Vibhor Agarwal

2020349

ML Assignment 1

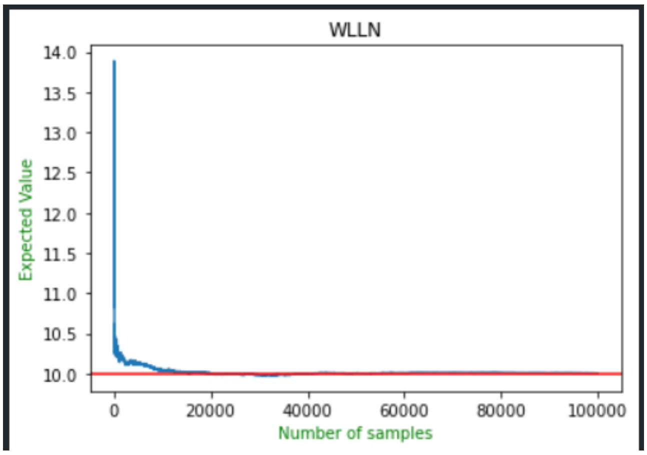
Section A

Pseudocode for Weak Law of Large Numbers

```
count = 1
sum = 0
x = []
y=[]
while count <= 100000:
    x.append(count)
    sum += random.gauss(10,4)
    y.append(sum/count)
    count += 1
plt.plot(x,y)
plt.xlabel('Number of samples',color='green')
plt.ylabel('Expected Value',color='green')
plt.title('WLLN')
plt.axhline(y=10, color='r', linestyle='-')
plt.show()
```

Results

The following results are for Gaussian (10,4).



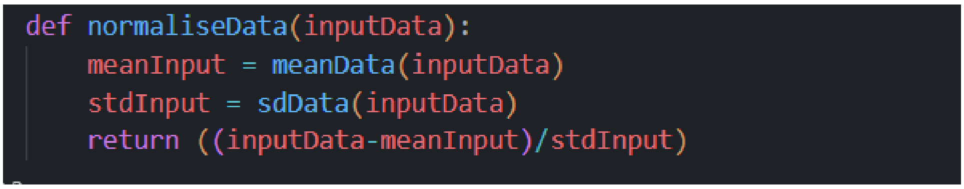
As is evident from the above image as the number of samples increase the sample mean converges to the expected mean which is 10 in this case.

Section B

Q(a)

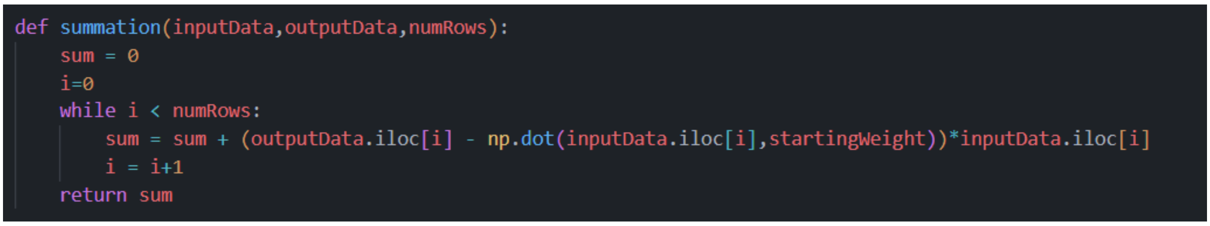
Our aim is to implement K-Fold cross-validation in this part. We take the K fold value in the range [2,5].

We first normalise the data using the below function



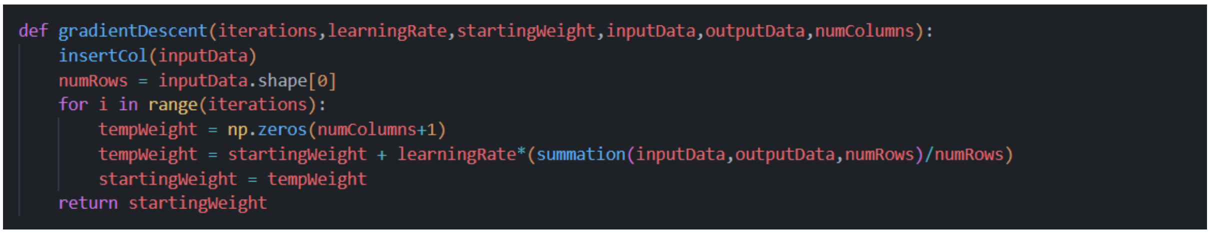
```
def normaliseData(inputData):
    meanInput = meanData(inputData)
    stdInput = sdData(inputData)
    return ((inputData-meanInput)/stdInput)
```

To run the gradient descent we first need to calculate the value of the output for the current value of the parameter



```
def summation(inputData,outputData,numRows):
    sum = 0
    i=0
    while i < numRows:
        sum = sum + (outputData.iloc[i] - np.dot(inputData.iloc[i],startingWeight))*inputData.iloc[i]
        i = i+1
    return sum
```

Now we need to calculate the gradient descent

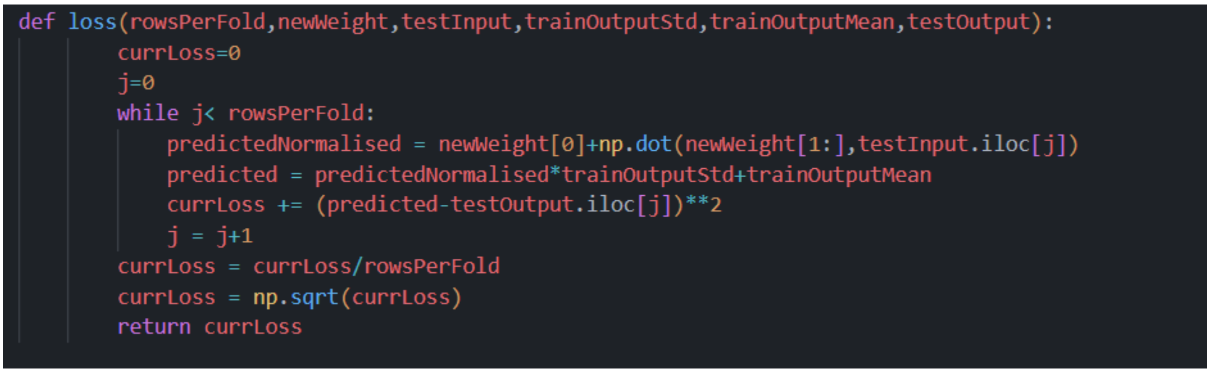


```
def gradientDescent(iterations,learningRate,startingWeight,inputData,outputData,numColumns):
    insertCol(inputData)
    numRows = inputData.shape[0]
    for i in range(iterations):
        tempWeight = np.zeros(numColumns+1)
        tempWeight = startingWeight + learningRate*(summation(inputData,outputData,numRows)/numRows)
        startingWeight = tempWeight
    return startingWeight
```

This follows the formula

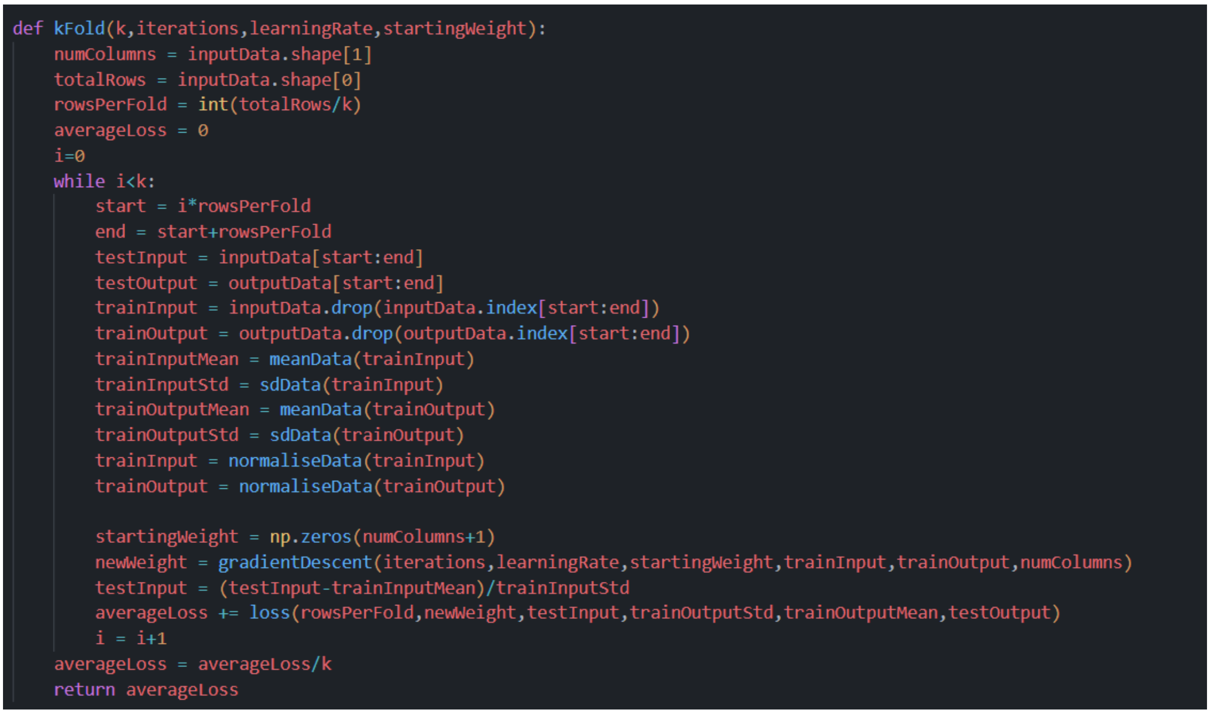
$$\theta := \theta + \alpha \sum_{i=1}^{n} \left(y^{(i)} - h_\theta(x^{(i)})\right) x^{(i)}$$

The net loss of the function for the current model is found using the following function which is RMSE error



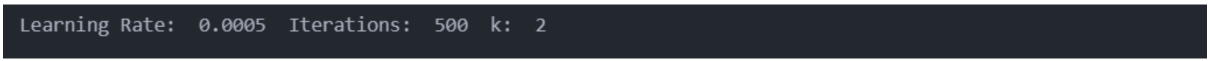
```
def loss(rowsPerFold,newWeight,testInput,trainOutputStd,trainOutputMean,testOutput):
        currLoss=0
        j=0
        while j< rowsPerFold:
            predictedNormalised = newWeight[0]+np.dot(newWeight[1:],testInput.iloc[j])
            predicted = predictedNormalised*trainOutputStd+trainOutputMean
            currLoss += (predicted-testOutput.iloc[j])**2
            j = j+1
        currLoss = currLoss/rowsPerFold
        currLoss = np.sqrt(currLoss)
        return currLoss
```

We also need to divide the data into training and testing set which is done by the below function

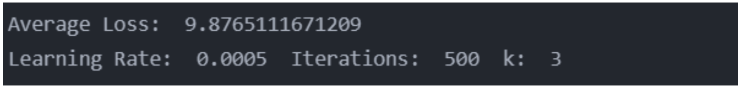


```
def kFold(k,iterations,learningRate,startingWeight):
    numColumns = inputData.shape[1]
    totalRows = inputData.shape[0]
    rowsPerFold = int(totalRows/k)
    averageLoss = 0
    i=0
    while i<k:
        start = i*rowsPerFold
        end = start+rowsPerFold
        testInput = inputData[start:end]
        testOutput = outputData[start:end]
        trainInput = inputData.drop(inputData.index[start:end])
        trainOutput = outputData.drop(outputData.index[start:end])
        trainInputMean = meanData(trainInput)
        trainInputStd = sdData(trainInput)
        trainOutputMean = meanData(trainOutput)
        trainOutputStd = sdData(trainOutput)
        trainInput = normaliseData(trainInput)
        trainOutput = normaliseData(trainOutput)

        startingWeight = np.zeros(numColumns+1)
        newWeight = gradientDescent(iterations,learningRate,startingWeight,trainInput,trainOutput,numColumns)
        testInput = (testInput-trainInputMean)/trainInputStd
        averageLoss += loss(rowsPerFold,newWeight,testInput,trainOutputStd,trainOutputMean,testOutput)
        i = i+1
    averageLoss = averageLoss/k
    return averageLoss
```

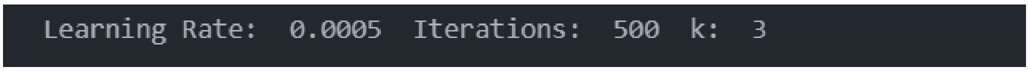
The results are as follows :



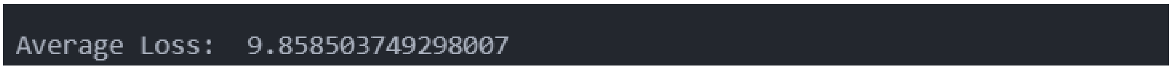
```
Learning Rate:  0.0005  Iterations:  500  k:  2
```



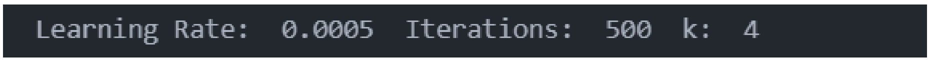
```
Average Loss:  9.8765111671209
Learning Rate:  0.0005  Iterations:  500  k:  3
```



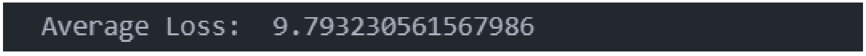
```
Learning Rate:  0.0005  Iterations:  500  k:  3
```



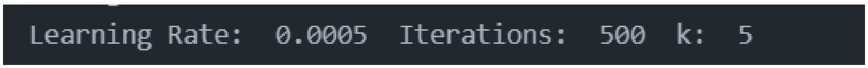
```
Average Loss:  9.858503749298007
```



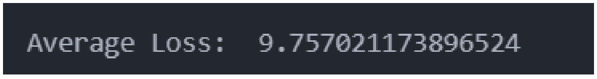
```
Learning Rate:  0.0005  Iterations:  500  k:  4
```



```
Average Loss:  9.793230561567986
```



```
Learning Rate:  0.0005  Iterations:  500  k:  5
```

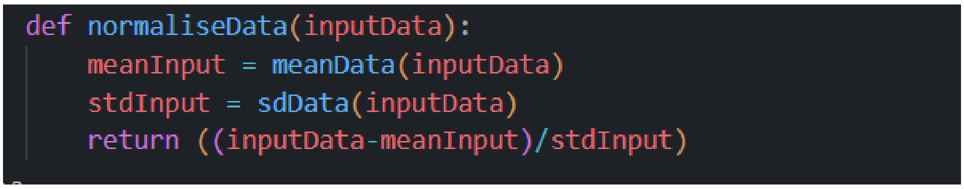


```
Average Loss:  9.757021173896524
```

Q(b)

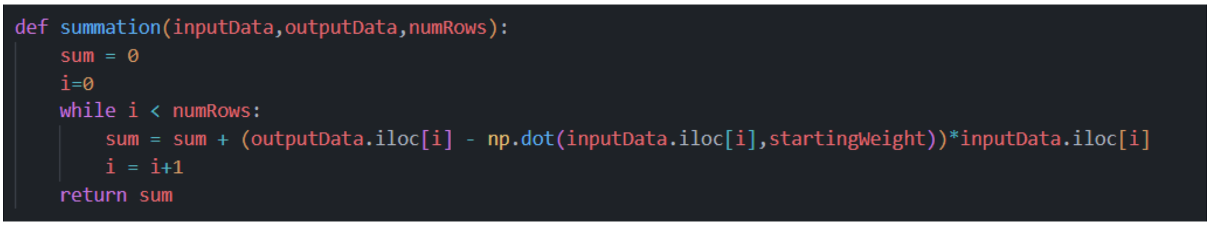
We need to draw the graph of RMSE Loss vs number of iterations for which gradient descent is run. We do this for K fold value of 4 which I found optimal.

We first normalise the data using the below function



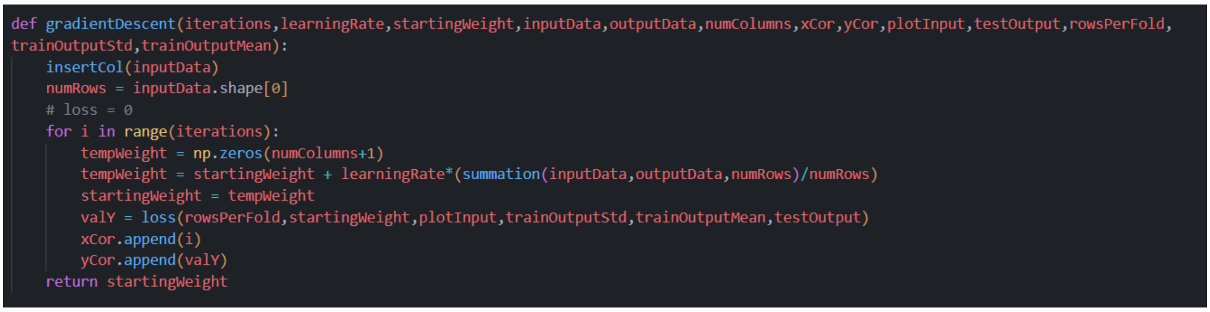
```
def normaliseData(inputData):
    meanInput = meanData(inputData)
    stdInput = sdData(inputData)
    return ((inputData-meanInput)/stdInput)
```

To run the gradient descent we first need to calculate the value of the output for the current value of the parameter



```
def summation(inputData,outputData,numRows):
    sum = 0
    i=0
    while i < numRows:
        sum = sum + (outputData.iloc[i] - np.dot(inputData.iloc[i],startingWeight))*inputData.iloc[i]
        i = i+1
    return sum
```

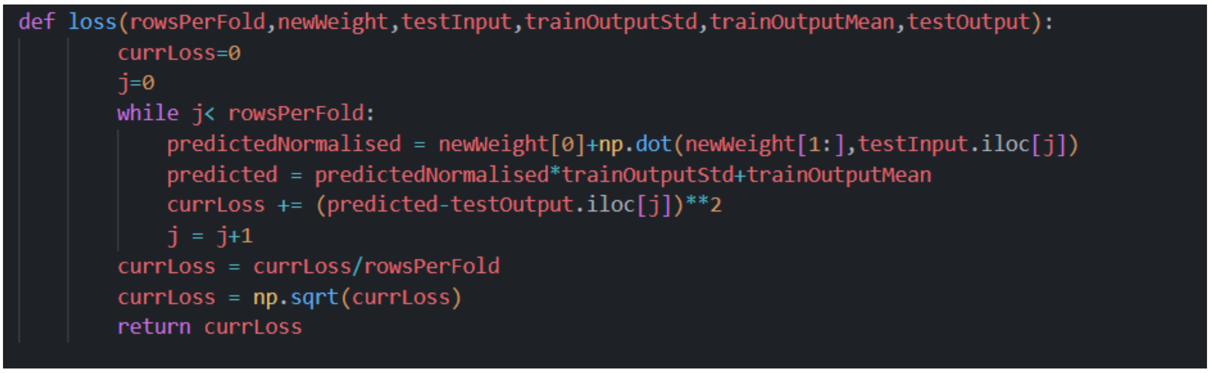
Now we need to calculate the gradient descent



```
def gradientDescent(iterations,learningRate,startingWeight,inputData,outputData,numColumns,xCor,yCor,plotInput,testOutput,rowsPerFold,
trainOutputStd,trainOutputMean):
    insertCol(inputData)
    numRows = inputData.shape[0]
    # loss = 0
    for i in range(iterations):
        tempWeight = np.zeros(numColumns+1)
        tempWeight = startingWeight + learningRate*(summation(inputData,outputData,numRows)/numRows)
        startingWeight = tempWeight
        valY = loss(rowsPerFold,startingWeight,plotInput,trainOutputStd,trainOutputMean,testOutput)
        xCor.append(i)
        yCor.append(valY)
    return startingWeight
```

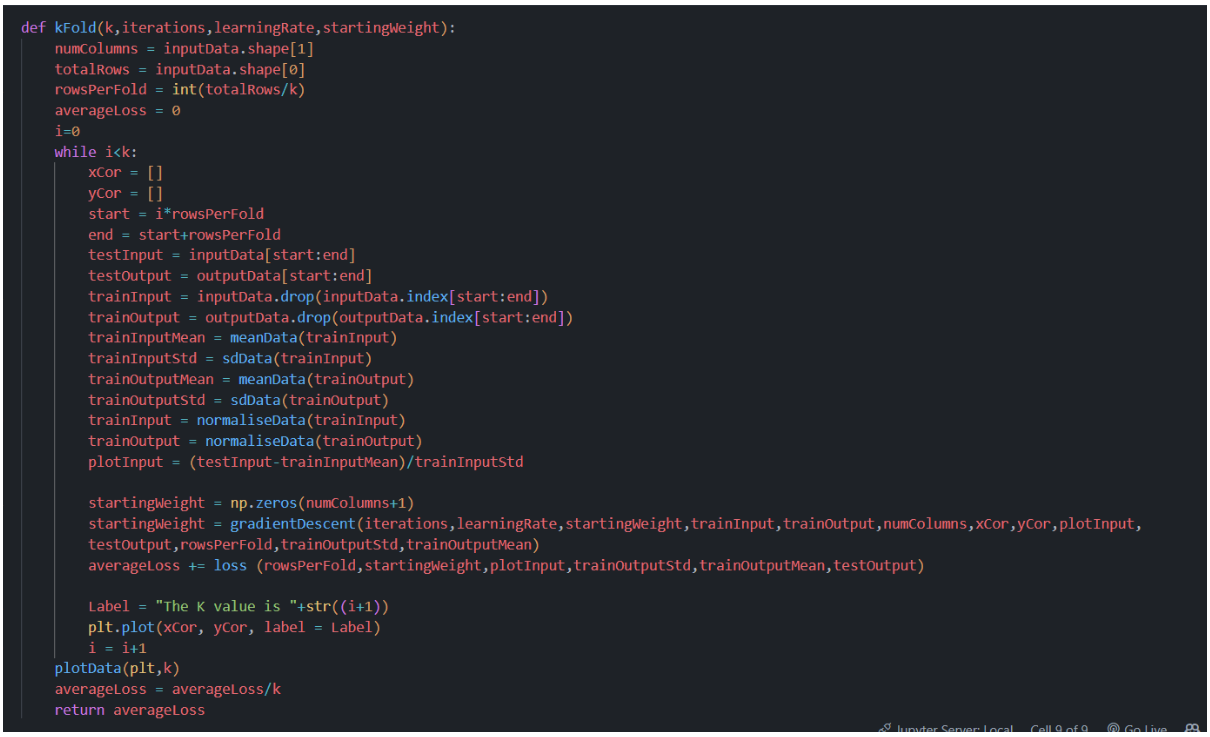
The gradient descent of this and the Above a part is similar but here we are appending the loss in the prediction so that we can plot the RMSE loss graph vs the number of iterations which will keep changing and increasing.

The net loss of the function for the current model is found using the following function which is RMSE error



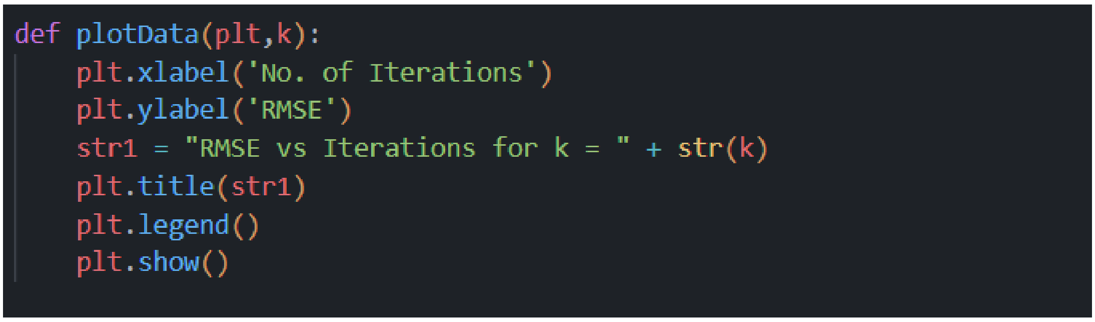
```
def loss(rowsPerFold,newWeight,testInput,trainOutputStd,trainOutputMean,testOutput):
        currLoss=0
        j=0
        while j< rowsPerFold:
            predictedNormalised = newWeight[0]+np.dot(newWeight[1:],testInput.iloc[j])
            predicted = predictedNormalised*trainOutputStd+trainOutputMean
            currLoss += (predicted-testOutput.iloc[j])**2
            j = j+1
        currLoss = currLoss/rowsPerFold
        currLoss = np.sqrt(currLoss)
        return currLoss
```

We also need to divide the data into training and testing set which is done by the below function. It also calls the gradient descent and gives the optimal ML Model. Then it calculates the loss and call the plotData() function to plot the data.



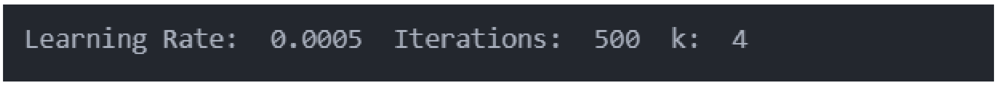
```python
def kFold(k,iterations,learningRate,startingWeight):
    numColumns = inputData.shape[1]
    totalRows = inputData.shape[0]
    rowsPerFold = int(totalRows/k)
    averageLoss = 0
    i=0
    while i<k:
        xCor = []
        yCor = []
        start = i*rowsPerFold
        end = start+rowsPerFold
        testInput = inputData[start:end]
        testOutput = outputData[start:end]
        trainInput = inputData.drop(inputData.index[start:end])
        trainOutput = outputData.drop(outputData.index[start:end])
        trainInputMean = meanData(trainInput)
        trainInputStd = sdData(trainInput)
        trainOutputMean = meanData(trainOutput)
        trainOutputStd = sdData(trainOutput)
        trainInput = normaliseData(trainInput)
        trainOutput = normaliseData(trainOutput)
        plotInput = (testInput-trainInputMean)/trainInputStd

        startingWeight = np.zeros(numColumns+1)
        startingWeight = gradientDescent(iterations,learningRate,startingWeight,trainInput,trainOutput,numColumns,xCor,yCor,plotInput,
        testOutput,rowsPerFold,trainOutputStd,trainOutputMean)
        averageLoss += loss (rowsPerFold,startingWeight,plotInput,trainOutputStd,trainOutputMean,testOutput)

        Label = "The K value is "+str((i+1))
        plt.plot(xCor, yCor, label = Label)
        i = i+1
    plotData(plt,k)
    averageLoss = averageLoss/k
    return averageLoss
```

We plot the data using the below plot function

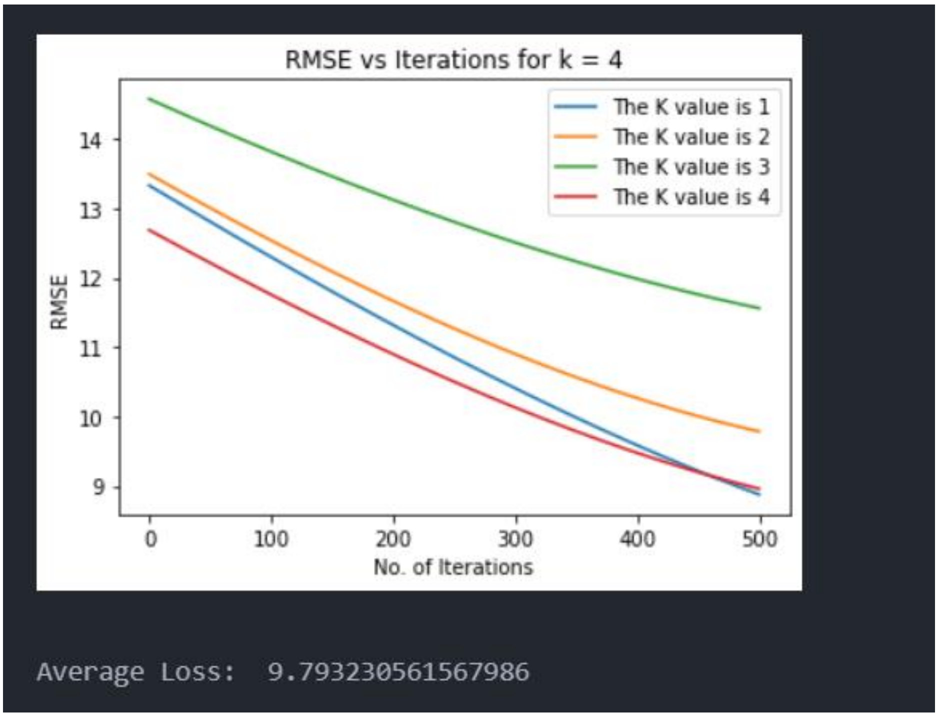


```python
def plotData(plt,k):
    plt.xlabel('No. of Iterations')
    plt.ylabel('RMSE')
    str1 = "RMSE vs Iterations for k = " + str(k)
    plt.title(str1)
    plt.legend()
    plt.show()
```

The results that we obtain are as follows:



```
Learning Rate:  0.0005  Iterations:  500  k:  4
```



```
Average Loss:  9.793230561567986
```

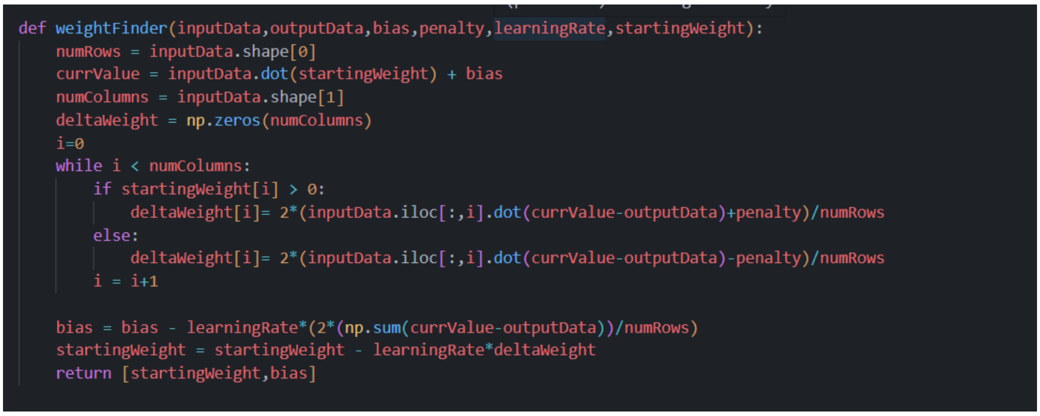
Here as is evident for the graph that for different values of K in K-Fold the RMSE error decreases with the increasing number of iterations.

For K value 4 the ie. 4-fold the RMSE error decreases from 13 to almost 9 after 500 iterations.

Q(c) – Lasso Regularisation

We need to avoid overfitting of the data, so we need to introduce Lasso Regularisation. It adds a penalty to our gradient descent to overcome the problem of overfitting.

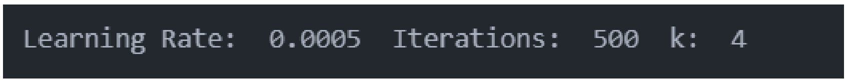
It is almost the same as the RMSE graph part as above just we need to update the function for finding the weights and include a penalty term. We do it for the optimal K fold value which is 4.



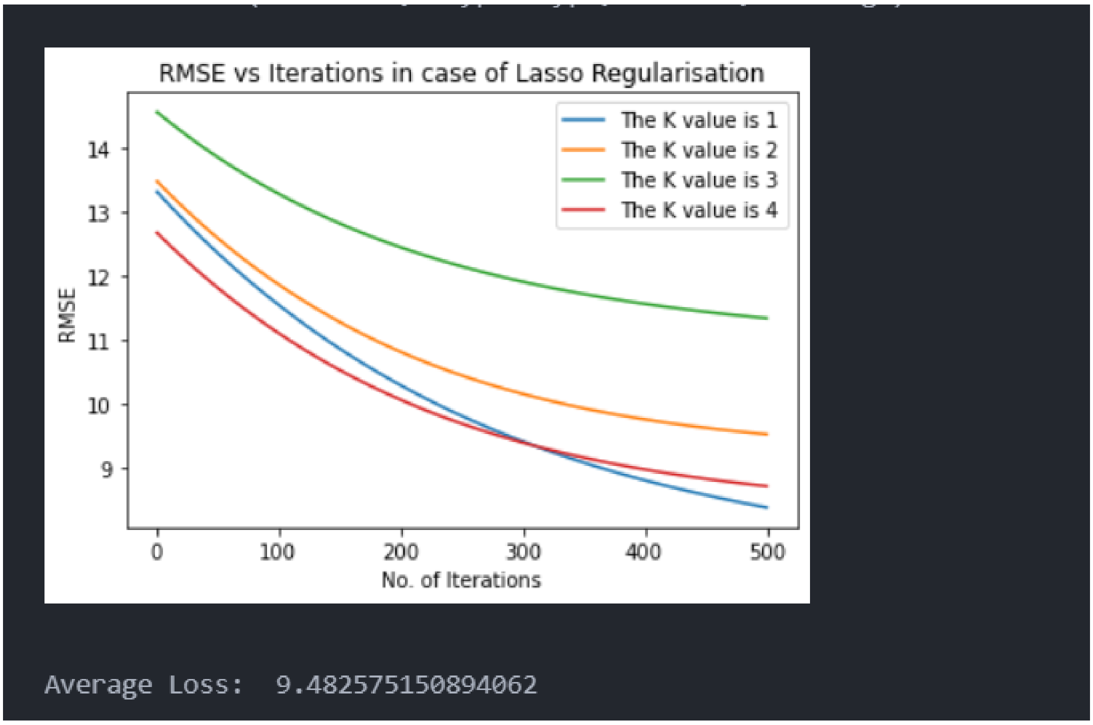
```python
def weightFinder(inputData,outputData,bias,penalty,learningRate,startingWeight):
    numRows = inputData.shape[0]
    currValue = inputData.dot(startingWeight) + bias
    numColumns = inputData.shape[1]
    deltaWeight = np.zeros(numColumns)
    i=0
    while i < numColumns:
        if startingWeight[i] > 0:
            deltaWeight[i]= 2*(inputData.iloc[:,i].dot(currValue-outputData)+penalty)/numRows
        else:
            deltaWeight[i]= 2*(inputData.iloc[:,i].dot(currValue-outputData)-penalty)/numRows
        i = i+1

    bias = bias - learningRate*(2*(np.sum(currValue-outputData))/numRows)
    startingWeight = startingWeight - learningRate*deltaWeight
    return [startingWeight,bias]
```

Now we just run the K-Fold which calls the gradient descent.

The results that we obtain are as follows



```
Learning Rate:  0.0005  Iterations:  500  k:  4
```

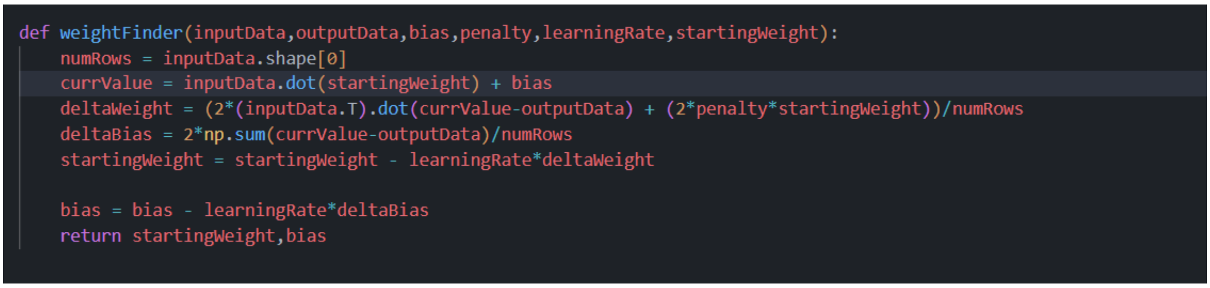


```
Average Loss:  9.482575150894062
```

Q(c)- Ridge Regularisation

Here also we need to do Regularisation to prevent overfitting only difference being that we use Ridge Regularisation Instead of Lasso Regularisation.

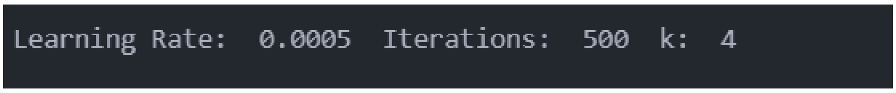
The only change in it from Lasso is the weight finding function which is given below.



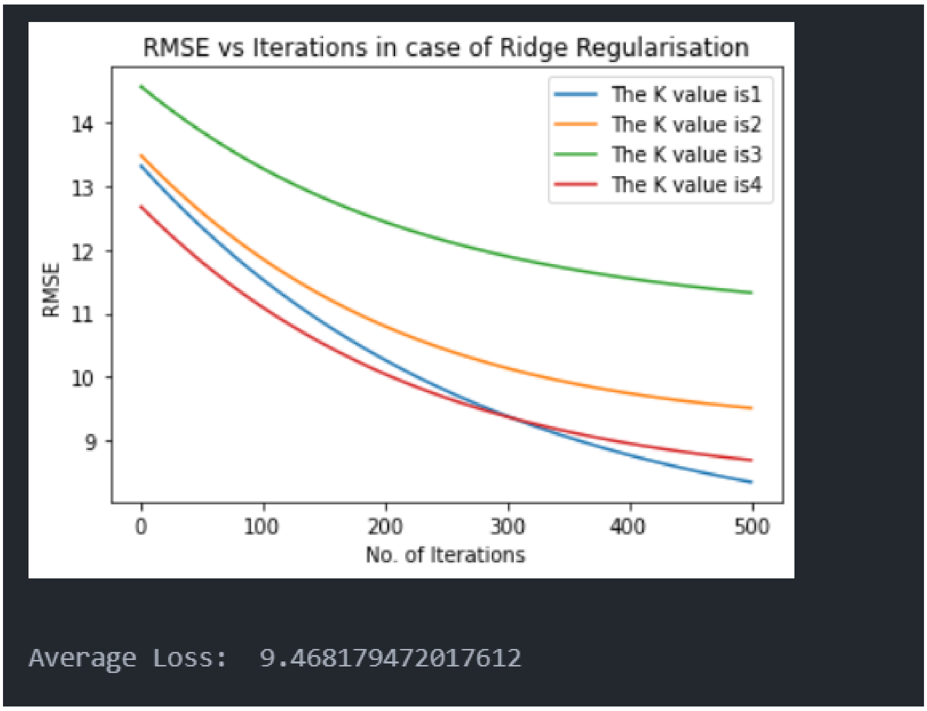
```
def weightFinder(inputData,outputData,bias,penalty,learningRate,startingWeight):
    numRows = inputData.shape[0]
    currValue = inputData.dot(startingWeight) + bias
    deltaWeight = (2*(inputData.T).dot(currValue-outputData) + (2*penalty*startingWeight))/numRows
    deltaBias = 2*np.sum(currValue-outputData)/numRows
    startingWeight = startingWeight - learningRate*deltaWeight

    bias = bias - learningRate*deltaBias
    return startingWeight,bias
```

Rest everything remains the same. We do it for K fold value of 4.

The results we obtain are as follows



```
Learning Rate:  0.0005  Iterations:  500  k:  4
```



```
Average Loss:  9.468179472017612
```

Final Conclusion for Regularisation

The loss obtained using Lasso Regularisation was 9.482575150894062

The loss obtained using the Ridge Regularisation was 9.468179472017612

The loss obtained without the regularisation is 9.793230561567986

Hence, we can see that regularisation does help us decrease the error. We also see that the performance of the Ridge is slightly better than the Lasso Regularisation.

Q(d)- Normal Equation

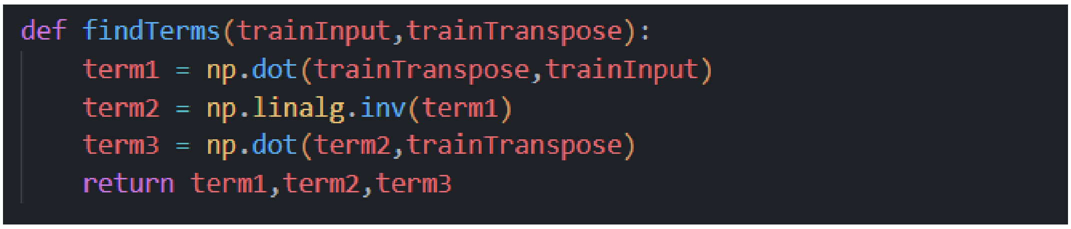
We can use the normal equation instead of gradient descent which will give more accurate answers but will be more computationally expensive.

The normal equation to minimise the cost function and get the most optimal parameters is

$$\theta = (X^TX)^{-1}X^T\vec{y}.$$

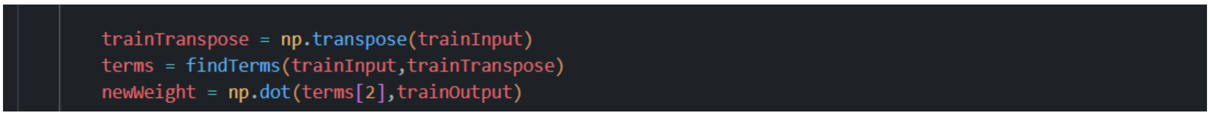
The procedure is same as above just in K-Fold we have removed the gradient descent and implemented the following function to get the optimal value for the parameters

The following function calculates the two terms in the above equation except the output y



```
def findTerms(trainInput,trainTranspose):
    term1 = np.dot(trainTranspose,trainInput)
    term2 = np.linalg.inv(term1)
    term3 = np.dot(term2,trainTranspose)
    return term1,term2,term3
```

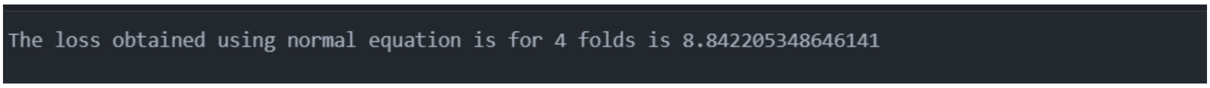
Now we take the dot product of the final term from above with the output



```
        trainTranspose = np.transpose(trainInput)
        terms = findTerms(trainInput,trainTranspose)
        newWeight = np.dot(terms[2],trainOutput)
```

We have taken the optimal K-Fold to be 4.

The results obtained are given below



```
The loss obtained using normal equation is for 4 folds is 8.842205348646141
```

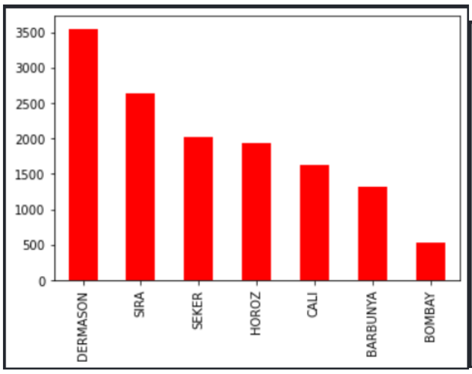
Section C

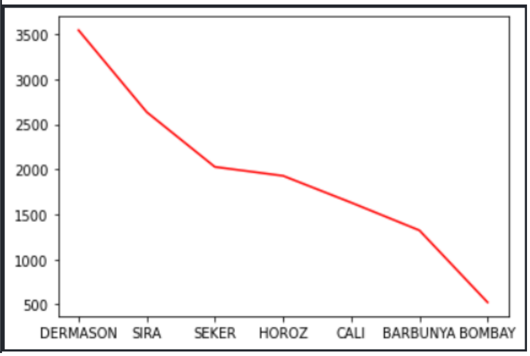
Q(a)

Since we are making use of Libraries here, we need to import all the relevant libraries like numpy, matlplotlib, seaborn and pandas.

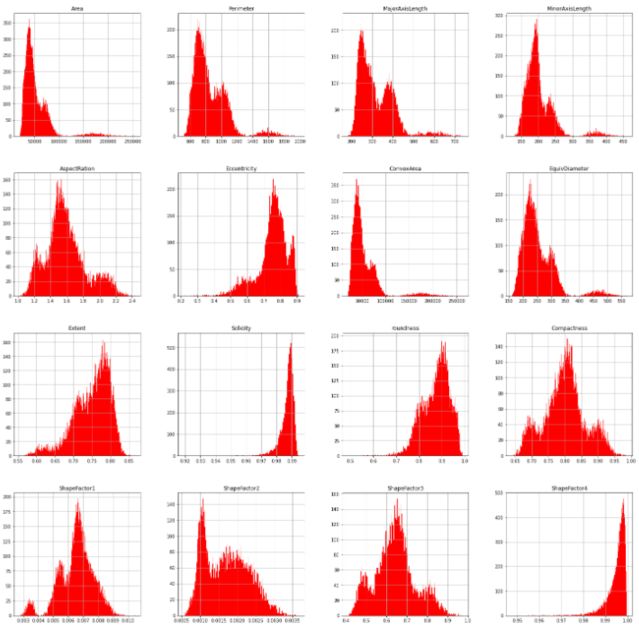
We then make a pandas dataframe from the given CSV Dataset file.

We then make a histogram and line plot according to the 'CLASS" column of the given dataset





We then plot the histogram of all the other columns in the dataset



Analysis from the Data

We can see that

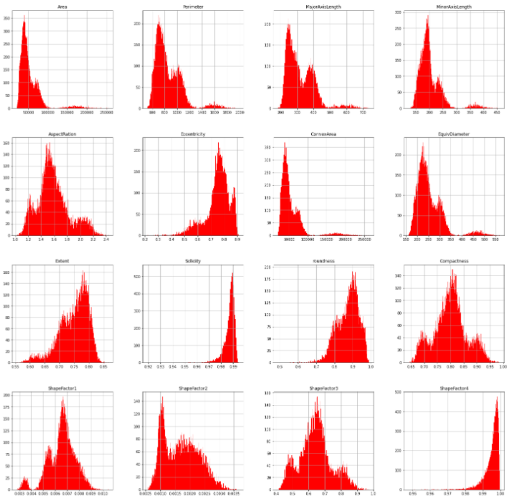
- Most Number of beans have a major axis length of around 250 and minor axis length of around 190.
- Area of around 42000 is the most common area
- Most number of beans belong to the class DERMASON and the least number of beans to the class BOMBAY

Q(b)

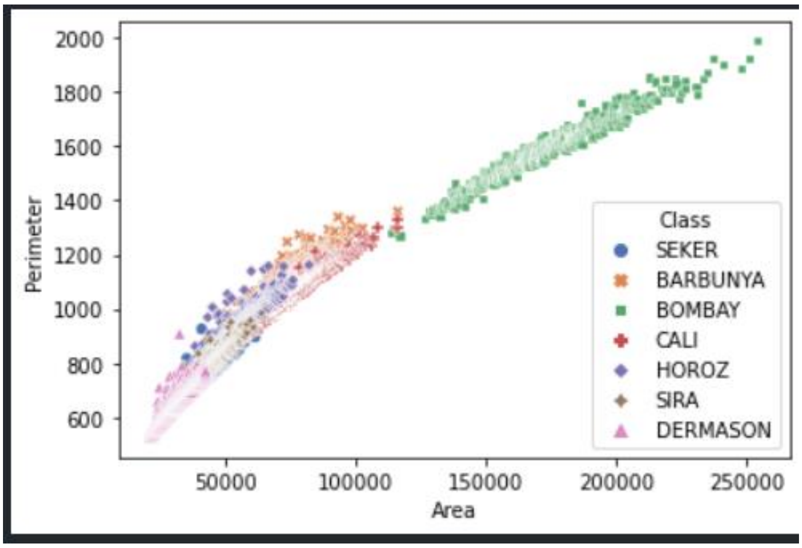
Here we need to perform and Explanatory Data Analysis.

We use df.info() to get the details about the dataset like number of columns, value types etc and df.describe() to get the details like count, average, maximum and minimum etc for the all the columns. We also use df.shape() to get the number of rows which is 13611 and number of columns which are 17 for our dataset.

Now first we plot the histogram for each column. This helps us better visualise the frequency of values present in a column.

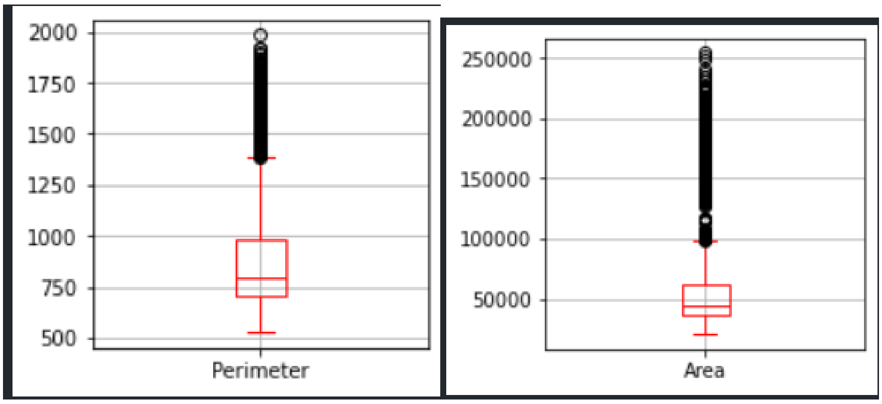


- Now we plot a scatterplot between Area and Perimeter which better helps understand the relationship between the two and what area and perimeter is common in a class

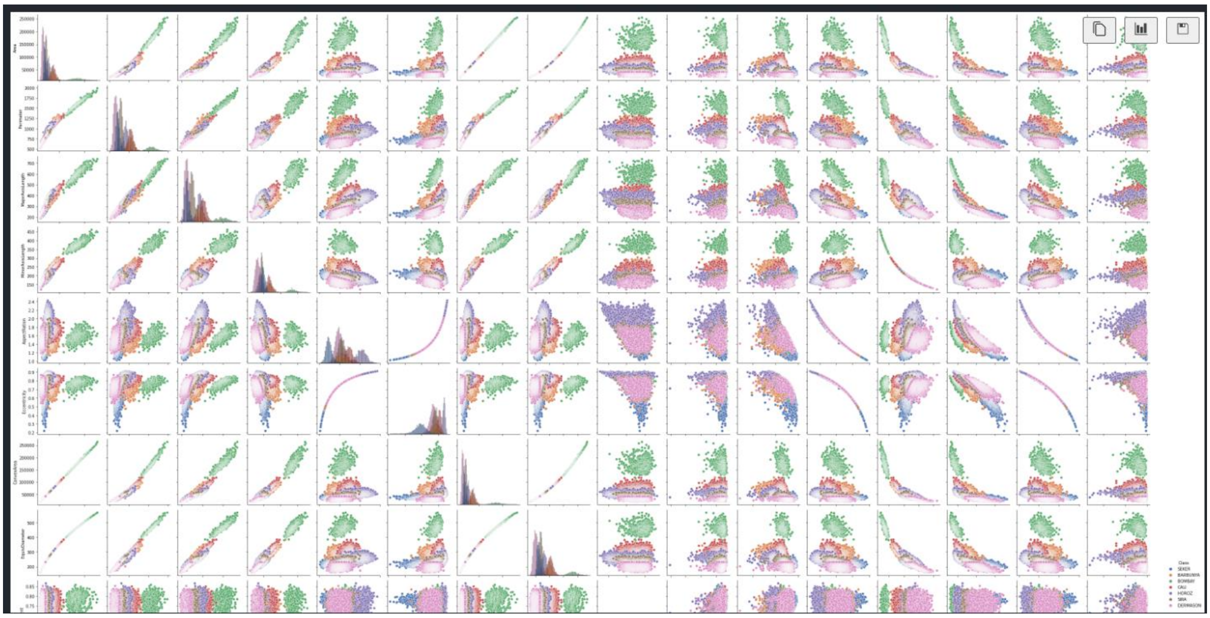


We observe that area is directly related to the perimeter and that the BOMBAY class tends to have the highest area and perimeter.

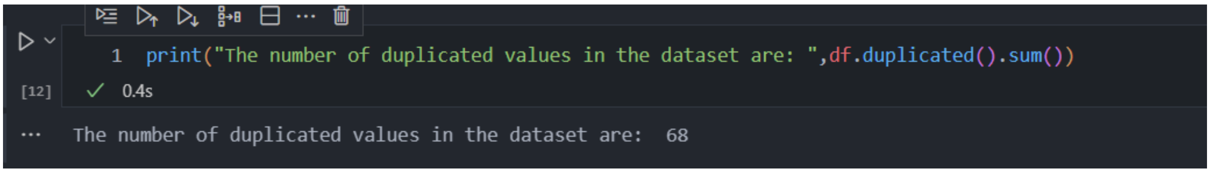
- We now make the Boxplots for Area and perimeter to check for the outliers in the data



- We now plot a pairplot to given the pairwise correlation between of the columns using sns.pairplot() function. A high positive correlation means that the columns are highly positively related and a high negative correlation means that the columns are highly negatively related. The correlation can take values between [– 1,1] and a low absolute values indicates low correlation.



- We now find the duplicate values in the dataset using df.duplicate() which turns out to be 68

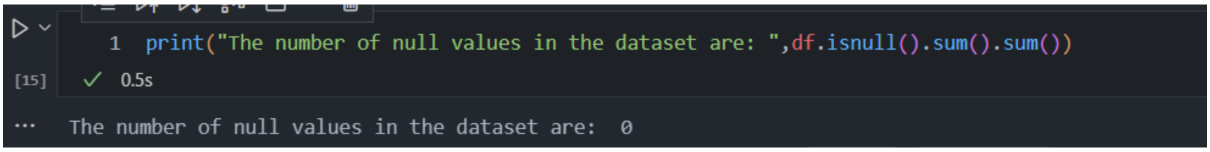


```
print("The number of duplicated values in the dataset are: ",df.duplicated().sum())
```

[12] ✓ 0.4s

```
The number of duplicated values in the dataset are:  68
```

- We find the number of null values in the dataset using df.null() which turns out to be 0



```
print("The number of null values in the dataset are: ",df.isnull().sum().sum())
```

[15] ✓ 0.5s

```
The number of null values in the dataset are:  0
```

The analysis drawn from the data are

- There are no null values in the dataset. So, there are missing values
- We can see that the DERMASON is the most abundant class while BOMBAY is the least abundant class
- BOMBAY class has the highest area and perimeter while the DERMASON class the lowest.
- There are many outliers in the area and perimeter in the dataset
- BOMBAY Class has high Major and Minor Axis length and Minor Axis Length.
- ConvexArea and Area have a direct correlation. i.e., As ConvexArea increases the Area increases.

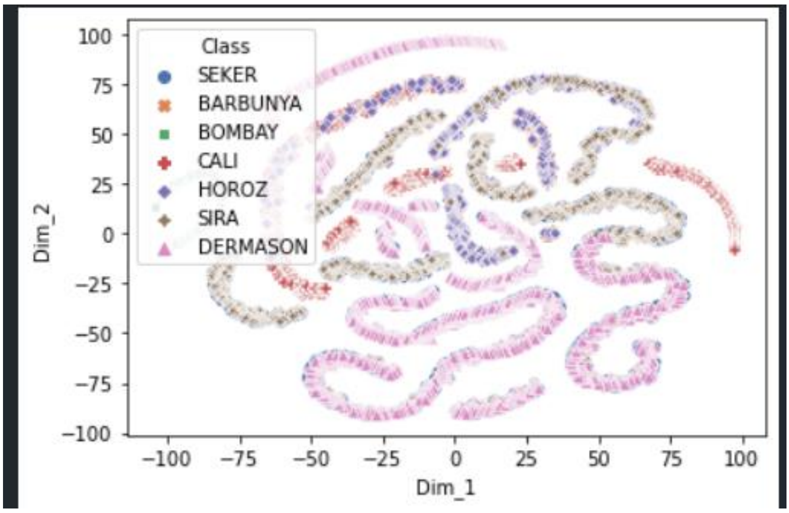
Q(c)

TCA helps us to analyse high dimensional data. It first measures the similarities in a high n dimensional space and then convert it to JOINT PDF to retain and reflect the probability as accurately as possible.

The target class is first separated to another variable.

We first fit the data into TSNE using fit_transform().

We now plot the result of the TSNE using a scatterplot

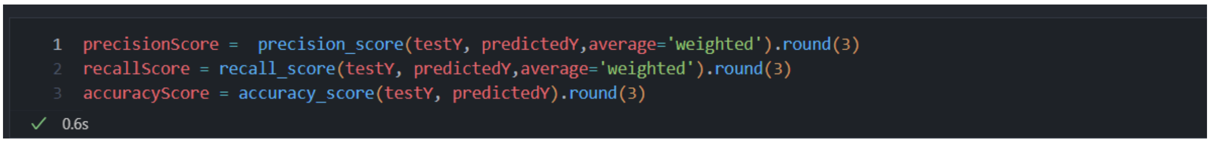


It is clear from the scatterplot that the data cannot be separated using a straight line as there is lot of overlap between different classes.

Q3(d)

Firstly, we do Gaussian Naïve Bayes

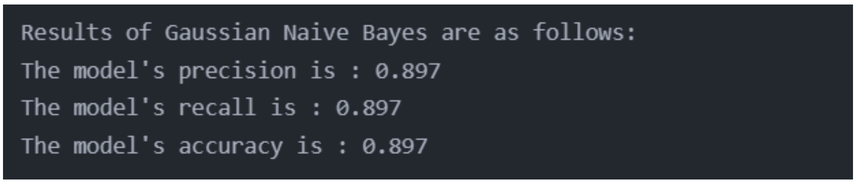
- We first make a train test split of the data in 80:20 fashion using **train_test_split()**
- We first need to import modules from the sklearn.
- We now need to run Multinomial and Gaussian Naïve Bayes.
- We first normalise the data using standard scalar
- We then fit our data in the gaussian imported from sklearn using gaussian.fit(trainX,trainY)
- We then get predictedY output using gaussian.predict(testX).
- We then calculate the precission score, recall score and accuracy score using precision_score,recall_score, and accuracy_score functions of sklearn



```
precisionScore =  precision_score(testY, predictedY,average='weighted').round(3)
recallScore = recall_score(testY, predictedY,average='weighted').round(3)
accuracyScore = accuracy_score(testY, predictedY).round(3)
```

✓ 0.6s

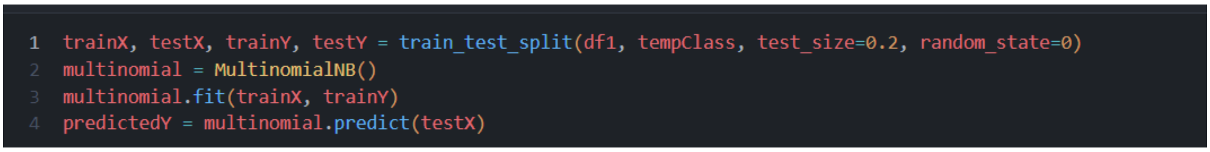
The results obtained using Gaussian Naïve Bayes are as follows:



```
Results of Gaussian Naive Bayes are as follows:
The model's precision is : 0.897
The model's recall is : 0.897
The model's accuracy is : 0.897
```

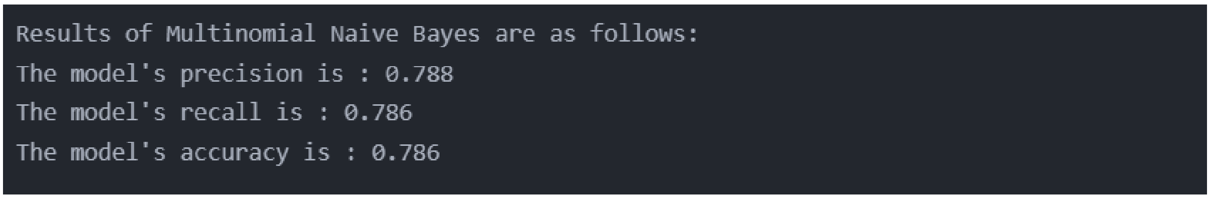
We now do the Multinomial Naïve Bayes using MultinomialNB of sklearn.

- We repeat the same steps as Gaussian above but we just use multinomial object from MultinomalNB().



```
trainX, testX, trainY, testY = train_test_split(df1, tempClass, test_size=0.2, random_state=0)
multinomial = MultinomialNB()
multinomial.fit(trainX, trainY)
predictedY = multinomial.predict(testX)
```

- The results obtained are as follows



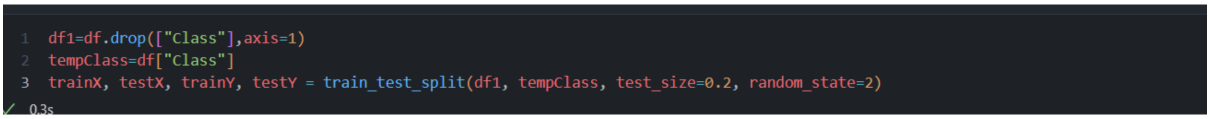
```
Results of Multinomial Naive Bayes are as follows:
The model's precision is : 0.788
The model's recall is : 0.786
The model's accuracy is : 0.786
```

As we observe that the accuracy of Gaussian Naïve Bayes is 0.897 while that of Multinomial Naïve Bayes is 0.788 which is considerably lower.

Hence is evident from above that Gaussian Naïve Bayes performs better than Multinomial Naïve Bayes for the given data.

Q3(e)

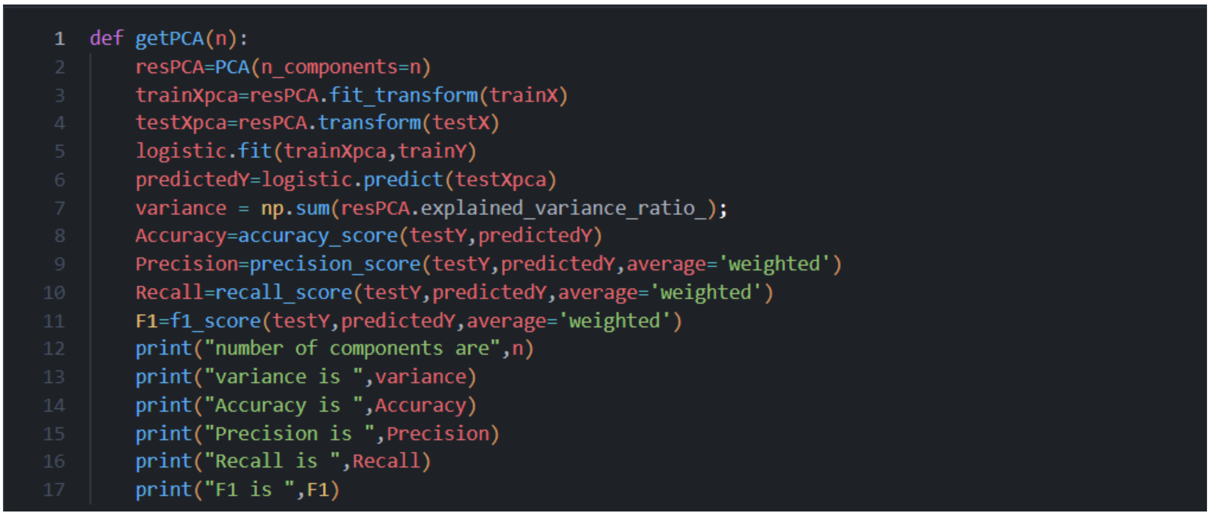
- PCA is used to transform high dimensional data to lower dimensional data while trying to retain as much information as possible. It helps us to analyse more complex data with multiple features efficiently.
- We make test and train split.



```
df1=df.drop(["Class"],axis=1)
tempClass=df["Class"]
trainX, testX, trainY, testY = train_test_split(df1, tempClass, test_size=0.2, random_state=2)
```

✓ 0.3s

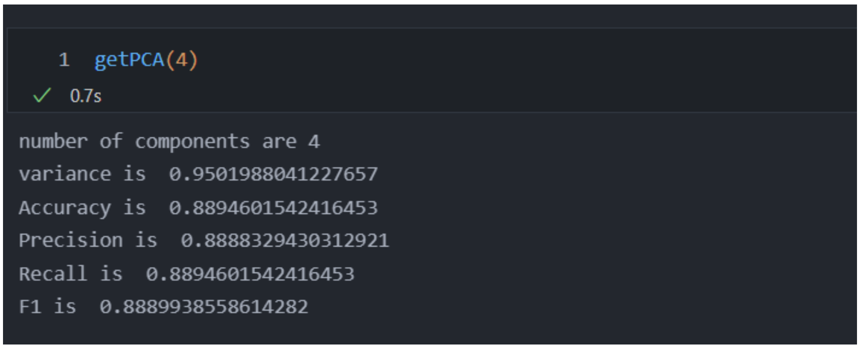
- We then make fit_transform() on our train data to normalise the train data and transform() for the test data.
- We now do logistic regression using logistic () from sklearn and use the data modified from PCA(n) function from sklearn for n components of PCA.



```
def getPCA(n):
    resPCA=PCA(n_components=n)
    trainXpca=resPCA.fit_transform(trainX)
    testXpca=resPCA.transform(testX)
    logistic.fit(trainXpca,trainY)
    predictedY=logistic.predict(testXpca)
    variance = np.sum(resPCA.explained_variance_ratio_);
    Accuracy=accuracy_score(testY,predictedY)
    Precision=precision_score(testY,predictedY,average='weighted')
    Recall=recall_score(testY,predictedY,average='weighted')
    F1=f1_score(testY,predictedY,average='weighted')
    print("number of components are",n)
    print("variance is ",variance)
    print("Accuracy is ",Accuracy)
    print("Precision is ",Precision)
    print("Recall is ",Recall)
    print("F1 is ",F1)
```

- The results for the different values of PCA are as follows

- PCA with 4 components

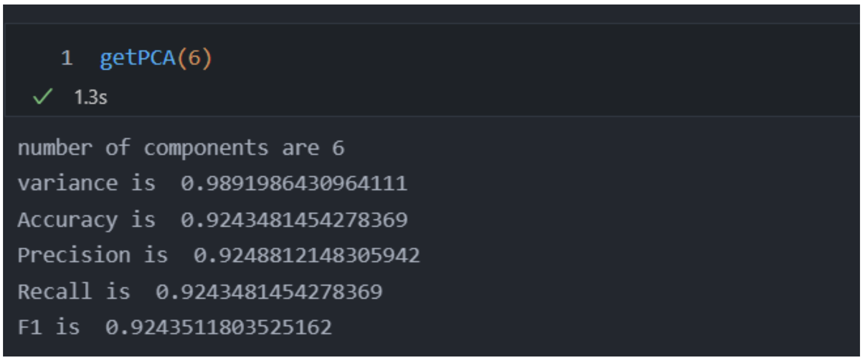


```
getPCA(4)
```

✓ 0.7s

```
number of components are 4
variance is  0.9501988041227657
Accuracy is  0.8894601542416453
Precision is  0.8888329430312921
Recall is  0.8894601542416453
F1 is  0.8889938558614282
```

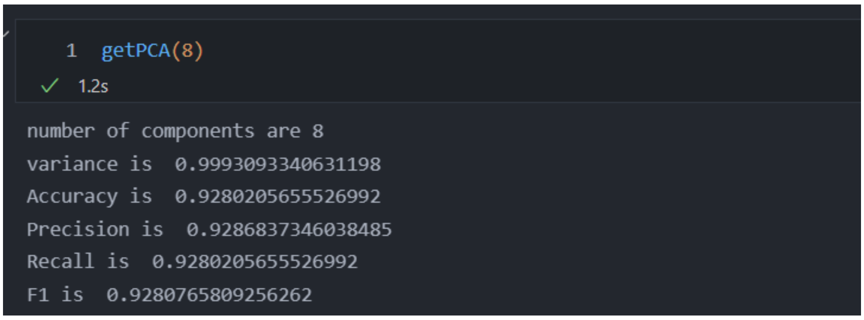
- PCA with 6 components



```
getPCA(6)
```

```
number of components are 6
variance is  0.9891986430964111
Accuracy is  0.9243481454278369
Precision is  0.9248812148305942
Recall is  0.9243481454278369
F1 is  0.9243511803525162
```

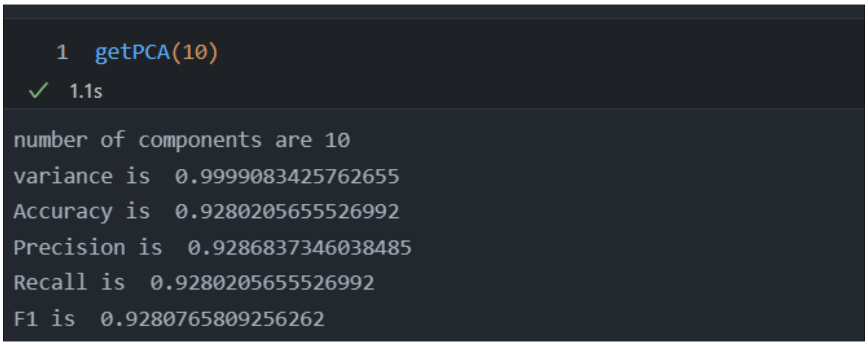
- PCA with 8 components



```
getPCA(8)
```

```
number of components are 8
variance is  0.9993093340631198
Accuracy is  0.9280205655526992
Precision is  0.9286837346038485
Recall is  0.9280205655526992
F1 is  0.9280765809256262
```

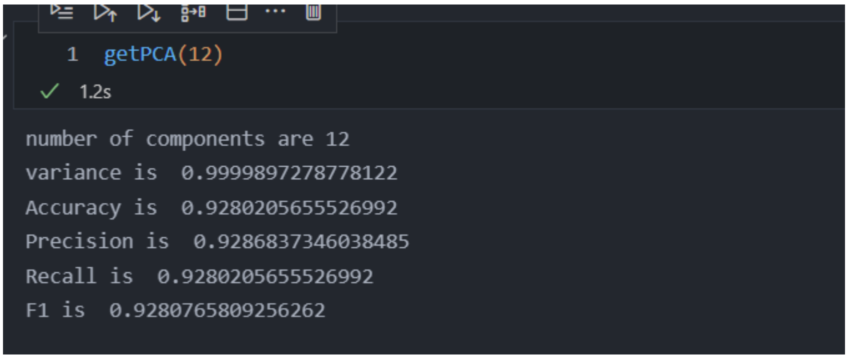
- PCA with 10 components



```
getPCA(10)
```

```
number of components are 10
variance is  0.9999083425762655
Accuracy is  0.9280205655526992
Precision is  0.9286837346038485
Recall is  0.9280205655526992
F1 is  0.9280765809256262
```

- PCA with 12 components



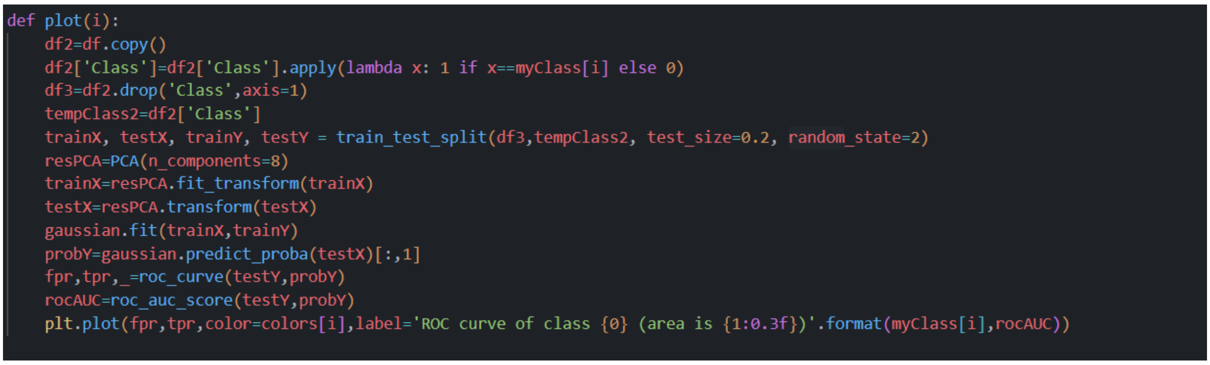
```
getPCA(12)
```

```
number of components are 12
variance is  0.9999897278778122
Accuracy is  0.9280205655526992
Precision is  0.9286837346038485
Recall is  0.9280205655526992
F1 is  0.9280765809256262
```

- As we observe on decreasing the PCA components from 12 to 4 the accuracy and variance decreases as the amount of information lost on compressing the data increases with higher compression ie. Lower PCA components.

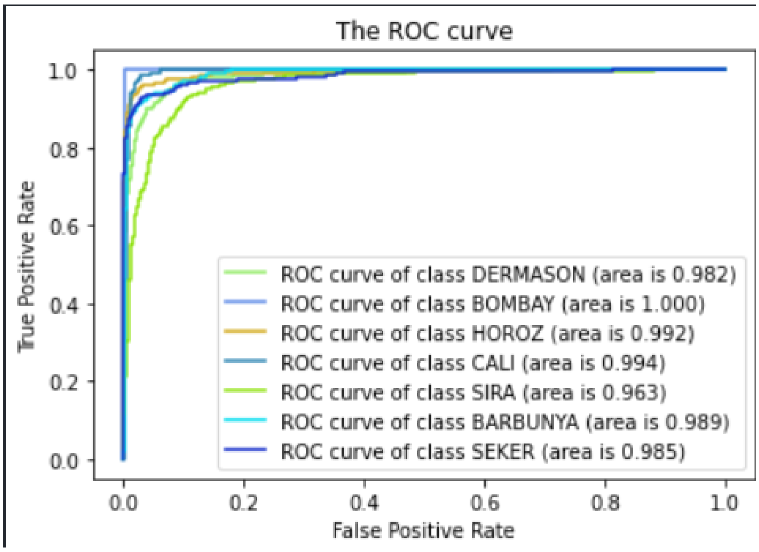
Q3(f)

- We first do the same process as done above for the PCA
- We do it for PCA components = 8



```
def plot(i):
    df2=df.copy()
    df2['Class']=df2['Class'].apply(lambda x: 1 if x==myClass[i] else 0)
    df3=df2.drop('Class',axis=1)
    tempClass2=df2['Class']
    trainX, testX, trainY, testY = train_test_split(df3,tempClass2, test_size=0.2, random_state=2)
    resPCA=PCA(n_components=8)
    trainX=resPCA.fit_transform(trainX)
    testX=resPCA.transform(testX)
    gaussian.fit(trainX,trainY)
    probY=gaussian.predict_proba(testX)[:,1]
    fpr,tpr,_=roc_curve(testY,probY)
    rocAUC=roc_auc_score(testY,probY)
    plt.plot(fpr,tpr,color=colors[i],label='ROC curve of class {0} (area is {1:0.3f})'.format(myClass[i],rocAUC))
```

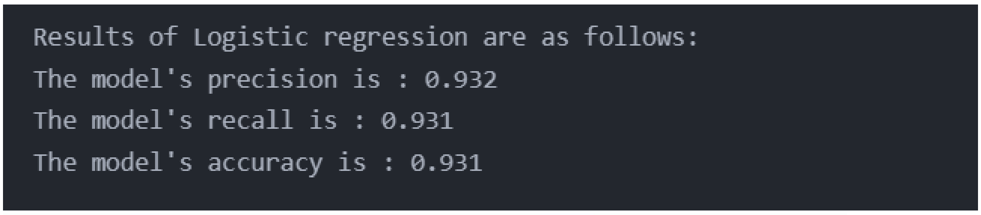
- Now we plot the ROC_AUC curve using roc_curve(testY,probY) for getting coordinates corresponding to false and true positive rates,where probY is the probability estimates for the output values and roc_auc_score(testY,probY) for getting the roc_auc Score



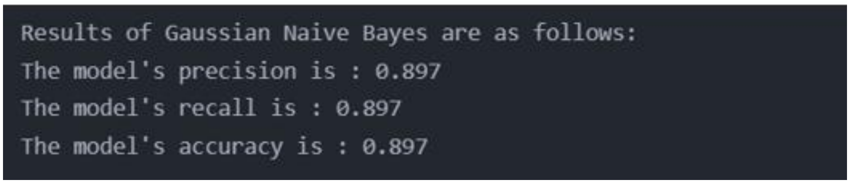
- As evident from the results all the models have a roc_auc score greater than 0.9 which demonstrate excellent results and much better than random guess which will give 0.5.
- We see that the BOMBAY class have the highest roc_auc score while the SIRA class has the smallest roc_auc score.

Q3(g)

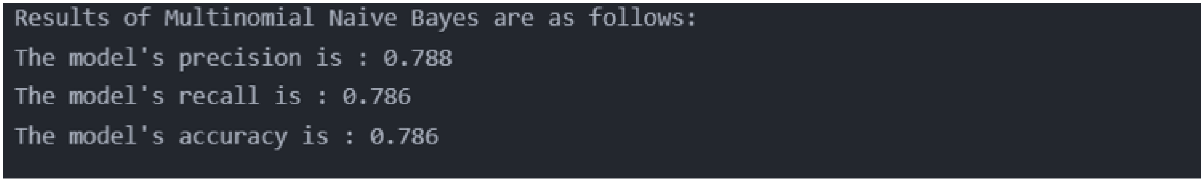
- We repeat the same steps as done in PCA for PCA components = 10
- We get the output using logistic.predict() as done in PCA.
- Now we get the
  precision_score
  accuracy_score
  and recall_score.
- The results are as follows



```
Results of Logistic regression are as follows:
The model's precision is : 0.932
The model's recall is : 0.931
The model's accuracy is : 0.931
```



```
Results of Gaussian Naive Bayes are as follows:
The model's precision is : 0.897
The model's recall is : 0.897
The model's accuracy is : 0.897
```



```
Results of Multinomial Naive Bayes are as follows:
The model's precision is : 0.788
The model's recall is : 0.786
The model's accuracy is : 0.786
```

- We find that the accuracy in the case of Logisitic Regression implementation of SKlearn the accuracy is much better than the Gaussian Naïve Bayes
- Accuracy for logistic is 0.931, Gaussian Naïve Bayes is 0.897 and Multinomial Naïve Bayes is 0.786.

Assignment - 1.

Q.1(a) To Prove :- The least square fit line always passes through the point $(\bar{x}, \bar{y})$, where $\bar{x}$ & $\bar{y}$ represent mean of X & Y.

To show $\bar{y}$ & $\bar{x}$ lie on the line we need to prove $\bar{y} = w_1\bar{x} + b$ where w_1 & b are parameters obtained from regression.
So, let y^i represent value of y for i^{th} sample & same follows for x^i

Let the linear regression model be

$$\hat{y} = w_1 x + b$$

Now $y^{(i)} = \hat{y}^{(i)} + \epsilon_i$ (True value = Predicted value + Error).

$$y^{(i)} = w_1 x^{(i)} + \epsilon_i + b$$

$$y^{(1)} = w_1 x^1 + b + \epsilon_1$$

$$y^{(2)} = w_1 x^{(2)} + b + \epsilon_2$$

$$\vdots$$

$$y^{(n)} = w_1 x^{(n)} + b + \epsilon_n$$

Summing above

$$\sum_{i=1}^{n} y^{(i)} = w_1 \sum_{i=1}^{n} x^{(i)} + bn + \sum_{i=1}^{n} \epsilon_i$$

dividing by n

$$\frac{1}{n}\sum_{i=1}^{n} y^{(i)} = \frac{w_1}{n}\sum_{i=1}^{n} x^{(i)} + \frac{bn}{n} + \frac{1}{n}\sum_{i=1}^{n} \epsilon_i$$

So, ~~Sum~~ to show that $\sum_{i=1}^{n} \epsilon_i = 0$,

Sum of mean square erros = $\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$

MSE $= \sum_{i=1}^{n} (y_i - w_1 x_i - b)^2$

For this to be min $\frac{\partial MSE}{\partial b} = 0$

$\Rightarrow \sum_{i=1}^{n} 2(y_i - w_1 x_i - b) -1 = 0$.

$\Rightarrow$ Now $\epsilon_i = y_i - w_1 x_i - b$

so $-2 \sum_{i=1}^{n} \epsilon_1 = 0 \Rightarrow \sum_{i=1}^{n} \epsilon_1 = 0$

So, $\bar{y} = w_1 \bar{x} + b + \bar{\epsilon}$

$\bar{y} = w_1 \bar{x} + b$ (as $\bar{\epsilon} = 0$)

Hence $(\bar{y}, \bar{x})$ lie on the least square fit line.

Q1 (b). Here let

$\rho_{x,y}$ denote correlation coefficient between x & y

Now, given

$\rho_{x,z} > 0$

$\rho_{y,z} > 0$

Now we claim $\rho_{x,y}$ need not be greater than 0 always

So, using partial correlation,

$$\rho_{xy.z} = \frac{\rho_{xy} - \rho_{xz} \cdot \rho_{zy}}{\sqrt{1-\rho^2_{xz}}\ \sqrt{1-\rho^2_{zy}}}$$

$$\Rightarrow \rho_{xy} = \rho_{xz} \cdot \rho_{yz} + \rho_{xy.z} \sqrt{1-\rho^2_{xz}}\ \sqrt{1-\rho^2_{yz}}$$

Now by condition of partial coeff. $-1 \leq \rho_{xy.z} \leq 1$.

So

$$\rho_{xy} = \rho_{xz} \cdot \rho_{yz} \pm \cancel{\rho_{xy.z}} . \sqrt{1-\rho^2_{xz}}\ \sqrt{1-\rho^2_{yz}}$$

So the upper & lower band of ρ_{xy} are

$\rho_{xz} \cdot \rho_{yz} + \sqrt{1-\rho^2_{xz}}\ \sqrt{1-\rho^2_{yz}}$ & $\rho_{xz} \cdot \rho_{yz} - \sqrt{1-\rho^2_{xz}}\ \sqrt{1-\rho^2_{yz}}$

respectively.

Now assuming $\rho_{xz} = 0.75$ & $\rho_{yz} = 0.8$ which indicates strong correlation b/w them

$$\rho_{xy} \in \left[(0.75)(0.8) - \sqrt{1-0.75^2}\sqrt{1-0.8^2}, (0.75)(0.8) + \sqrt{1-0.75^2}\sqrt{1-0.8^2}\right]$$

$$\rho_{xy} \in \left[0.6 - \sqrt{0.4375}\sqrt{0.36}, 0.6 + \sqrt{0.4375}\sqrt{0.36}\right]$$

$$\rho_{x,y} \in [0.6 - 0.39, 0.6 + 0.39]$$

$$\rho_{x,y} \in [0.21, 0.99].$$

So $\rho_{x,y}$ can be between 0.21 to 0.3 as well. which indicates a very weak correlation b/w x, y.

Hence a strong correlation is not guaranteed always

Hence the proof.

Q1 ⓒ <u>Given</u> :- $x_1, x_2, x_3 \dots x_n$ a sequence of iid random variables.

$E(x_1) = E(x_2) = \dots E(x_n) = \mu$

<u>To Prove</u> :- for $n \to \infty$ $\lim_{n\to\infty} \frac{x_1 + x_2 + \dots x_n}{n} \to \mu$

ie

~~$P\left(\left[\left|\frac{x_1+x_2+\dots x_n}{n}\right| > \mu\right]\right)$~~ ;

ie let $S_n = \frac{x_1+x_2+\dots x_n}{1}$

$\lim_{n\to\infty} P\left\{\left(\left|\frac{S_n}{n} - \mu\right|\right) \geq \epsilon\right) \longrightarrow 0$ for any $\epsilon > 0$

<u>Proof</u>

$S_n = x_1 + x_2 + \dots + x_n$

$\overline{S_n} = \frac{S_n}{n}$

$\overline{S_n} = \frac{x_1 + x_2 + \dots + x_n}{n}$

$\overline{S_n} = \frac{\mu n}{n}$

$\overline{S_n} = \mu$

Now, By Chebyshev's inequality,

$$P\left[|\bar{S_n}-u| \geq \epsilon\right] \leq \frac{Var(\bar{S_n})}{\epsilon^2}$$

Now

$$Var(\bar{S_n}) = Var\left(\frac{X_1+X_2+\dots+X_n}{n}\right)$$

$$= \frac{1}{n^2}\,Var(X_1+X_2+\dots+X_n)$$

$$= \frac{1}{n^2}\left[Var(X_1)+\dots Var(X_n)\right]$$ (as X_i are iid)

$$= \frac{n\sigma^2}{n^2}$$ (as $Var(X_1) = Var(X_2) = \dots Var(X_n) = \sigma^2$ as they are iid).

$$= \frac{\sigma^2}{n}$$

So, $P\left[|\bar{S_n}-u| \geq \epsilon\right] \leq \frac{\sigma^2}{n\epsilon^2}$

So $\lim\limits_{n\to\infty} P\left[|\bar{S_n}-u| \geq \epsilon\right] \leq \lim\limits_{n\to\infty} \frac{\sigma^2}{n\epsilon^2}$

≤ 0 (as $\lim\limits_{n\to\infty} \frac{1}{n} = 0$)

Hence Proved.

Q@ To Derive :- Maximum A Posteriori (MAP) Sol^n for linear regression

Now, $y = w^T x + \epsilon$.
here
$\epsilon \sim$ Normal $(0, \sigma^2)$

Hence $y|x \sim$ Normal $(w^T x, \sigma^2)$

$$\Rightarrow P(y_i|x_i) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x_i^T w - y_i)^2}{2\sigma^2}}$$

Now,

$$w = \underset{w}{\text{argmax}} \; P(w \,/\, y_1, x_1, y_2, x_2 -- y_n, x_n)$$

$$w = \underset{w}{\text{argmax}} \; \frac{P(y_1, x_1, -- y_n, x_n | w) P(w)}{P(y_1, x_1, -- y_n, x_n)} \quad \text{(Bays' Rule)}$$

$$= \underset{w}{\text{argmax}} \; P(y_1, x_1, -- y_n, x_n | w) \, P(w) .$$

Now we assume w to be $w \sim N(\vec{0}, \tau^2 I)$

$$\text{So, } P(w) = \cancel{\frac{1}{\sqrt{2\pi\sigma^2}}} \; \frac{1}{\sqrt{2\pi\tau^2}} \, e^{-\frac{1}{2}\left(\frac{w-0}{\tau}\right)^2} .$$

$$P(w) = \frac{1}{\sqrt{2\pi\tau^2}} \cdot e^{\frac{-w^2}{2\tau^2}}$$

Now $P(y_1, x_1 \cdots y_n, x_n | w) = \prod_{i=1}^{n} P(y_i, x_i | w)$]

Now ~~$P(x,y|z)$~~ = P·

$P(y, x|z) = P(y|x, z)\, P(x|z)$ (as y_i depends on x_i & are iid)

So, $P(y_1, x_1 - - y_n, x_n | w) = \prod_{i=1}^{n} \left[P(y_i | x_i, w)\; P(x_i|w)\right]$

$w = \underset{w}{\arg\max} \prod_{i=1}^{n} P[(y_i | x_i, w)]\; P(x_i|w) \cdot P(w)$

$= \underset{w}{\arg\max} \left[\prod_{i=1}^{n} P[y_i | x_i, w]\; P(x_i)\right] \cdot P(w).$ (as x_i independant of w).

$= \underset{w}{\arg\max} \sum_{i=1}^{n} \left[\log P(y_i | x_i, w) + \log P(w)\right] \rightarrow$ (as $P(x_i)$ is constant)

$= \underset{w}{\arg\max} \sum_{i=1}^{n} \left[\log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + \log\left(e^{\frac{-(x_i^T w - y_i)^2}{2\sigma^2}}\right)\right] + \left[\log\left(\frac{-w^2}{2\tau^2}\right) + \log\left(\frac{1}{\sqrt{2\pi\tau^2}}\right)\right]$

=

$= \underset{w}{\arg\min} \frac{1}{2\sigma^2} \sum_{i=1}^{n} (x_i^T w - y_i)^2 + \frac{1}{2\tau^2} w^T w$]

as $\log\left(\frac{1}{\sqrt{2\pi\tau^2}}\right)$ is constant so ignored.

$$\boxed{w = \underset{w}{\arg\min} \frac{1}{n} \sum_{i=1}^{n} (x_i^T w - y_i)^2 + \lambda (w^T w)}$$ (where $\lambda = \frac{\sigma^2}{n\tau^2}$).