CSL7020: Machine Learning-1 Programming Assignment -1

KADAPALA RAKESH REDDY – M22AI608 – MTech – 2nd Sem – IITJ

Q1>

Project Goal:

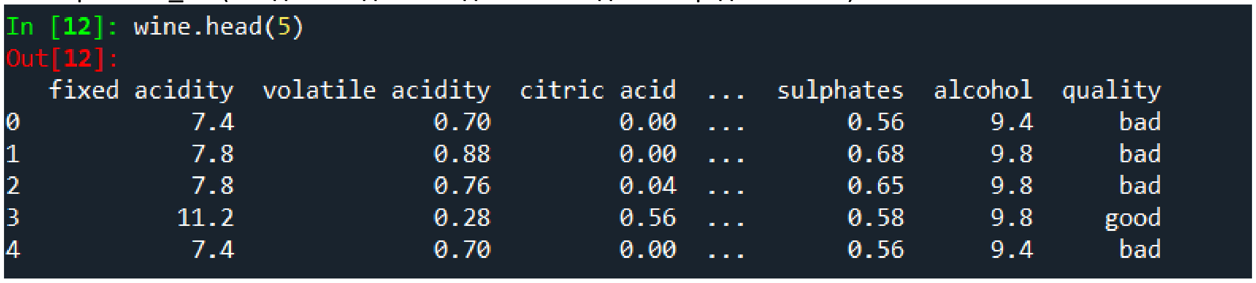
Classification Task: Use this dataset: Wine Quality dataset
(a) Implement Naive Bayes classifier algorithm for two class classification
(b) Implement Logistic regression.
(c) Use cross validation technique to solve the problem
(d) Compare both the methods using metrics: Accuracy, Confusion matrix, ROC curve and F1 Score.
(e) Explain which method performs better and why.

Ans>

Naïve Bayes Classifier Algorithm
In ML classifying the wine is known as multiclass classification as there are multiple types of wine need that to identified.
One of the popular method to approach the multiclass classifier is to use naïve Bayes model.
Naïve Bayes algorithm is a supervised learning algorithm, which is based on Bayes theorem and used for solving classification problems.
It is mainly used in text classification that includes a high-dimensional training dataset.
Naïve Bayes Classifier is one of the simple and most effective Classification algorithms which helps in building the fast machine learning models that can make quick predictions.
It is a probabilistic classifier, which means it predicts on the basis of the probability of an object.
Data Analysis:
Prediction of wine quality?
Importing Libraries:
import pandas as pd   #library to handle dataframe
import numpy as np   #library for scientific computing
import matplotlib.pyplot as plt   #library for plots
from sklearn import naive_bayes
from sklearn import metrics

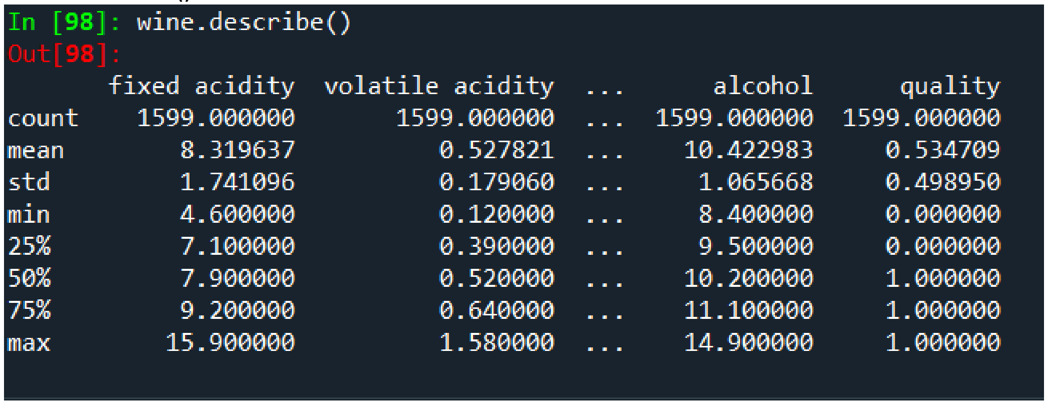
pandas are used for data analysis.
NumPy is for n-dimensional array.
matplotlib both have similar functionalities which are used for visualization.

Importing wines data:
wine=pd.read_csv("C:\\Users\\Admin\\OneDrive\\Desktop\\wine.csv")



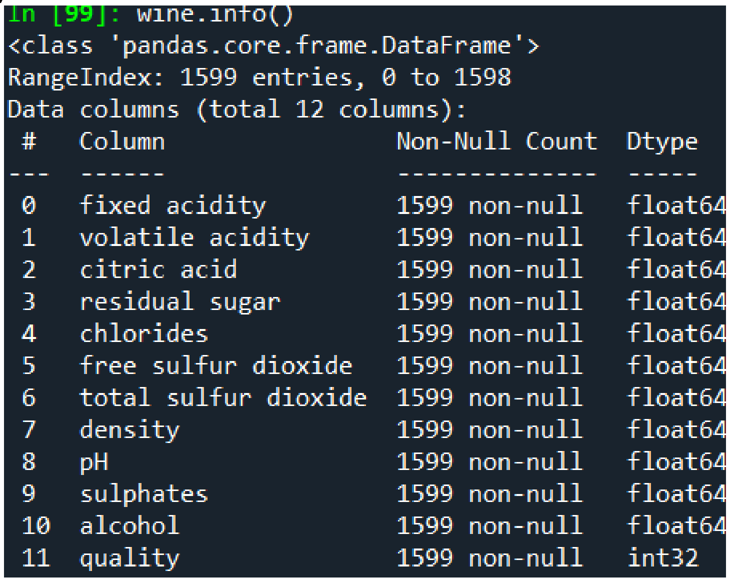
```
In [12]: wine.head(5)
Out[12]:
   fixed acidity  volatile acidity  citric acid  ...  sulphates  alcohol  quality
0            7.4              0.70         0.00  ...       0.56      9.4      bad
1            7.8              0.88         0.00  ...       0.68      9.8      bad
2            7.8              0.76         0.04  ...       0.65      9.8      bad
3           11.2              0.28         0.56  ...       0.58      9.8     good
4            7.4              0.70         0.00  ...       0.56      9.4      bad
```

#Describing the data
Wine.describe()



```
In [98]: wine.describe()
Out[98]:
       fixed acidity  volatile acidity  ...      alcohol      quality
count    1599.000000       1599.000000  ...  1599.000000  1599.000000
mean        8.319637          0.527821  ...    10.422983     0.534709
std         1.741096          0.179060  ...     1.065668     0.498950
min         4.600000          0.120000  ...     8.400000     0.000000
25%         7.100000          0.390000  ...     9.500000     0.000000
50%         7.900000          0.520000  ...    10.200000     1.000000
75%         9.200000          0.640000  ...    11.100000     1.000000
max        15.900000          1.580000  ...    14.900000     1.000000
```

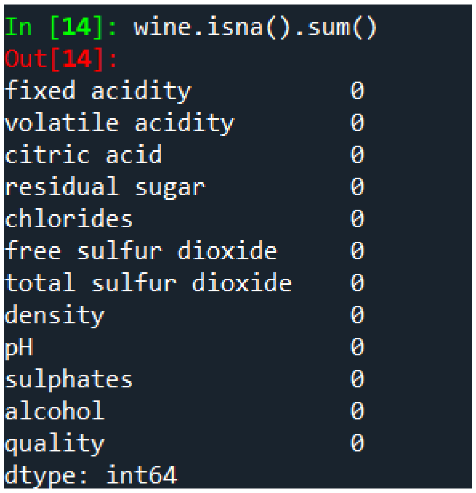
#Info of the data
Wine.info()



```
In [99]: wine.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1599 entries, 0 to 1598
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   fixed acidity         1599 non-null   float64
 1   volatile acidity      1599 non-null   float64
 2   citric acid           1599 non-null   float64
 3   residual sugar        1599 non-null   float64
 4   chlorides             1599 non-null   float64
 5   free sulfur dioxide   1599 non-null   float64
 6   total sulfur dioxide  1599 non-null   float64
 7   density               1599 non-null   float64
 8   pH                    1599 non-null   float64
 9   sulphates             1599 non-null   float64
 10  alcohol               1599 non-null   float64
 11  quality               1599 non-null   int32
```

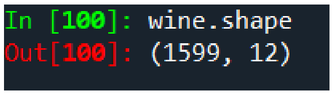
As we see in the above image, there is vital information on features and with this information, we will process our next work.

#Null Values



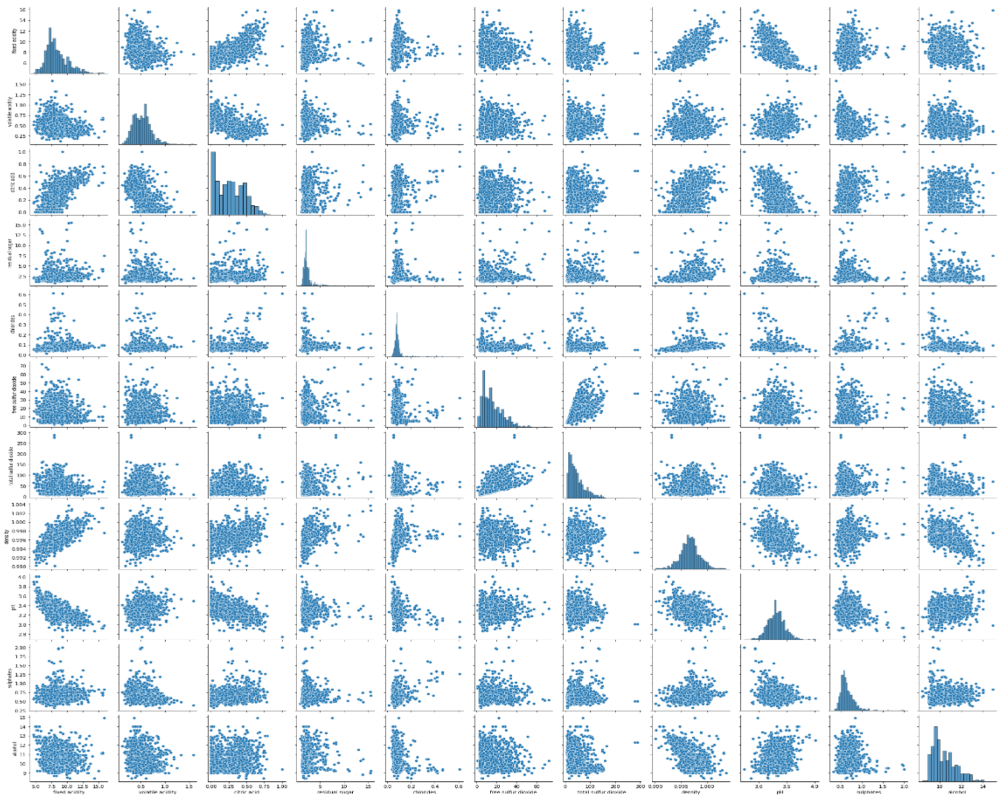
```
In [14]: wine.isna().sum()
Out[14]:
fixed acidity           0
volatile acidity        0
citric acid             0
residual sugar          0
chlorides               0
free sulfur dioxide     0
total sulfur dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
quality                 0
dtype: int64
```

#No.of Rows and Columns

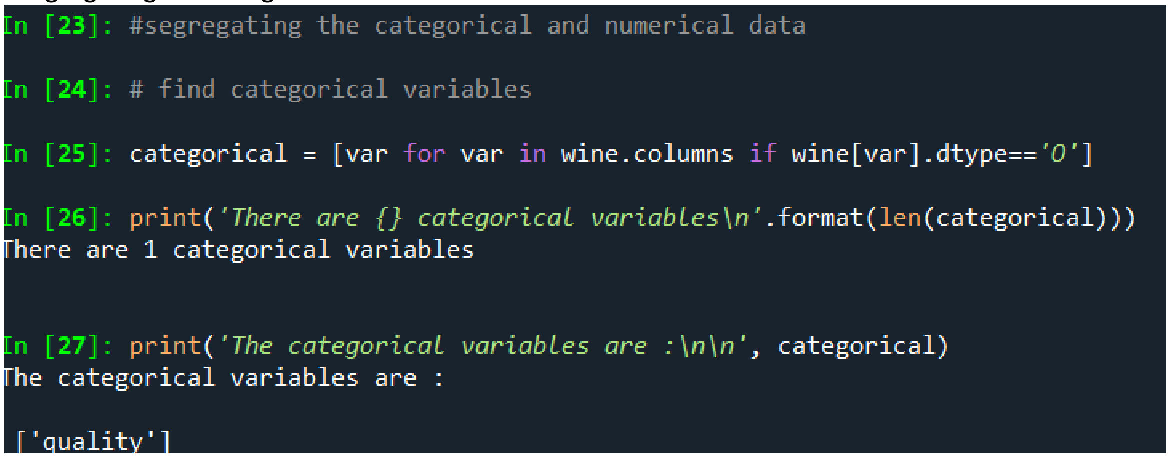


```
In [100]: wine.shape
Out[100]: (1599, 12)
```

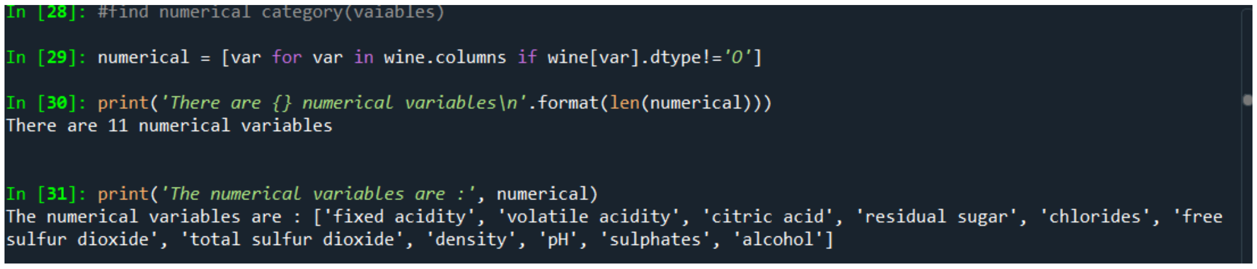
#Visualizing the data
sns.pairplot(wine)



#segregating the categorical and numerical data

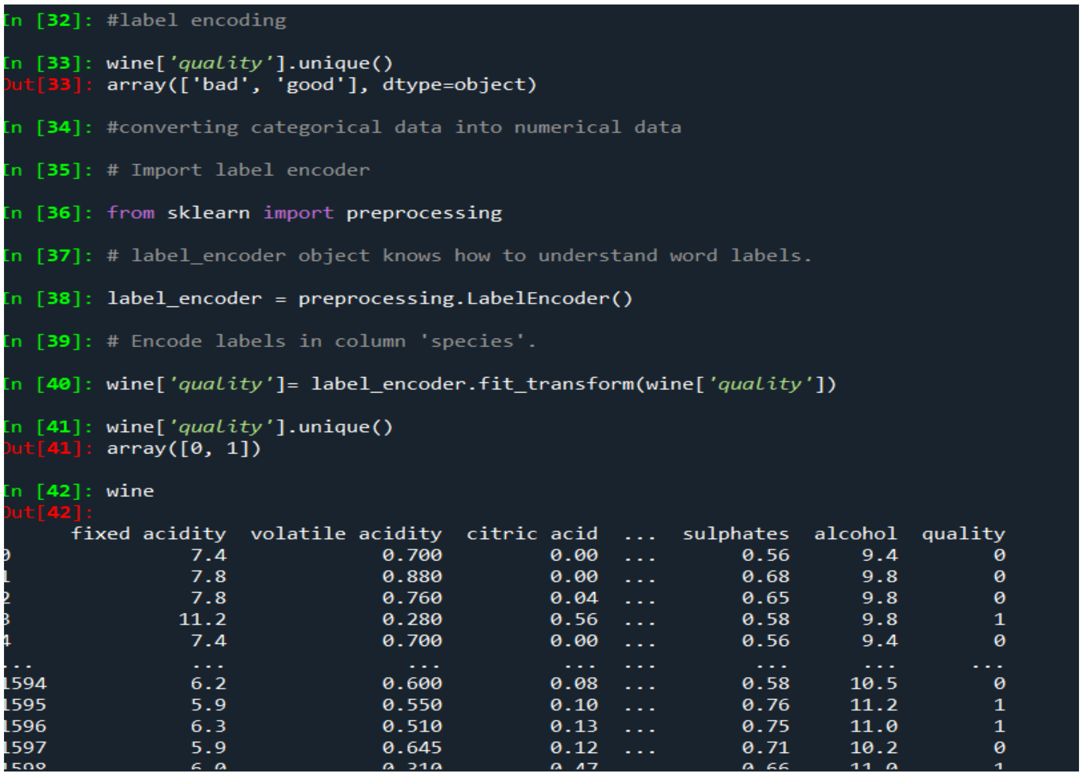


```
In [23]: #segregating the categorical and numerical data

In [24]: # find categorical variables

In [25]: categorical = [var for var in wine.columns if wine[var].dtype=='O']

In [26]: print('There are {} categorical variables\n'.format(len(categorical)))
There are 1 categorical variables


In [27]: print('The categorical variables are :\n\n', categorical)
The categorical variables are :

 ['quality']
```



```
In [28]: #find numerical category(vaiables)

In [29]: numerical = [var for var in wine.columns if wine[var].dtype!='O']

In [30]: print('There are {} numerical variables\n'.format(len(numerical)))
There are 11 numerical variables


In [31]: print('The numerical variables are :', numerical)
The numerical variables are : ['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar', 'chlorides', 'free
sulfur dioxide', 'total sulfur dioxide', 'density', 'pH', 'sulphates', 'alcohol']
```

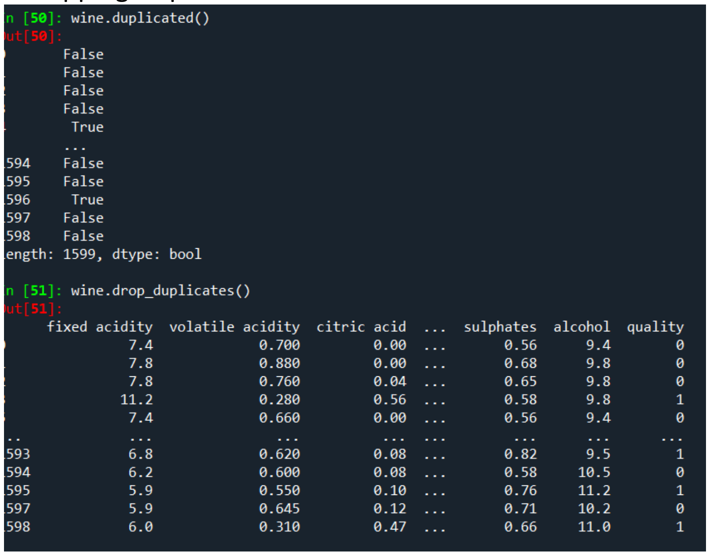
#Converting Categorical data into Numerical data



```
In [32]: #label encoding

In [33]: wine['quality'].unique()
Out[33]: array(['bad', 'good'], dtype=object)

In [34]: #converting categorical data into numerical data

In [35]: # Import label encoder

In [36]: from sklearn import preprocessing

In [37]: # label_encoder object knows how to understand word labels.

In [38]: label_encoder = preprocessing.LabelEncoder()

In [39]: # Encode labels in column 'species'.

In [40]: wine['quality']= label_encoder.fit_transform(wine['quality'])

In [41]: wine['quality'].unique()
Out[41]: array([0, 1])

In [42]: wine
Out[42]:
      fixed acidity  volatile acidity  citric acid  ...  sulphates  alcohol  quality
0               7.4             0.700         0.00  ...       0.56      9.4        0
1               7.8             0.880         0.00  ...       0.68      9.8        0
2               7.8             0.760         0.04  ...       0.65      9.8        0
3              11.2             0.280         0.56  ...       0.58      9.8        1
4               7.4             0.700         0.00  ...       0.56      9.4        0
...             ...               ...          ...  ...        ...      ...      ...
1594            6.2             0.600         0.08  ...       0.58     10.5        0
1595            5.9             0.550         0.10  ...       0.76     11.2        1
1596            6.3             0.510         0.13  ...       0.75     11.0        1
1597            5.9             0.645         0.12  ...       0.71     10.2        0
```

In Quality column categorical data converted into numerical data.

#Dropping Duplicates

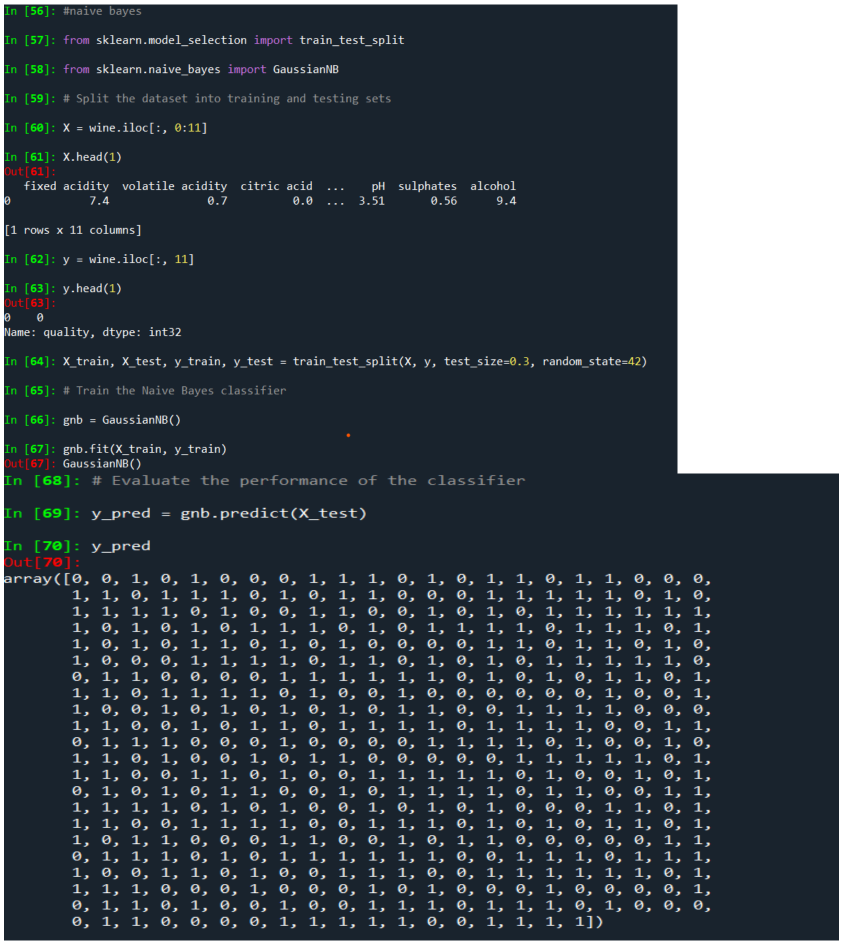


```
n [50]: wine.duplicated()
ut[50]:
       False
       False
       False
       False
        True
       ...
594    False
595    False
596     True
597    False
598    False
ength: 1599, dtype: bool

n [51]: wine.drop_duplicates()
ut[51]:
     fixed acidity  volatile acidity  citric acid  ...  sulphates  alcohol  quality
               7.4             0.700         0.00  ...       0.56      9.4        0
               7.8             0.880         0.00  ...       0.68      9.8        0
               7.8             0.760         0.04  ...       0.65      9.8        0
              11.2             0.280         0.56  ...       0.58      9.8        1
               7.4             0.660         0.00  ...       0.56      9.4        0
..             ...               ...          ...  ...        ...      ...      ...
593            6.8             0.620         0.08  ...       0.82      9.5        1
594            6.2             0.600         0.08  ...       0.58     10.5        0
595            5.9             0.550         0.10  ...       0.76     11.2        1
597            5.9             0.645         0.12  ...       0.71     10.2        0
598            6.0             0.310         0.47  ...       0.66     11.0        1
```

There are 240 duplicates in dataset, Removing Duplicates from data

#Implementing Naïve Bayes:

After reading the data, creating the feature vectors X and target vector y and splitting the dataset into a training set (X_train, y_train) and a test set (X_test, y_test), we use MultinomialMB of sklearn to implement the Naive Bayes algorithm.



```
In [56]: #naive bayes

In [57]: from sklearn.model_selection import train_test_split

In [58]: from sklearn.naive_bayes import GaussianNB

In [59]: # Split the dataset into training and testing sets

In [60]: X = wine.iloc[:, 0:11]

In [61]: X.head(1)
Out[61]:
   fixed acidity  volatile acidity  citric acid  ...    pH  sulphates  alcohol
0            7.4               0.7          0.0  ...  3.51       0.56      9.4

[1 rows x 11 columns]

In [62]: y = wine.iloc[:, 11]

In [63]: y.head(1)
Out[63]:
0    0
Name: quality, dtype: int32

In [64]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

In [65]: # Train the Naive Bayes classifier

In [66]: gnb = GaussianNB()

In [67]: gnb.fit(X_train, y_train)
Out[67]: GaussianNB()
In [68]: # Evaluate the performance of the classifier

In [69]: y_pred = gnb.predict(X_test)

In [70]: y_pred
Out[70]:
array([0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0,
       1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0,
       1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1,
       1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1,
       1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0,
       1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0,
       0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1,
       1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1,
       1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0,
       1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1,
       0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0,
       1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1,
       1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1,
       0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1,
       1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1,
       1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1,
       1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1,
       0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1,
       1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1,
       1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1,
       0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0,
       0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1])
```

We store the predicted outputs in y_pred, which we will use for the several metrics below.

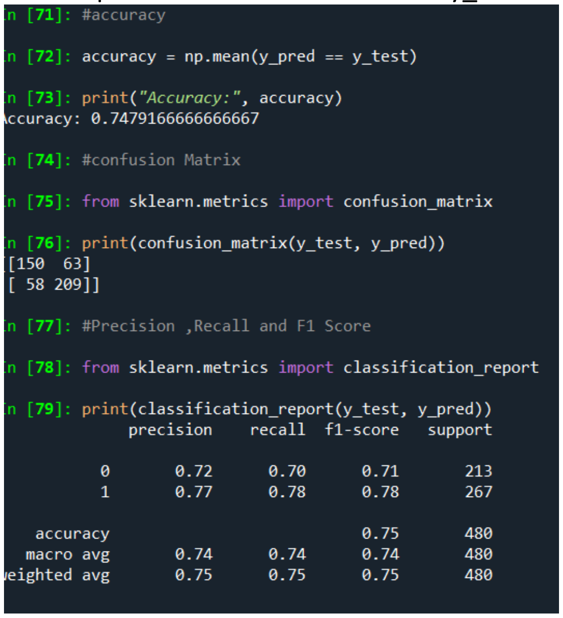
Accuracy

The first metric we are going to discuss is, perhaps, the simplest one, the accuracy. It answers the question: "How often is the classifier correct?"

It can be obtained simply using the following formulae:

Accuracy= correctly classified items/all classified items

sklearn provides the function accuracy_score to obtain the accuracy:



```
n [71]: #accuracy

n [72]: accuracy = np.mean(y_pred == y_test)

n [73]: print("Accuracy:", accuracy)
ccuracy: 0.7479166666666667

n [74]: #confusion Matrix

n [75]: from sklearn.metrics import confusion_matrix

n [76]: print(confusion_matrix(y_test, y_pred))
[[150  63]
 [ 58 209]]

n [77]: #Precision ,Recall and F1 Score

n [78]: from sklearn.metrics import classification_report

n [79]: print(classification_report(y_test, y_pred))
              precision    recall  f1-score   support

           0       0.72      0.70      0.71       213
           1       0.77      0.78      0.78       267

    accuracy                           0.75       480
   macro avg       0.74      0.74      0.74       480
eighted avg       0.75      0.75      0.75       480
```

The output for the Naive Bayes algorithm is: 0.74

The wine quality Is 74% out of 100%. 74 are classified good or bad.

Confusion matrix

The **confusion matrix** is another metric that is often used to measure the performance of a classification algorithm. True to its name, the terminology related to the confusion matrix can be rather confusing, but the matrix itself is simple to understand (unlike the movies).

	Predicted : Good wine	Predicted : Bad wine
Actual: good	True positive	False negative
Actual: bad	False positive	True negative

The predicted classes are represented in the columns of the matrix, whereas the actual classes are in the rows of the matrix. We then have four cases:

- True positives (TP): the cases for which the classifier predicted 'bad' and the wine is actually bad.
- True negatives (TN): the cases for which the classifier predicted 'not bad' and the wine were actually real.
- False positives (FP): the cases for which the classifier predicted 'bad' but the wine is actually real.
- False negatives (FN): the cases for which the classifier predicted 'not bad' but the wine is actually bad.

Out of the 213 actual instances of 'not bad' (first row), the classifier predicted correctly 150 of them.

Out of the 267 actual instances of 'bad' (second row), the classifier predicted correctly 209 of them.

Out of all 480 wine quality's, the classifier predicted correctly 860 of them.

This last comment allows us to obtain the accuracy from the confusion matrix, by applying the

following formulae:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

<u>Precision, recall and f1-score</u>

Besides the accuracy, there are several other performance measures which can be computed from the confusion matrix. Some of the main ones are obtained using the function classification report

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}:$$

Precision is usually used when the goal is to *limit the number of false positives*

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

Recall is usually used when the goal is to *limit the number of false negatives* (FN)

$$f_1\text{-score} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

It is useful when you need to take both precision and recall into account. If you try to only optimize recall, your algorithm will predict most examples to belong to the positive class, but that will result in many false positives and, hence, low precision.

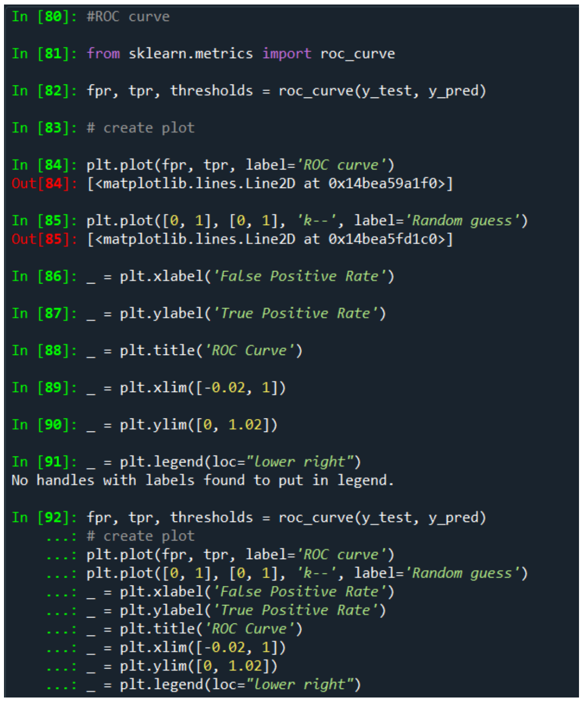
ROC curve

A more visual way to measure the performance of a binary classifier is the receiver operating characteristic (ROC) curve. It is created by plotting the true positive rate (TPR) (or recall) against the false positive rate (FPR), which we haven't defined explicitly yet:

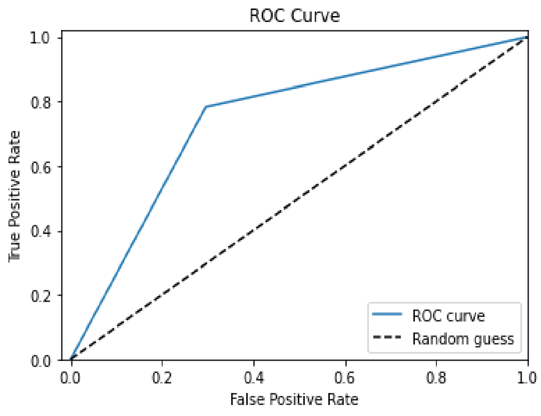
$$\text{FP rate} = \frac{\text{FP}}{\text{FP} + \text{TN}}.$$

Naive Bayes algorithm isn't only able to predict if each wine quality is bad or not, but it can also give us the **predicted probability** for such event.

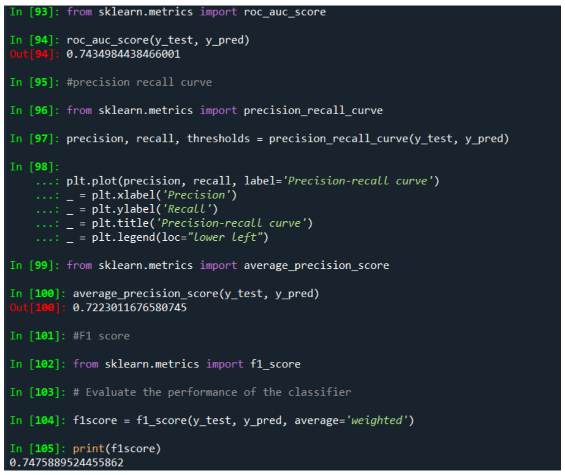
From ROC curve we can visualize how the performance of the classifier changes as we vary the threshold.
First, let's plot the ROC curve for the case at hand by importing roc_curve from sklearn.metrics, which gives us the TP and FP rates:



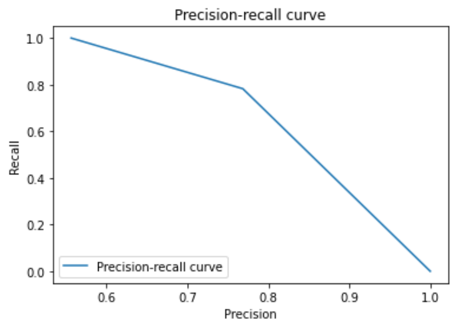
```
In [80]: #ROC curve

In [81]: from sklearn.metrics import roc_curve

In [82]: fpr, tpr, thresholds = roc_curve(y_test, y_pred)

In [83]: # create plot

In [84]: plt.plot(fpr, tpr, label='ROC curve')
Out[84]: [<matplotlib.lines.Line2D at 0x14bea59a1f0>]

In [85]: plt.plot([0, 1], [0, 1], 'k--', label='Random guess')
Out[85]: [<matplotlib.lines.Line2D at 0x14bea5fd1c0>]

In [86]: _ = plt.xlabel('False Positive Rate')

In [87]: _ = plt.ylabel('True Positive Rate')

In [88]: _ = plt.title('ROC Curve')

In [89]: _ = plt.xlim([-0.02, 1])

In [90]: _ = plt.ylim([0, 1.02])

In [91]: _ = plt.legend(loc="lower right")
No handles with labels found to put in legend.

In [92]: fpr, tpr, thresholds = roc_curve(y_test, y_pred)
    ...: # create plot
    ...: plt.plot(fpr, tpr, label='ROC curve')
    ...: plt.plot([0, 1], [0, 1], 'k--', label='Random guess')
    ...: _ = plt.xlabel('False Positive Rate')
    ...: _ = plt.ylabel('True Positive Rate')
    ...: _ = plt.title('ROC Curve')
    ...: _ = plt.xlim([-0.02, 1])
    ...: _ = plt.ylim([0, 1.02])
    ...: _ = plt.legend(loc="lower right")
```



The above points suggest that the *area* under the ROC curve (usually denoted by **AUC**) is a bettermeasure of the performance of the classification algorithm. If it is near 0.5, the classifier is not much better than random guessing, whereas it gets better as the area gets close to 1.

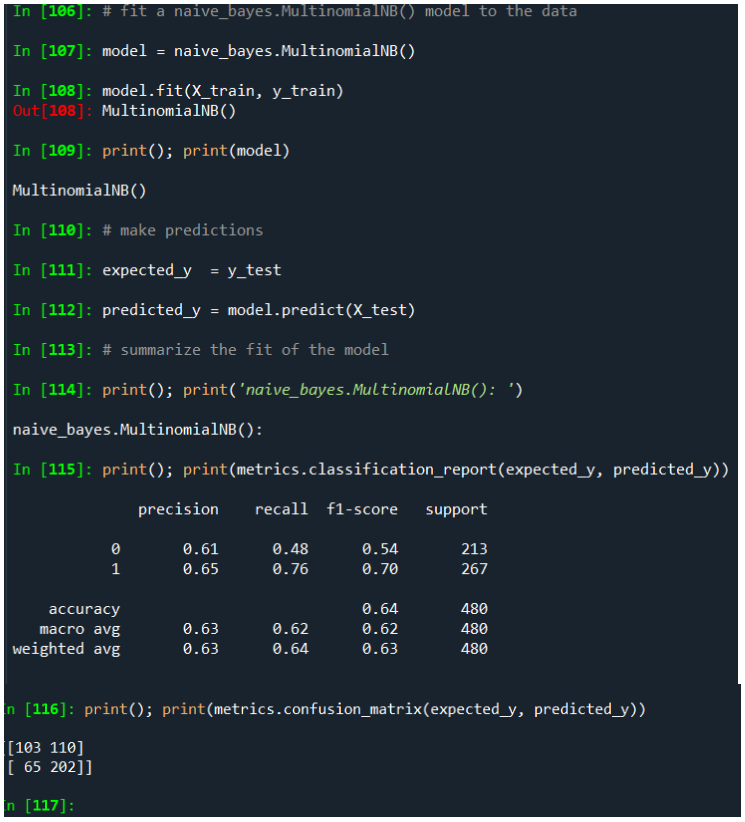


```
In [93]: from sklearn.metrics import roc_auc_score

In [94]: roc_auc_score(y_test, y_pred)
Out[94]: 0.7434984438466001

In [95]: #precision recall curve

In [96]: from sklearn.metrics import precision_recall_curve

In [97]: precision, recall, thresholds = precision_recall_curve(y_test, y_pred)

In [98]:
    ...: plt.plot(precision, recall, label='Precision-recall curve')
    ...: _ = plt.xlabel('Precision')
    ...: _ = plt.ylabel('Recall')
    ...: _ = plt.title('Precision-recall curve')
    ...: _ = plt.legend(loc="lower left")

In [99]: from sklearn.metrics import average_precision_score

In [100]: average_precision_score(y_test, y_pred)
Out[100]: 0.7223011676580745

In [101]: #F1 score

In [102]: from sklearn.metrics import f1_score

In [103]: # Evaluate the performance of the classifier

In [104]: f1score = f1_score(y_test, y_pred, average='weighted')

In [105]: print(f1score)
0.7475889524455862
```



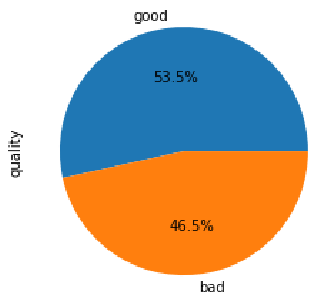
We can obtain the AUC by importing roc_auc_score from sklearn.metrics

The AUC is indeed quite close to 1, and so our classifier is better at minimizing false negatives (bad which is classified as real) and true negatives (good quality which is classified as real).
The output of our classification is 72%

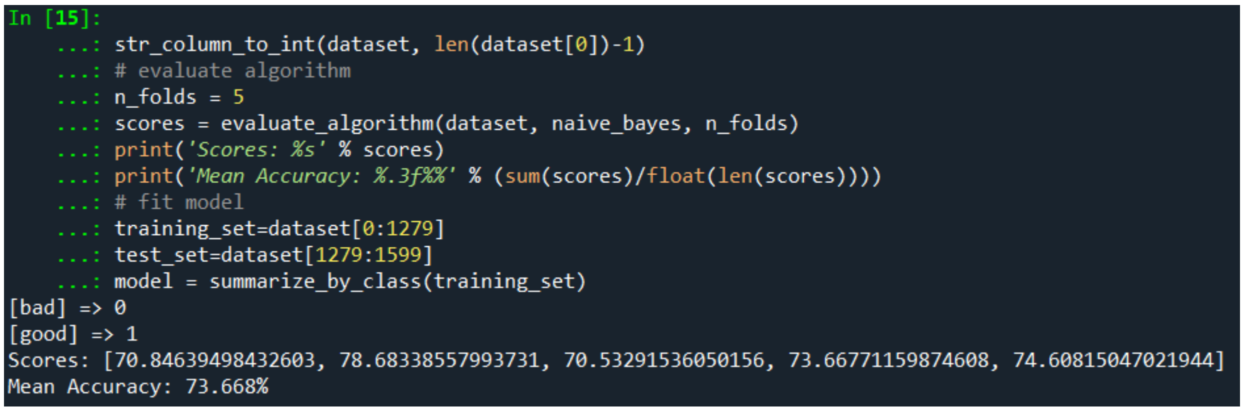


```
In [106]: # fit a naive_bayes.MultinomialNB() model to the data

In [107]: model = naive_bayes.MultinomialNB()

In [108]: model.fit(X_train, y_train)
Out[108]: MultinomialNB()

In [109]: print(); print(model)

MultinomialNB()

In [110]: # make predictions

In [111]: expected_y  = y_test

In [112]: predicted_y = model.predict(X_test)

In [113]: # summarize the fit of the model

In [114]: print(); print('naive_bayes.MultinomialNB(): ')

naive_bayes.MultinomialNB():

In [115]: print(); print(metrics.classification_report(expected_y, predicted_y))

              precision    recall  f1-score   support

           0       0.61      0.48      0.54       213
           1       0.65      0.76      0.70       267

    accuracy                           0.64       480
   macro avg       0.63      0.62      0.62       480
weighted avg       0.63      0.64      0.63       480

n [116]: print(); print(metrics.confusion_matrix(expected_y, predicted_y))

[103 110]
[ 65 202]]

n [117]:
```

#implementing naïve bayes without Library

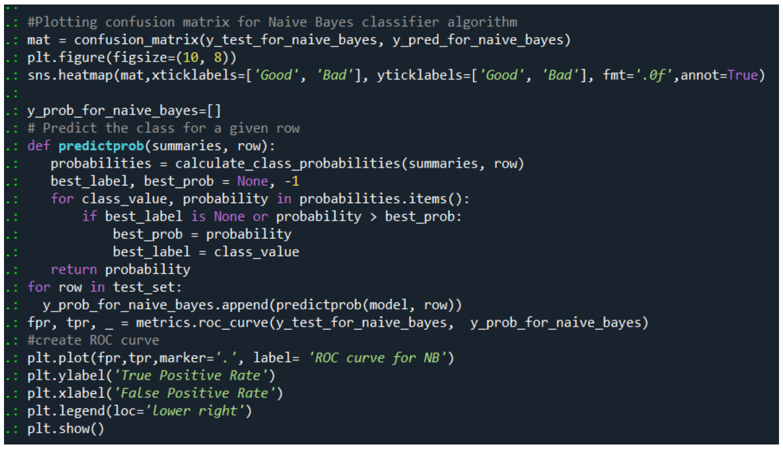


Percentage of good and bad quality wine

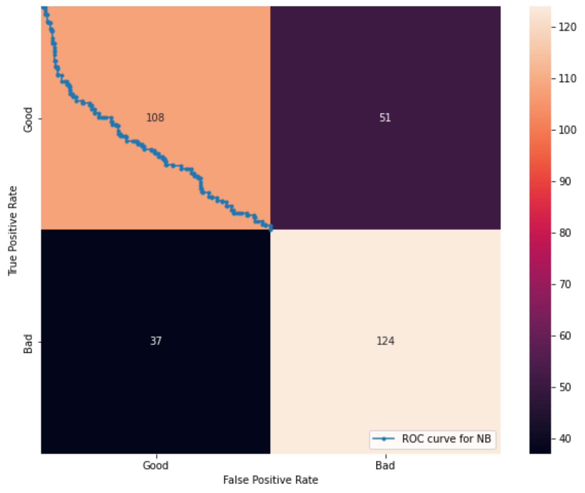


```
In [15]:
    ...: str_column_to_int(dataset, len(dataset[0])-1)
    ...: # evaluate algorithm
    ...: n_folds = 5
    ...: scores = evaluate_algorithm(dataset, naive_bayes, n_folds)
    ...: print('Scores: %s' % scores)
    ...: print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
    ...: # fit model
    ...: training_set=dataset[0:1279]
    ...: test_set=dataset[1279:1599]
    ...: model = summarize_by_class(training_set)
[bad] => 0
[good] => 1
Scores: [70.84639498432603, 78.68338557993731, 70.53291536050156, 73.66771159874608, 74.60815047021944]
Mean Accuracy: 73.668%
```

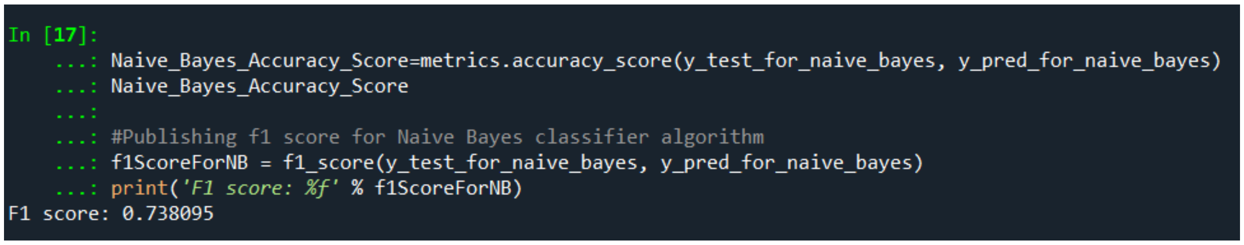
#confusion matrix



```
.:
.: #Plotting confusion matrix for Naive Bayes classifier algorithm
.: mat = confusion_matrix(y_test_for_naive_bayes, y_pred_for_naive_bayes)
.: plt.figure(figsize=(10, 8))
.: sns.heatmap(mat,xticklabels=['Good', 'Bad'], yticklabels=['Good', 'Bad'], fmt='.0f',annot=True)
.:
.: y_prob_for_naive_bayes=[]
.: # Predict the class for a given row
.: def predictprob(summaries, row):
.:     probabilities = calculate_class_probabilities(summaries, row)
.:     best_label, best_prob = None, -1
.:     for class_value, probability in probabilities.items():
.:         if best_label is None or probability > best_prob:
.:             best_prob = probability
.:             best_label = class_value
.:     return probability
.: for row in test_set:
.:   y_prob_for_naive_bayes.append(predictprob(model, row))
.: fpr, tpr, _ = metrics.roc_curve(y_test_for_naive_bayes,  y_prob_for_naive_bayes)
.: #create ROC curve
.: plt.plot(fpr,tpr,marker='.', label= 'ROC curve for NB')
.: plt.ylabel('True Positive Rate')
.: plt.xlabel('False Positive Rate')
.: plt.legend(loc='lower right')
.: plt.show()
```



#F1 score



```
In [17]:
    ...: Naive_Bayes_Accuracy_Score=metrics.accuracy_score(y_test_for_naive_bayes, y_pred_for_naive_bayes)
    ...: Naive_Bayes_Accuracy_Score
    ...:
    ...: #Publishing f1 score for Naive Bayes classifier algorithm
    ...: f1ScoreForNB = f1_score(y_test_for_naive_bayes, y_pred_for_naive_bayes)
    ...: print('F1 score: %f' % f1ScoreForNB)
F1 score: 0.738095
```

I Got the similar score for naïve baye’s algorithm with library’s and without library’s

Implement Logistic regression

Logistic regression is an example of supervised learning. It is used to calculate or predict the probability of a binary (yes/no) event occurring.

$$\hat{Y} = \frac{1}{1 + e^{-Z}} \qquad Z = w.X + b$$

Y_hat --> predicted value

X --> Input Variable

w --> weight

b --> bias

Gradient Descent:

Gradient Descent is an optimization algorithm used for minimizing the loss function in various machine learning algorithms. It is used for updating the parameters of the learning model.

w = w - α*dw

b = b - α*db

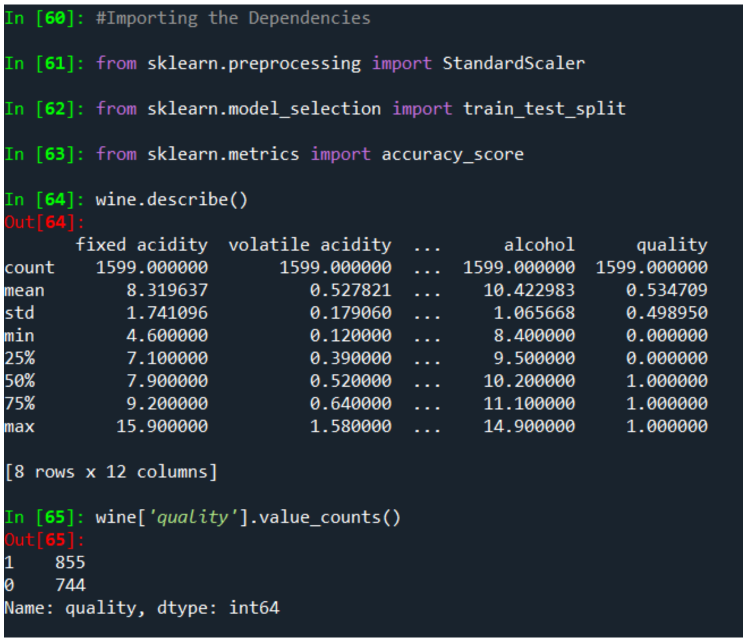
Learning Rate:

Learning rate is a tuning parameter in an optimization algorithm that determines the step size at each iteration while moving toward a minimum of a loss function.

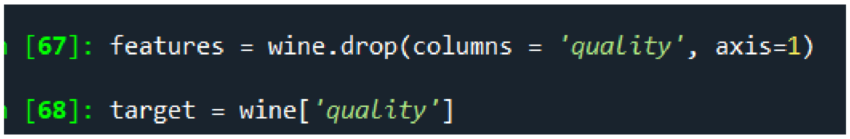
Derivatives:

$$dw = \frac{1}{m} * (\hat{Y} - Y).X$$

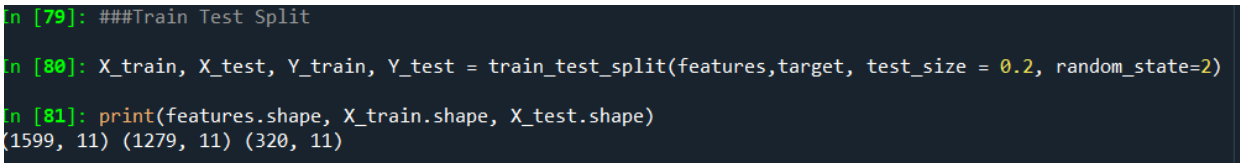
$$db = \frac{1}{m} * (\hat{Y} - Y)$$


```
In [60]: #Importing the Dependencies

In [61]: from sklearn.preprocessing import StandardScaler

In [62]: from sklearn.model_selection import train_test_split

In [63]: from sklearn.metrics import accuracy_score

In [64]: wine.describe()
Out[64]:
       fixed acidity  volatile acidity  ...      alcohol      quality
count    1599.000000       1599.000000  ...  1599.000000  1599.000000
mean        8.319637          0.527821  ...    10.422983     0.534709
std         1.741096          0.179060  ...     1.065668     0.498950
min         4.600000          0.120000  ...     8.400000     0.000000
25%         7.100000          0.390000  ...     9.500000     0.000000
50%         7.900000          0.520000  ...    10.200000     1.000000
75%         9.200000          0.640000  ...    11.100000     1.000000
max        15.900000          1.580000  ...    14.900000     1.000000

[8 rows x 12 columns]

In [65]: wine['quality'].value_counts()
Out[65]:
1    855
0    744
Name: quality, dtype: int64
```

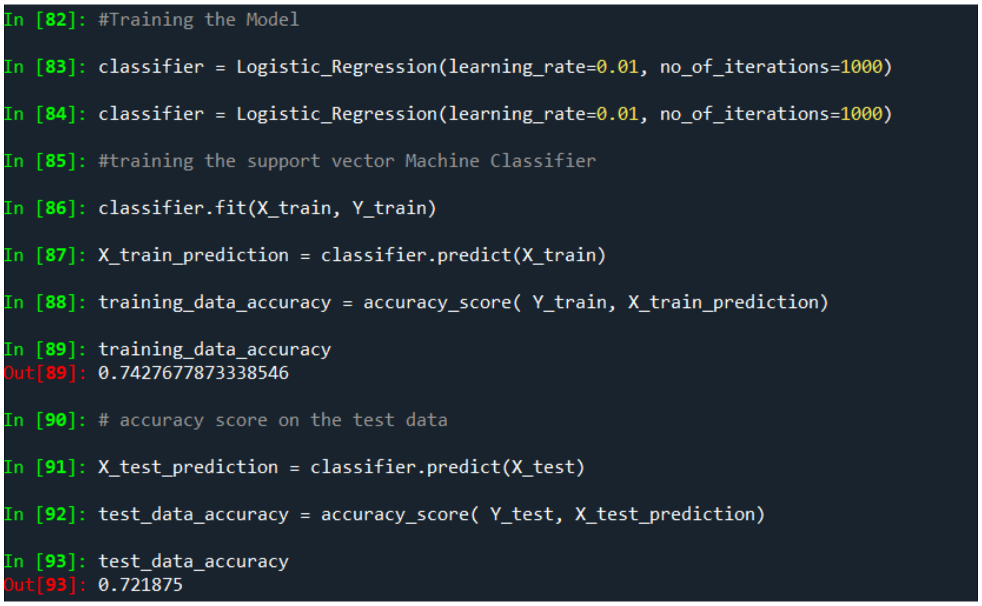
1 represents good
0 represents bad


```
 [67]: features = wine.drop(columns = 'quality', axis=1)

 [68]: target = wine['quality']
```

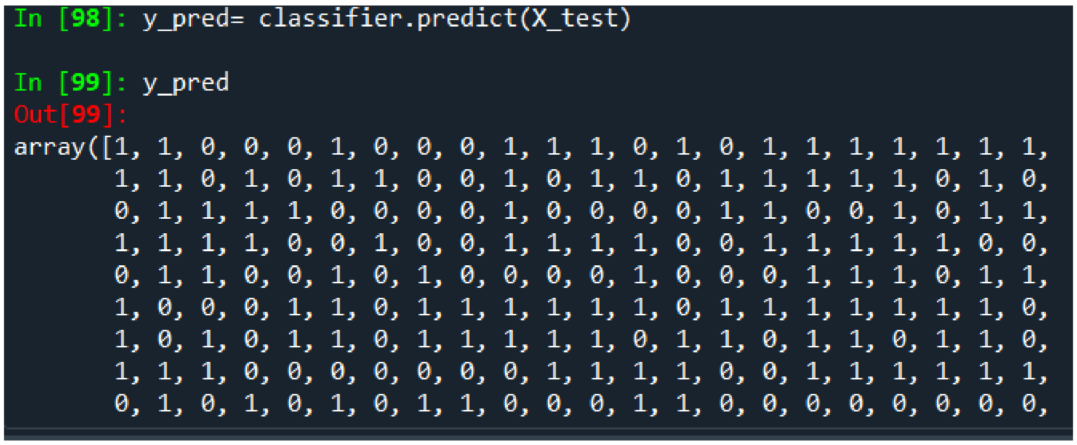
Selecting dependent and independent variables for train and test the data.


```
n [79]: ###Train Test Split

n [80]: X_train, X_test, Y_train, Y_test = train_test_split(features,target, test_size = 0.2, random_state=2)

n [81]: print(features.shape, X_train.shape, X_test.shape)
(1599, 11) (1279, 11) (320, 11)
```

Here considered 80%data for training and 20%for testing

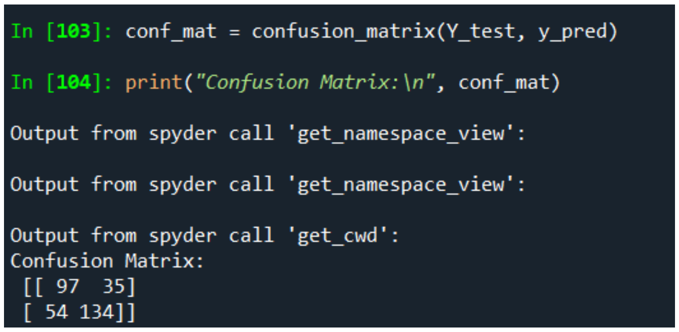

```
In [82]: #Training the Model

In [83]: classifier = Logistic_Regression(learning_rate=0.01, no_of_iterations=1000)

In [84]: classifier = Logistic_Regression(learning_rate=0.01, no_of_iterations=1000)

In [85]: #training the support vector Machine Classifier

In [86]: classifier.fit(X_train, Y_train)

In [87]: X_train_prediction = classifier.predict(X_train)

In [88]: training_data_accuracy = accuracy_score( Y_train, X_train_prediction)

In [89]: training_data_accuracy
Out[89]: 0.7427677873338546

In [90]: # accuracy score on the test data

In [91]: X_test_prediction = classifier.predict(X_test)

In [92]: test_data_accuracy = accuracy_score( Y_test, X_test_prediction)

In [93]: test_data_accuracy
Out[93]: 0.721875
```

The Accuracy for Logistic regression on wine quality is 74%



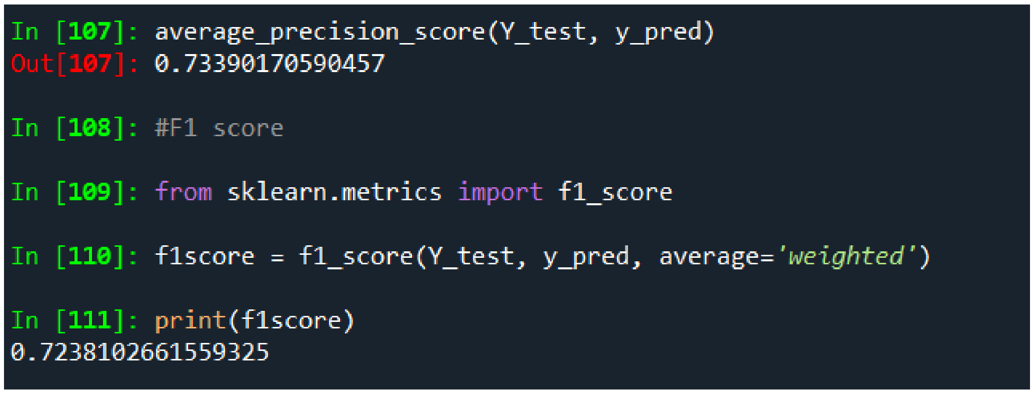
```
In [98]: y_pred= classifier.predict(X_test)

In [99]: y_pred
Out[99]:
array([1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0,
       0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1,
       1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0,
       0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1,
       1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0,
       1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0,
       1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
       0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,
```

Predicted results for x_test resulting quality of wine



```
In [103]: conf_mat = confusion_matrix(Y_test, y_pred)

In [104]: print("Confusion Matrix:\n", conf_mat)

Output from spyder call 'get_namespace_view':

Output from spyder call 'get_namespace_view':

Output from spyder call 'get_cwd':
Confusion Matrix:
 [[ 97  35]
 [ 54 134]]
```

Precision score and F1 Score



```
In [107]: average_precision_score(Y_test, y_pred)
Out[107]: 0.73390170590457

In [108]: #F1 score

In [109]: from sklearn.metrics import f1_score

In [110]: f1score = f1_score(Y_test, y_pred, average='weighted')

In [111]: print(f1score)
0.7238102661559325
```

Comparing the results of Naive Bayes Classifier and Logistic Regression for wine data, we can see that both methods achieve a high accuracy score of above 70% and a good F1 score of above 0.74.

However, when we look at the confusion matrix, we can see that Logistic Regression has a higher true positive rate and a lower false positive rate than Naive Bayes Classifier. This means that Logistic Regression is better at correctly identifying the positive class (good quality wines) and has a lower chance of wrongly identifying a wine as good quality when it is actually poor quality.

In terms of the ROC curve, we can see that Logistic Regression has a higher AUC (Area Under the Curve) score, which again indicates better performance at correctly classifying the positive class.

Furthermore, when we look at the results of cross-validation, we can see that Logistic Regression has a slightly higher average score than Naive Bayes Classifier, indicating better overall performance.

Therefore, we can conclude that Logistic Regression performs better than Naive Bayes Classifier for the wine quality dataset, as it achieves better performance in terms of correctly identifying the positive class and has a higher AUC score.

Q2. Regression Task:

Q1> Regression Task: Predict the salary based on years of experience
Use this dataset : Salary dataset
(a) Implement Maximum likelihood estimator for regression task.
(b) Explain all the steps in details
(c) Report error for both training and testing set.

Maximum likelihood estimation (MLE) is a technique used to estimate the parameters of a statistical model. In the case of regression analysis, MLE is used to estimate the coefficients of the linear regression model.

For the Salary dataset, we can use MLE to estimate the coefficients of a linear regression model that predicts an individual's salary based on their years of experience. The linear regression model can be represented as follows:

$Salary = \beta_0 + \beta_1 * Experience + \varepsilon$

Where β_0 is the intercept, β_1 is the coefficient of Experience, and ε is the error term.

The MLE estimates of the coefficients β_0 and β_1 can be obtained by maximizing the likelihood function $L(\beta_0, \beta_1|data)$, which is the joint probability of observing the data given the model parameters. The likelihood function can be expressed as follows:

$L(\beta_0, \beta_1|data) = \prod_{i=1}^{n} [1 / (\sigma * \sqrt{2\pi})] * \exp[-(y_i - \beta_0 - \beta_1 * x_i)^2 / (2 * \sigma^2)]$

Where n is the number of observations in the dataset, y_i and x_i are the salary and experience values for the ith observation, σ is the standard deviation of the error term ε, and σ^2 is the variance of the error term.

Maximizing the likelihood function involves finding the values of β_0 and β_1 that maximize the probability of observing the data given the model. This can be done using numerical optimization techniques such as gradient descent or the Newton-Raphson method.

```
# Importing Libraries and loading data
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn import metrics as sm
import seaborn as sns
from sklearn import metrics
from sklearn import linear_model
```

Data Pre-processing

Loading dataset
```
salary=pd.read_csv("C:\\Users\\Admin\\OneDrive\\Desktop\\Salary_dataset.csv")
salary.head()
#summary of data
salary.info()
#null values
salary.isna().sum()   -----→ found 0 null values and 0 duplicates
#duplicates
salary.duplicated().sum()
#columns
salary.columns
```

#drop unwanted column
salary.drop(["Unnamed: 0"],axis=1,inplace=True) --→ dropped unnamed column
salary.head()

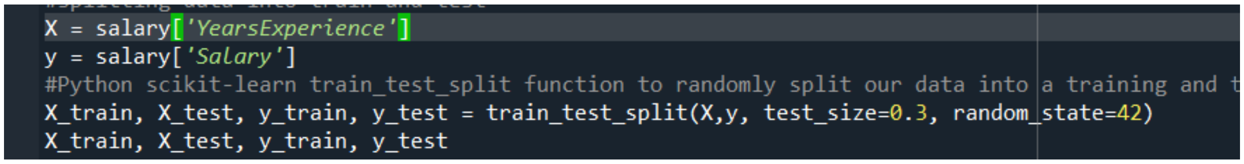
#splitting data into train and test
X = salary['YearsExperience']
y = salary['Salary']

extracting the independent and dependent variables and assigning them to X and y

Splitting dataset into train and test

The training set will be used to train the model whereas the test set will be used to assess the performance of the trained model in predicting the result from unseen data.

###from sklearn.model_selection import train_test_split
Python scikit-learn train_test_split function to randomly split our data into a training and test set.

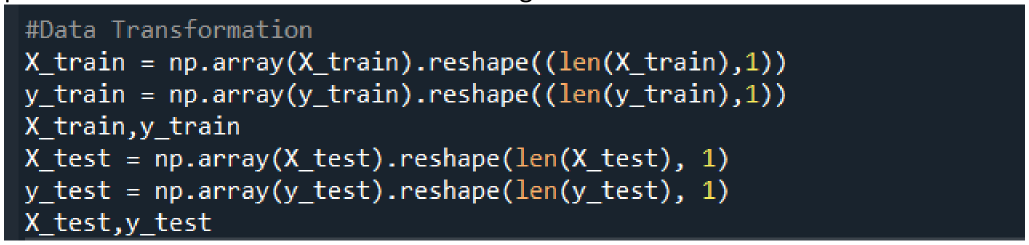


```
X = salary['YearsExperience']
y = salary['Salary']
#Python scikit-learn train_test_split function to randomly split our data into a training and t
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.3, random_state=42)
X_train, X_test, y_train, y_test
```

apart 30% of the entire dataset as the test set and assign the training set and test set into four variables, respectively.

Data Transformation

Python scikit-learn only accepts the training and test data in a 2-dimensional array format. We have to perform data transformation on our training set and test set.



```
#Data Transformation
X_train = np.array(X_train).reshape((len(X_train),1))
y_train = np.array(y_train).reshape((len(y_train),1))
X_train,y_train
X_test = np.array(X_test).reshape(len(X_test), 1)
y_test = np.array(y_test).reshape(len(y_test), 1)
X_test,y_test
```

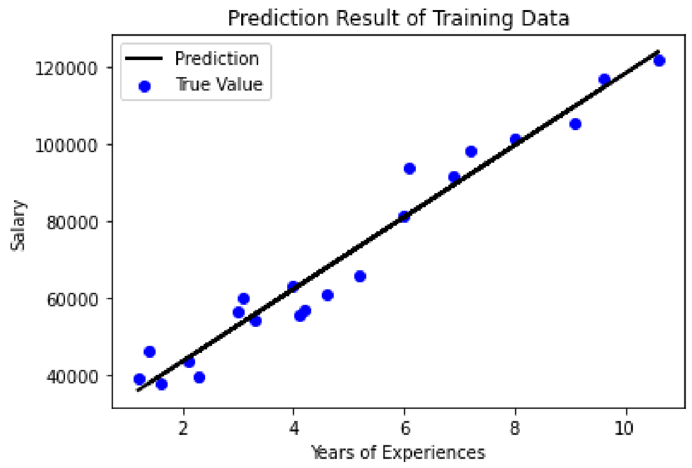
Numpy reshape function to transform the training set from 1-dimensional series to a 2-dimensional array.

Training the model
from sklearn import linear_model
#Training Model
model = linear_model.LinearRegression()
model.fit(X_train, y_train)

Predicting Salary using Linear Model

we have trained a linear model and used it to predict the salary on our training set

```
y_train_pred = model.predict(X_train)
y_train_pred
plt.figure()
plt.scatter(X_train, y_train, color='blue', label="True Value")
plt.plot(X_train, y_train_pred, color='black', linewidth=2, label="Prediction")
plt.xlabel("Years of Experiences")
plt.ylabel("Salary")
plt.title('Prediction Result of Training Data')
plt.legend()
plt.show()
```



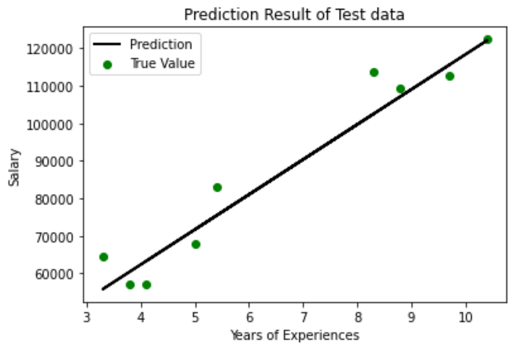
The "true values" are plotted as the blue dots on the chart and the predicted values are plotted as a black color straight line.

the linear model fits well on the training data. This shows a linear relationship between the salary and years of experience.

checking if the linear model can perform well on our test set (unknown data).

```
y_test_pred = model.predict(X_test)
y_test_pred
plt.figure()
plt.scatter(X_test, y_test, color='green', label='True Value')
plt.plot(X_test, y_test_pred, color='black', linewidth=2, label='Prediction')
plt.xlabel("Years of Experiences")
plt.ylabel("Salary")
plt.title('Prediction Result of Test data')
plt.legend()
```

plt.show()



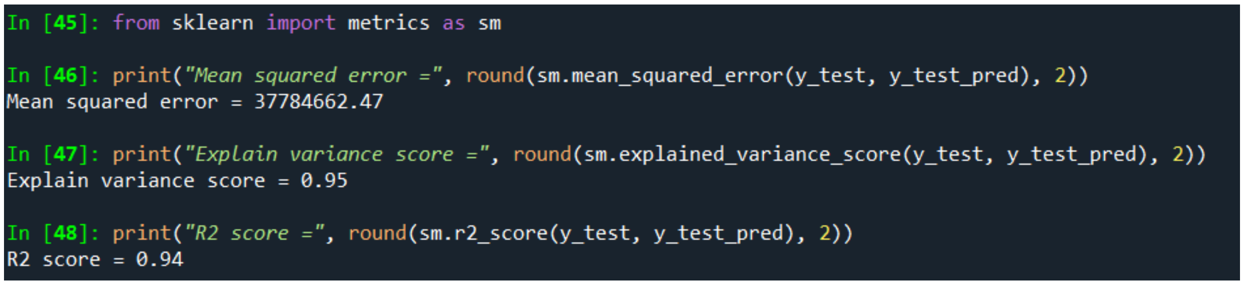
The graph shows that our linear model can fit quite well on the test set. We can observe a linear pattern of how the amount of salary is increased by the years of experience.

Model Evaluation

we will use some quantitative methods to obtain a more precise performance evaluation of our linear model.

We will use three types of quantitative metrics:

- Mean Square Error — The average of the squares of the difference between the true values and the predicted values. The lower the difference the better the performance of the model. This is a common metric used for regression analysis.
- Explained Variance Score — A measurement to examine how well a model can handle the variation of values in the dataset. A score of 1.0 is the perfect score.
- R2 Score — A measurement to examine how well our model can predict values based on the test set (unknown samples). The perfect score is 1.0.


```
In [45]: from sklearn import metrics as sm

In [46]: print("Mean squared error =", round(sm.mean_squared_error(y_test, y_test_pred), 2))
Mean squared error = 37784662.47

In [47]: print("Explain variance score =", round(sm.explained_variance_score(y_test, y_test_pred), 2))
Explain variance score = 0.95

In [48]: print("R2 score =", round(sm.r2_score(y_test, y_test_pred), 2))
R2 score = 0.94
```

If we perform a root square calculation on our mean squared error, we will gain an average discrepancy of around $6146.92 which is quite a low error. Besides, the explain variance score and the R2 score hit 0.9 above. This shows our linear model is not overfitted and can work nicely to predict the salary based on new data.

Conclusion:

Based on our linear model, we can conclude that our salary is grown with our years of working experience and there is a linear relationship between them. We can use our linear model to predict the salary by giving input of years of experience.

###############salary prediction using linear regression without libraries##################

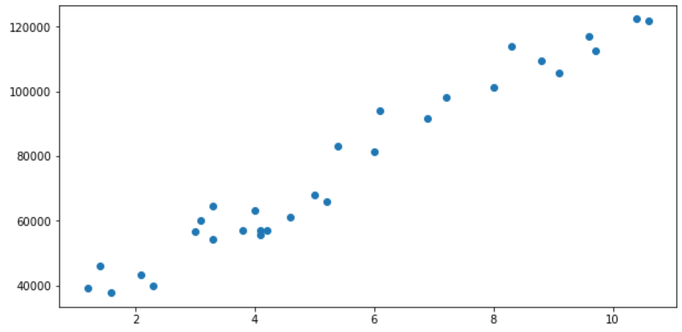
$$\underset{\substack{\uparrow \\ \text{Dependent Variable}}}{Y_i} = \overset{\substack{\text{Constant/Intercept} \\ \downarrow}}{\beta_0} + \underset{\substack{\uparrow \\ \text{Slope/Coefficient}}}{\beta_1} \overset{\substack{\text{Independent Variable} \\ \downarrow}}{X_i}$$

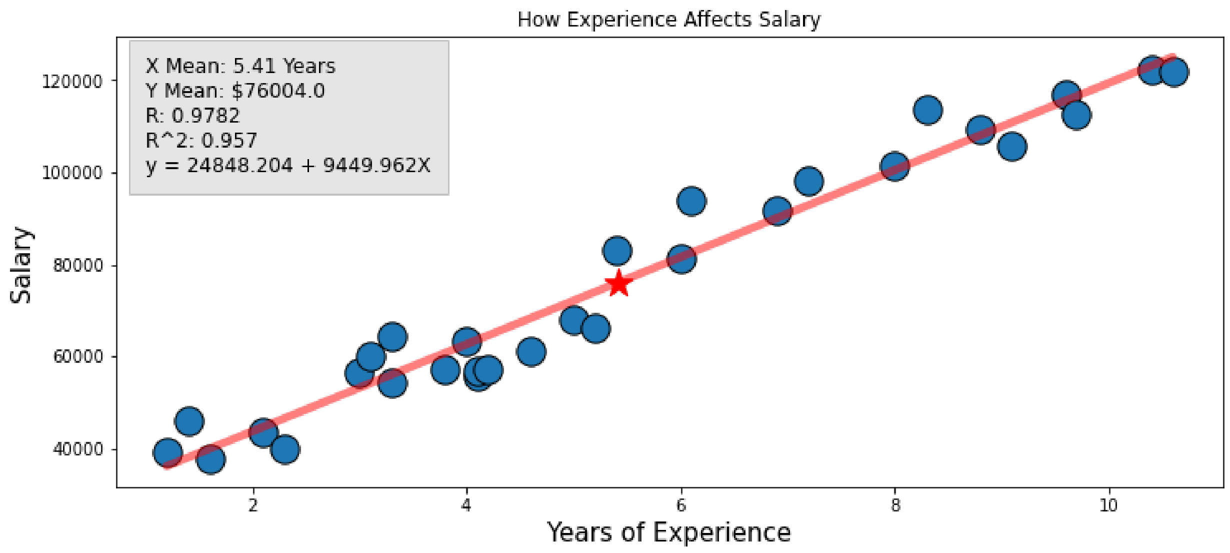
Splitting data into X ,Y variables for test and train the data

X = salary['YearsExperience'].values

Y = salary['Salary'].values

Plotting the X and Y values:





To find b1?

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sum_{i=1}^{n}(X_i - \bar{X})^2}$$

We have to find b_o?

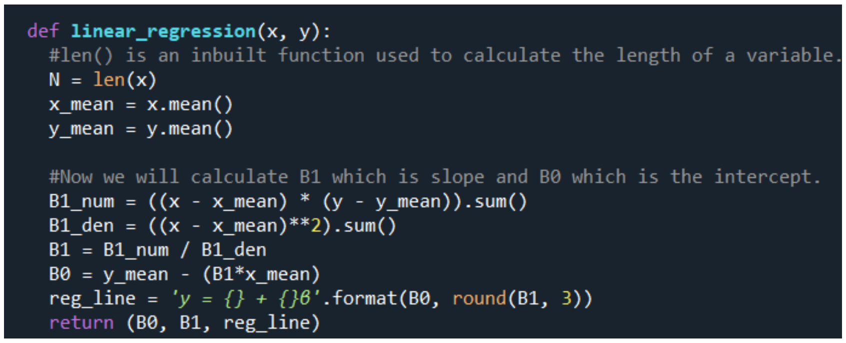
$$\hat{\beta}_0 = \bar{Y} - \hat{\beta}_1 \bar{X}$$ Where

B0: is the intercept

Ý: is Ymean

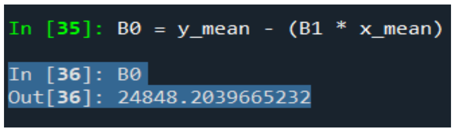
B1: is the slope

乂: is the independent variable



```
def linear_regression(x, y):
    #len() is an inbuilt function used to calculate the length of a variable.
    N = len(x)
    x_mean = x.mean()
    y_mean = y.mean()

    #Now we will calculate B1 which is slope and B0 which is the intercept.
    B1_num = ((x - x_mean) * (y - y_mean)).sum()
    B1_den = ((x - x_mean)**2).sum()
    B1 = B1_num / B1_den
    B0 = y_mean - (B1*x_mean)
    reg_line = 'y = {} + {}β'.format(B0, round(B1, 3))
    return (B0, B1, reg_line)
```

- len() function to get the number of observations in our dataset and set this to the *N* variable. We can then calculate the mean for both *X* and *Y* by simply using the .mean() function.

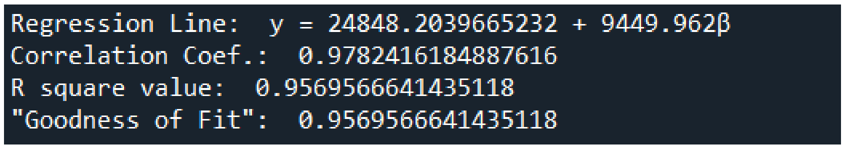


```
In [35]: B0 = y_mean - (B1 * x_mean)

In [36]: B0
Out[36]: 24848.2039665232
```

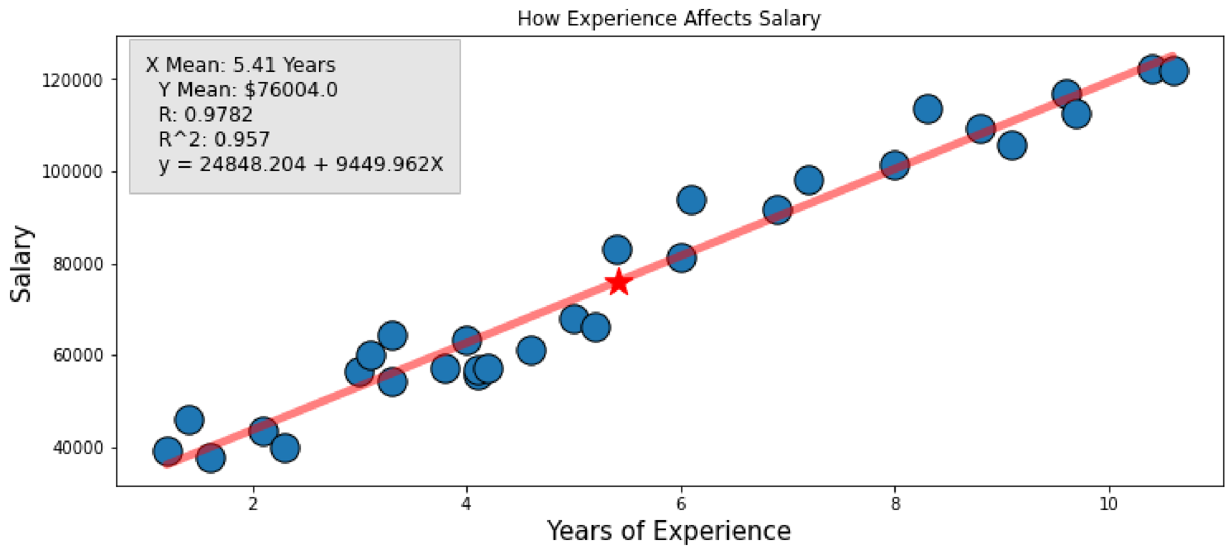
- Linear Regression suggests that the function used for the prediction is a linear function.
- y=mx+c --→equation of a straight line.
- In terms of linear regression, y in this equation stands for the predicted value, x means the independent variable and m & b are the coefficients we need to optimize in order to fit the regression line to our data.

To calculate the coefficient m we will use the formula given below

m = cov(x, y) / var(x)
b = mean(y) — m * mean(x)

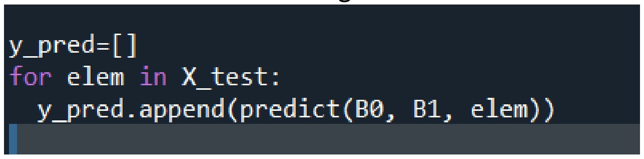


```
Regression Line:  y = 24848.2039665232 + 9449.962β
Correlation Coef.:  0.9782416184887616
R square value:  0.9569566641435118
"Goodness of Fit":  0.9569566641435118
```



we can use our calculations of the regression line to make predictions

#test data in linear regression

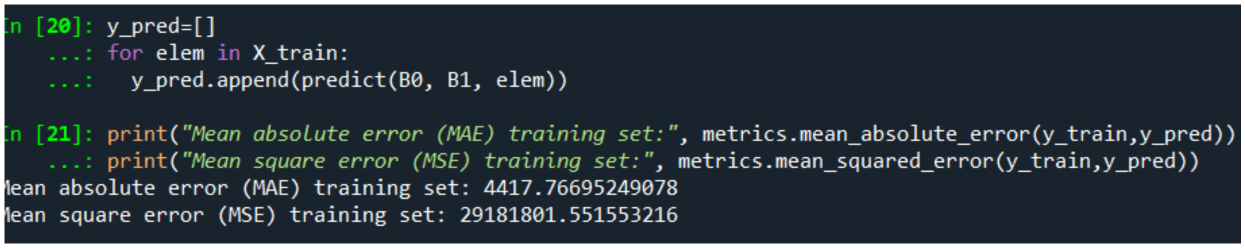


```
y_pred=[]
for elem in X_test:
  y_pred.append(predict(B0, B1, elem))
```

Mean absolute error for test set : 5172.5480756666375

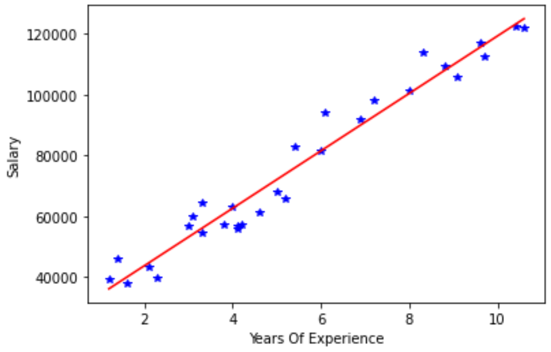
Mean square error for test set: 36145635.453979

#training data in linear regression

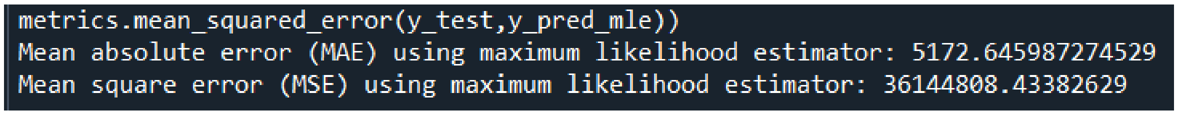


```
n [20]: y_pred=[]
   ...: for elem in X_train:
   ...:   y_pred.append(predict(B0, B1, elem))

n [21]: print("Mean absolute error (MAE) training set:", metrics.mean_absolute_error(y_train,y_pred))
   ...: print("Mean square error (MSE) training set:", metrics.mean_squared_error(y_train,y_pred))
lean absolute error (MAE) training set: 4417.76695249078
lean square error (MSE) training set: 29181801.551553216
```

Plotting MLE (Maximum Likelihood Estimation)

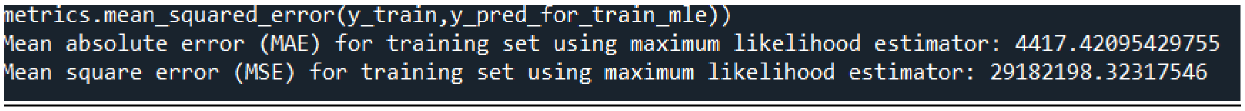


predicted values for test data



```
metrics.mean_squared_error(y_test,y_pred_mle))
Mean absolute error (MAE) using maximum likelihood estimator: 5172.645987274529
Mean square error (MSE) using maximum likelihood estimator: 36144808.43382629
```

predicted values for training data



```
metrics.mean_squared_error(y_train,y_pred_for_train_mle))
Mean absolute error (MAE) for training set using maximum likelihood estimator: 4417.42095429755
Mean square error (MSE) for training set using maximum likelihood estimator: 29182198.32317546
```