy\_score* function. The resulting accuracy scores are assigned to three variables and the scores are printed.

from sklearn.metrics import accuracy\_score

knn\_score = accuracy\_score(y\_test, y\_pred\_knn)

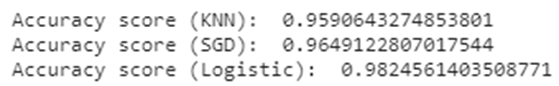
sgd\_score = accuracy\_score(y\_test, y\_pred\_sgd)

log\_score = accuracy\_score(y\_test, y\_pred\_log)

print("Accuracy score (KNN): ", knn\_score)

print("Accuracy score (SGD): ", sgd\_score)

print("Accuracy score (Logistic): ", log\_score)



The resulting accuracy score for three classifiers

The resulting accuracy scores reveal that the logistic regression classifier shows the highest performance among the three. However, all three classifiers are generally high (> 95% accuracy) and this gives us a rough idea that those classifiers are effective in identifying a malignant or benign tumour.

3.2 Confusion matrix

**3.2.1 Concept**

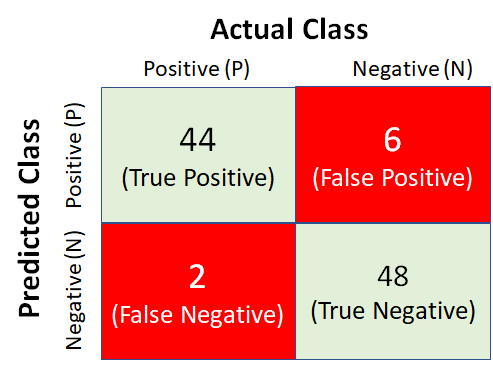
Accuracy scores only give us a fraction of correct prediction in overall. What if we would like to know more about our classifier in the following aspects:

* how many malignant tumours in our samples are correctly and incorrectly predicted?
* how many benign tumours in our samples are correctly and incorrectly predicted?

These questions are essential as no single patient will be happy to receive a misreported medical result. Ideally, the prediction of both malignant and benign classes should achieve high accuracy to minimize the false report. If our classifier shows an obvious bias to either malignant or benign class, we might need to retrain our classifiers by adjusting some hyperparameters or fetching additional data into the training pipeline.

Hence, a better way to evaluate the performance of a classifier is to use **confusion matrix**. A confusion matrix (also known as error matrix)is a two-dimensional table that permits visualization of correctly labelled and mislabeled instances by a classifier.

Given a sample of 100 instances in a test set where each instance belongs to either positive or negative class. The performance of a classifier on that test set can be represented as the confusion matrix below:



A sample of the confusion matrix

Each column in the confusion matrix above represents an actual class while each row represents a predicted class. The top-left quadrant shows that there are 44 positive instances in the test set which are predicted accurately because the actual class (positive) is matched with the predicted class (positive). This type of outcome is known as **True Positive (TP)**. On another hand, the top -right quadrant shows 6 negative instances which are wrongly predicted as the positive class and these are known as **False Positive(FP)**.

At the bottom-right quadrant, there are 48 negative instances which are predicted correctly and are known as **True Negative (TN)**. Lastly, the bottom-left quadrant displays 2 positive instances which are wrongly predicted as a negative class and these are called **False Negative (FN)**.

From the sample confusion matrix above, we can observe that the classifier shows a slightly better performance in identifying negative class than a positive class. Ideally, a good classifier would have an as least possible number of false positive and false negative as possible. This means the higher the non-zero number in the main diagonal of a confusion matrix, the better performance of a classifier.

**3.2.2** **Implementation in Scikit-Learn**

While the explanation of the confusion matrix may sound verbose, the implementation just requires few lines of codes (thanks to Scikit-Learn). Let’s try to use *confusion\_matrix* function from Scikit-Learn to build a confusion matrix table for our KNN classifier.

Firstly, we import the *confusion\_matrix* function into our Python environment. By passing two required parameters, true value (*y\_test*) and predicted value (*y\_pred\_knn*) to the function, a confusion matrix represented as a 2D array.

from sklearn.metrics import confusion\_matrix

confusion\_matrix(y\_test, y\_pred\_knn)

confusion\_matrix(y\_test, y\_pred\_sgd)

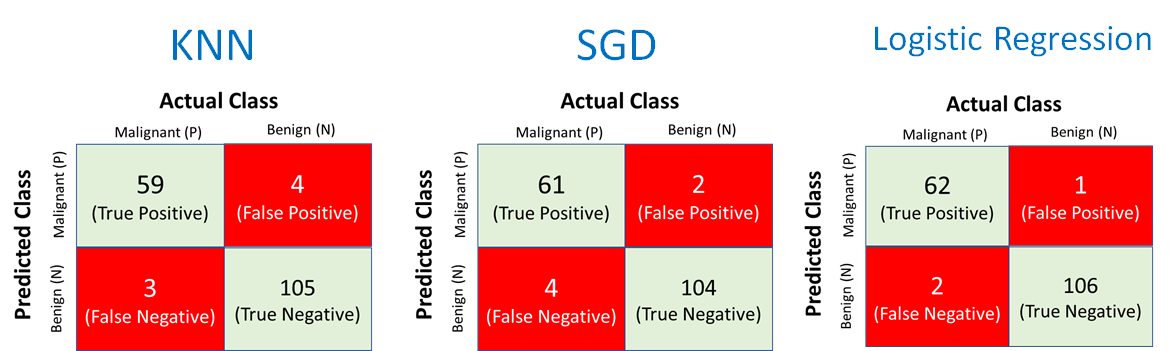
confusion\_matrix(y\_test, y\_pred\_log)

To ease our task to interpret the 2D array in our context of breast cancer prediction, we may try to map the 2D array to our earlier confusion matrix table **by referring Positive class as malignant and Negative class as benign.**

Given a total number of 171 instances in the test set, the confusion matrix reveals there are 59 malignant tumours which are correctly classified (True Positive), 4 benign tumours are wrongly classified as malignant (False Positive), 105 benign tumours are correctly classified (True Negative) and another 3 malignant tumours are wrongly classified as benign (False Negative).

Basically, the KNN classifier demonstrates a decent prediction job for both malignant and benign tumour. By repeating the similar steps above, we can also create the confusion matrix for our SGD classifier and logistic regression classifier and then compare them with KNN classifier.

Now, let’s look at all three confusion matrixes and make a comparison among them.



Comparison of confusion matrixes among three classifiers

From the result above, we can observe that SGD classifier show a slightly higher number of false negative even though its overall accuracy score is better than KNN. If we are only given two choices of classifiers, KNN or SGD, and also aim for a lower false negative, KNN might be a better option even though its overall accuracy score is slightly lower than SGD.

On another hand, it is pretty obvious that the logistic regression classifier is still the winner here as it shows the lowest number of false positive and false negative among the three classifiers.

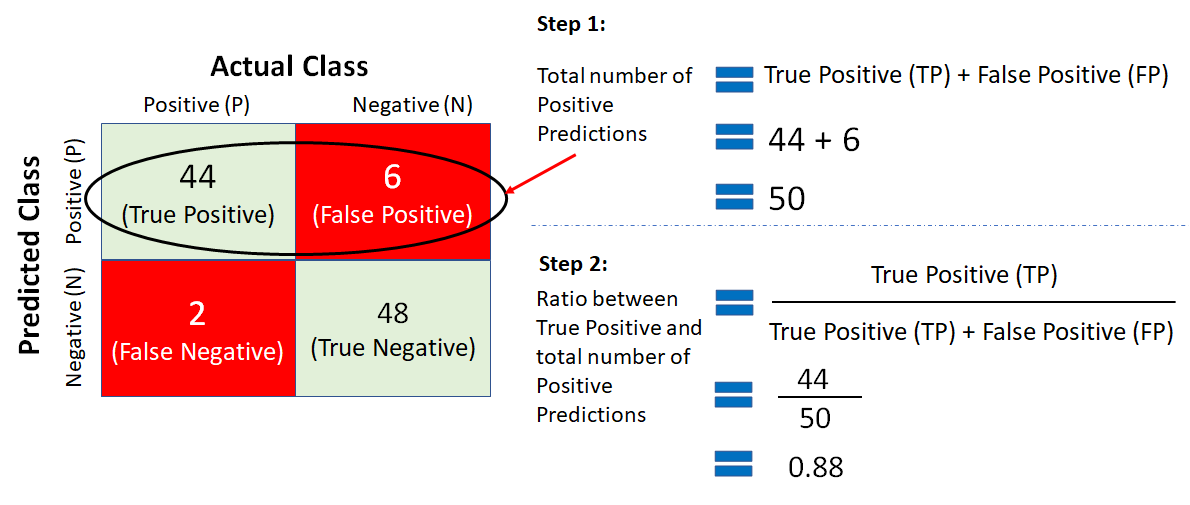
3.3 Precision, Recall and F1 Score

**3.3.1 Concept**

The confusion matrix shows us the exact number of true positive, false positive, true negative and false negative in a classification result. In fact, we can go further to derive several useful metrics from the confusion matrix.

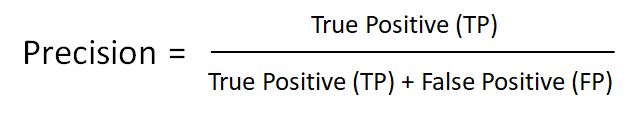
Firstly, we may ask ourselves: **What is the fraction of positive predictions which are true positive?**

To address the question, let us walk through a simple mathematic calculation logic as presented below.



Calculation of precision

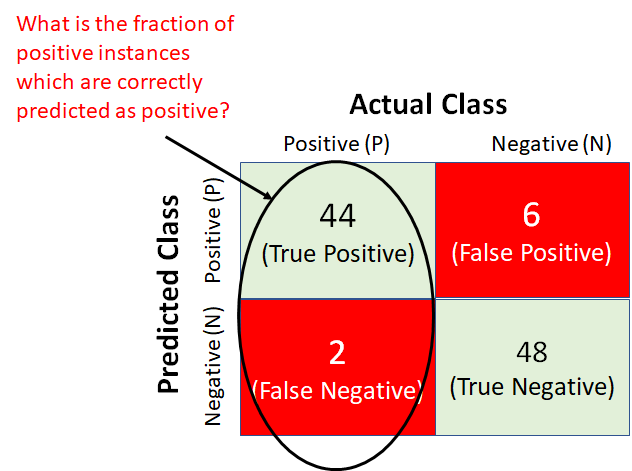
The ratio between the number of True Positives and the total number of Positives are the metric known as **precision.**The formula of precision is given below:



Precision formula

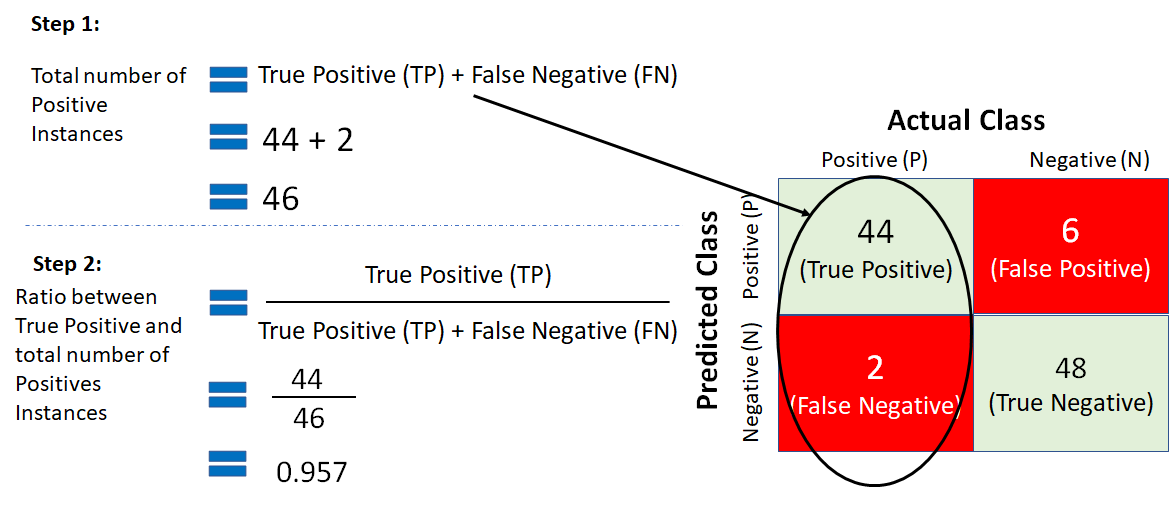
Precision gives us a quantifier to reveal the fraction of the predicted positive instances which are true positive. The sample classifier above achieves the precision score of 0.88 and this also means 88% of predicted positives are true positive.

Now, let us think of another subtle question: **What is the fraction of positive instances which are correctly predicted as positive?**



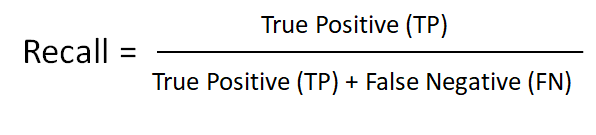
The fraction of positive instances which are predicted as positive

Once again we are going to use a simple math logics to address the question.



Calculation of recall

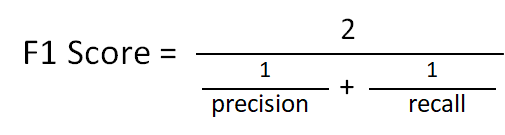
The calculation above result in a ratio between true positive and the total number of positive instances and this ratio is known as **recall**. Recall is also called sensitivity or true positive rate (TPR). The formula of recall is given below:



Recall formula

The sample classifier above hits a recall score of 0.957 which is higher than its precision. This means among all the 46 positive instances, 95.7% of them are correctly predicted as positive.

Precision and recall can also be combined into a single metric called the **F1 Score**. The F1 Score is the harmonic mean of precision and recall.



F1 Score formula

The F1 Score formula may seem a little daunting at the first glance but it is just a specific mean that gives much more weight to low values (in comparison to the regular mean that treats all values equally). A high F1 score indicates a similar precision and recall. For example, if we apply the F1 Score formula to our sample classifier, we will get a score of approximately 0.917. This shows both the precision and recall rates of the sample classifier are quite high.

This is important to learn that precision and recall may not be always similar and in some cases, we would prefer a higher precision than recall or vice versa. For example, we would expect a higher recall rate in medical use cases such as cancer detection. It is always better to enable our classifier to accurately identify as many true positive cancer instances as possible rather than letting go of some positive instances unnoticed. On another hand, bank analysts may prefer their loan classifier performs with higher precision so that they won’t accidentally reject a potential customer and lose the business.

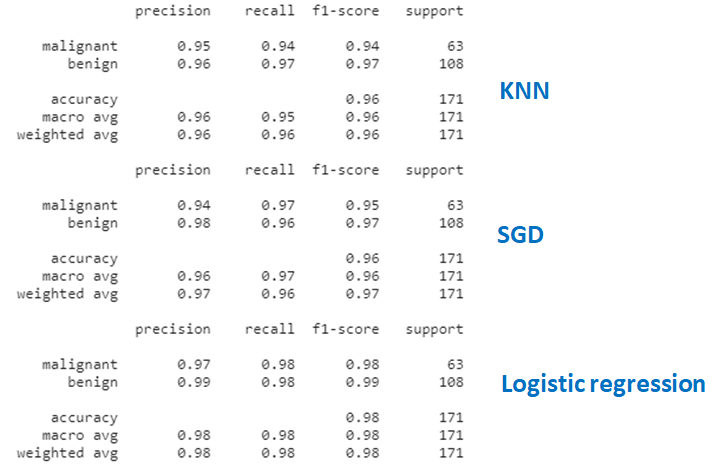
Unfortunately, we can’t always tune our classifier to achieve almost equally high precision and recall rate. Increasing precision results in a reduction of recall and vice versa. This is known as the **precision/recall tradeoff.**

**3.3.2 Implementation in Scikit-Learn**

Now it’s time to get our hand dirty again and implement the metrics we cover in this section using Scikit-Learn. The precision, recall and F1 Score metrics can easily be obtained using *classification\_report* function offered by Sckit-Learn.

As usual, we import the required *classification\_report* function into our Python environment. Next, let’s try to extract the target names (malignant and benign)from our dataset and assign it to a variable, *targets*. The names held by the *targets* variable will be displayed as part of the final result later.

At last, we pass three parameters, true value (*y\_test*), predicted value (*y\_pred\_knn, y\_pred\_sgd & y\_pred\_log*) and target names (*targets*), to the *classification\_report* function and print out the result as shown below.



Classification report for three classifiers

from sklearn.metrics import classification\_report

targets = br\_cancer.target\_names

print(classification\_report(y\_test, y\_pred\_knn, target\_names=targets)) # knn classification report

print(classification\_report(y\_test, y\_pred\_sgd, target\_names=targets)) # sgd classification report

print(classification\_report(y\_test, y\_pred\_log, target\_names=targets)) # logistic classification report

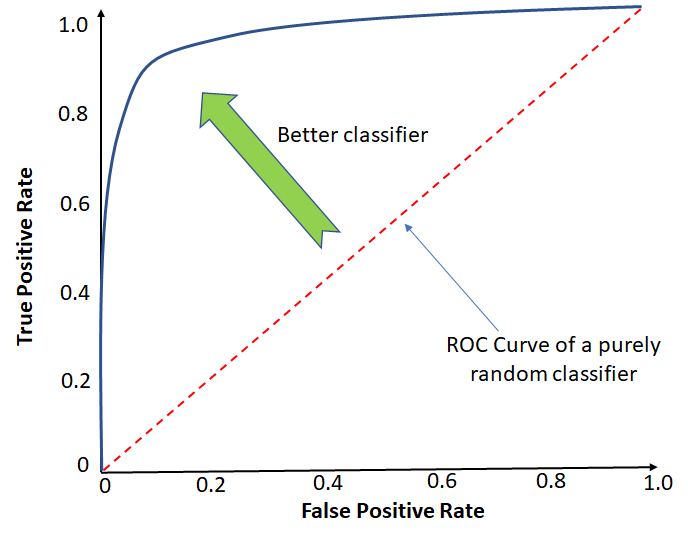
In general, there are not a very sharp difference in precision, recall and F1 scores among the three classifiers. Logistic regression is still a clear winner here by having a F1-score of 0.98 and 0.99 for malignant and benign class, respectively. Besides, its high recall rate, 0.98, has proven itself as a reliable classifier to make a reliable detection on a malignant tumour.

3.4 The ROC Curve and AUROC

**3.4.1 Concept**

**The receiver operating characteristic (ROC) curve** offers us a visual approach to examine the performance of our trained classifier. It is a curve that plots the true positive rate (TPR)of a classifier against its false positive rate (FPR). As highlighted in the earlier section, the TPR is also known as recall. On another hand, the FPR is the ratio of negative instances that are wrongly predicted as positive.

A sample of the ROC curve is given below:



ROC Curve

There is a tradeoff between the TPR and FPR — the higher the TPR, the more FPR will be produced by the classifier. The diagonal dotted line in the middle represents the ROC curve which is a random classifier.

The ROC curve of a good classifier should stay as far away from the diagonal line as possible by heading towards the top left corner. Based on this rationale, we can measure and compare our classifiers’ performance by calculating the area under the curve (AUC) which will result in a score called **AUROC (**Area Under the Receiver Operating Characteristics**)**. A perfect AUROC should have a score of 1 whereas a random classifier will have a score of 0.5.

**3.4.2 Implementation in Scikit-Learn**

Scikit-Learn offers a handy function, *RocCurveDisplay*, that we can use to create the ROC curve and also calculate the AUROC score.

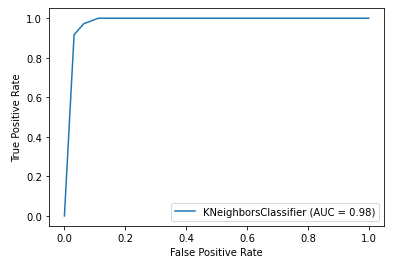
The *RocCurveDisplay* function only requires three parameters which are classifier (*knn\_model*), test set (*X\_test*), and true value (*y\_test*) and it will automatically generate the ROC curve and calculate AUROC score for our KNN classifier on the fly.

from sklearn.metrics import RocCurveDisplay

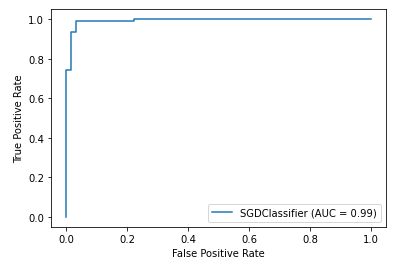
sgd\_disp = RocCurveDisplay.from\_estimator(sgd\_model, X\_test, y\_test)

log\_disp = RocCurveDisplay.from\_estimator(log\_model, X\_test, y\_test)

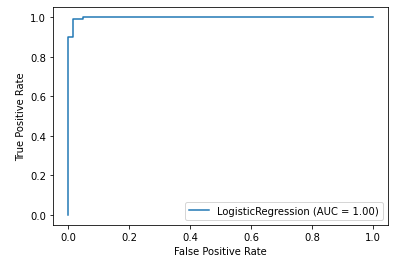
knn\_disp = RocCurveDisplay.from\_estimator(knn\_model, X\_test, y\_test)



ROC Curve and AUROC Score for KNN Classifier

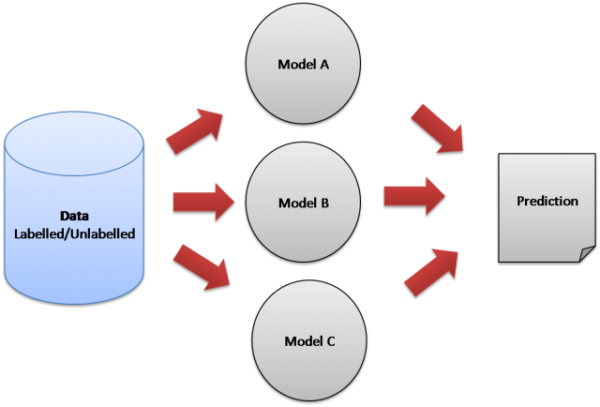


ROC Curve and AUROC Score for SGD Classifier



ROC Curve and AUROC Score for Log Reg Classifier

The AUROC for our logistic regression classifier hits the perfect score which is 1. By looking at the results of all the metrics that we cover here, we can conclude that the logistic regression classifier is the top performer among the three. This classifier is proven as the most reliable model to predict the type of breast cancer tumour.



Ensemble Learning

Ensemble learning uses multiple machine learning models to try to make better predictions on a dataset. An ensemble model works by training different models on a dataset and having each model make predictions individually. The predictions of these models are then combined in the ensemble model to make a final prediction.

Every model has its strengths and weaknesses. Ensemble models can be beneficial by combining individual models to help hide the weaknesses of an individual model.

In this tutorial, we will be using a Voting Classifier in which the ensemble model makes the prediction by majority vote. For example, if we use three models and they predict [1, 0, 1] for the target variable, the final prediction that the ensemble model would make would be 1, since two out of the three models predicted 1.

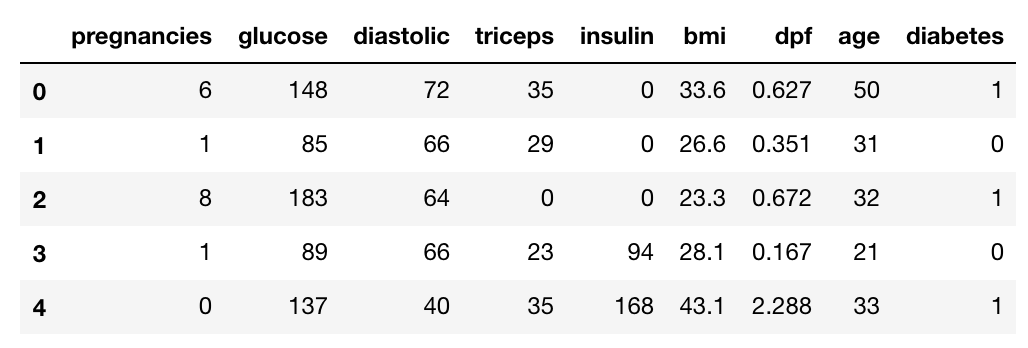
We will use three different models to put into our Voting Classifier: k-Nearest Neighbors, Random Forest, and Logistic Regression. We will use the Scikit-learn library in Python to implement these methods and use the diabetes dataset in our example.

Note: Ensemble models can also be used for regression problems, where the ensemble model will use either the mean output of the different models or weighted averages for its final prediction.

## Reading in the training data

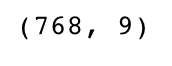
The first step is to read in the data we will use as input. For this example, we are using the diabetes dataset. To start, we will use the Pandas library to read in the data.

import pandas as pd#read in the dataset  
df = pd.read\_csv(‘data/diabetes\_data.csv’)#take a look at the data  
df.head()



Next, let’s see how much data we have. We will call the ‘shape’ function on our dataframe to see how many rows and columns there are in our data. The rows indicate the number of patients and the columns indicate the number of features (age, weight, etc.) in the dataset for each patient.

#check dataset size  
df.shape



## Split up the dataset into inputs and targets

Now let’s split up our dataset into inputs (X) and our target (y). Our input will be every column except ‘diabetes’ because ‘diabetes’ is what we will be attempting to predict. Therefore, ‘diabetes’ will be our target.

We will use the pandas ‘drop’ function to drop the column ‘diabetes’ from our dataframe and store it in the variable ‘X’.

#split data into inputs and targets  
X = df.drop(columns = [‘diabetes’])  
y = df[‘diabetes’]

## Split the dataset into train and test data

Now we will split the dataset into into training data and testing data. The training data is the data that the model will learn from. The testing data is the data we will use to see how well the model performs on unseen data.

Scikit-learn has a function we can use called ‘train\_test\_split’ that makes it easy for us to split our dataset into training and testing data.

from sklearn.model\_selection import train\_test\_split#split data into train and test sets  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, stratify=y)

‘train\_test\_split’ takes in 5 parameters. The first two parameters are the input and target data we split up earlier. Next, we will set ‘test\_size’ to 0.3. This means that 30% of all the data will be used for testing, which leaves 70% of the data as training data for the model to learn from.

Setting ‘stratify’ to y makes our training split represent the proportion of each value in the y variable. For example, in our dataset, if 25% of patients have diabetes and 75% don’t have diabetes, setting ‘stratify’ to y will ensure that the random split has 25% of patients with diabetes and 75% of patients without diabetes.

## Building the models

Next, we have to build our models. Each model we build has a set of hyper parameters that we can tune. Tuning parameters is when you go through a process to find the optimal parameters for your model to improve accuracy. We will use grid search to find the optimal hyperparamters for each model.

Grid search works by training our model multiple times on a range of parameters that we specify. That way, we can test our model with each hyperparameter value and figure out the optimal values to get the best accuracy results.

## k-Nearest Neighbors (k-NN)

The first model we will build is k-Nearest Neighbors (k-NN). k-NN models work by taking a data point and looking at the ‘k’ closest labeled data points. The data point is then assigned the label of the majority of the ‘k’ closest points.

For example, if k = 5, and 3 of points are ‘green’ and 2 are ‘red’, then the data point in question would be labeled ‘green’, since ‘green’ is the majority (as shown in the above graph).

Here is the code:

import numpy as np  
from sklearn.model\_selection import GridSearchCV  
from sklearn.neighbors import KNeighborsClassifier#create new a knn model  
knn = KNeighborsClassifier()#create a dictionary of all values we want to test for n\_neighbors  
params\_knn = {‘n\_neighbors’: np.arange(1, 25)}#use gridsearch to test all values for n\_neighbors  
knn\_gs = GridSearchCV(knn, params\_knn, cv=5)#fit model to training data  
knn\_gs.fit(X\_train, y\_train)

First, we will create a new k-NN classifier. Next, we need to create a dictionary to store all the values we will test for ‘n\_neighbors’, which is the hyperparameter we need to tune. We will test 24 different values for ‘n\_neighbors’. Then we will create our grid search, inputing our k-NN classifier, our set of hyperparamters and our cross validation value.

Cross-validation is when the dataset is randomly split up into ‘k’ groups. One of the groups is used as the test set and the rest are used as the training set. The model is trained on the training set and scored on the test set. Then the process is repeated until each unique group as been used as the test set.

In our case, we are using 5-fold cross validation. The dataset is split into 5 groups, and the model is trained and tested 5 separate times so each group would get a chance to be the test set. This is how we will score our model running with each hyperparamter value to see which value for ‘n\_neighbors’ gives us the best score.

Then we will use the ‘fit’ function to run our grid search.

Now we will save our best k-NN model to ‘knn\_best’ using the ‘best\_estimator\_’ function and check what the best value was for ‘n\_neighbors’.

#save best model  
knn\_best = knn\_gs.best\_estimator\_

#check best n\_neigbors value  
print(knn\_gs.best\_params\_)

check best n_neigbors value

## Random Forest

The next model we will build is a random forest. A random forest is considered an ensemble model in itself, since it is a collection of decision trees combined to make a more accurate model.

from sklearn.ensemble import RandomForestClassifier#create a new random forest classifier  
rf = RandomForestClassifier()

#create a dictionary of all values we want to test for n\_estimators  
params\_rf = {‘n\_estimators’: [50, 100, 200]}

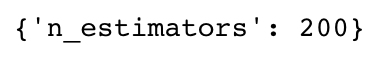
#use gridsearch to test all values for n\_estimators  
rf\_gs = GridSearchCV(rf, params\_rf, cv=5)#fit model to training data  
rf\_gs.fit(X\_train, y\_train)

We will create a new random forest classifier and set the hyperparameters we want to tune. ‘n\_estimators’ is the number of trees in our random forest. Then we can run our grid search to find the optimal number of trees.

Just like before, we will save our best model and print the best ‘n\_estimators’ value.

#save best model  
rf\_best = rf\_gs.best\_estimator\_

#check best n\_estimators value  
print(rf\_gs.best\_params\_)



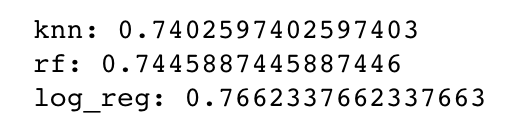
## Logistic Regression

Our last model is logistic regression. Even though it has ‘regression’ in its name, logistic regression is a classification method. This one is more simple since we won’t tune any hyperparameters. We just need to create and train the model.

from sklearn.linear\_model import LogisticRegression#create a new logistic regression model  
log\_reg = LogisticRegression()#fit the model to the training data  
log\_reg.fit(X\_train, y\_train)

Now let’s check the accuracy scores of all three of our models on our test data.

#test the three models with the test data and print their accuracy scoresprint(‘knn: {}’.format(knn\_best.score(X\_test, y\_test)))  
print(‘rf: {}’.format(rf\_best.score(X\_test, y\_test)))  
print(‘log\_reg: {}’.format(log\_reg.score(X\_test, y\_test)))



As you can see from the output, logistic regression is the most accurate out of the three.

## Voting Classifier

Now that we’ve built our three individual models, it’s time we built our voting classifier.

Here is the code:

from sklearn.ensemble import VotingClassifier

#create a dictionary of our models  
estimators=[(‘knn’, knn\_best), (‘rf’, rf\_best), (‘log\_reg’, log\_reg)]

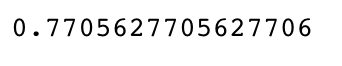
#create our voting classifier, inputting our models  
ensemble = VotingClassifier(estimators, voting=’hard’)

First, let’s place our three models in an array called ‘estimators’. Next, we will create our voting classifier. It takes two inputs. The first is our estimator array of our three models. We will set the voting parameter to hard, which tells our classifier to make predicitons by majority vote.

Now we can fit our ensemble model to our training data and score it on our testing data.

#fit model to training data  
ensemble.fit(X\_train, y\_train)

#test our model on the test data  
ensemble.score(X\_test, y\_test)



Awesome! Our ensemble model performed better than our individual k-NN, random forest and logistic regression models!