Machine Learning

Assignment 1

Sahil Goyal

2020326

1.

Machine Learning
Assignment 1

1 We need to prove that for simple linear regression, the least square fit line always passes through the point $(\bar{x}, \bar{y})$ where $\bar{x}$ and $\bar{y}$ represent the arithmetic mean of the independant variables and dependant variables respectively.

We need to prove that $\bar{y} = B_1 \bar{x} + B_0$ where B_1 and B_0 are the parameters we have obtained from regression.

$$\hat{y} = B_0 + B_1 x$$

$$y = \hat{y} + \varepsilon \quad \text{(True value = Predicted value + error)}$$

∴ Let us assume we have n values

$$\therefore y_1 = \hat{y}_1 + \varepsilon_1$$
$$y_2 = \hat{y}_2 + \varepsilon_2$$
$$\vdots$$

$$y_n = \hat{y}_n + \varepsilon_n$$

Take sum of all equations

$$\sum_{i=1}^{n} y_i = \sum_{i=1}^{n} \hat{y}_i + \sum_{i=1}^{n} \varepsilon_i$$

$$\therefore \sum_{i=1}^{n} y_i = \sum_{i=1}^{n} (B_0 + B_1 x_i) + \sum_{i=1}^{n} \varepsilon_i \quad (\hat{y} = B_0 + B_1 x)$$

$$\therefore \sum_{i=1}^{n} y_i = \sum_{i=1}^{n} B_0 + \sum_{i=1}^{n} B_1 x_i + \sum_{i=1}^{n} \varepsilon_i$$

$$\therefore \sum_{i=1}^{n} y_i = nB_0 + B_1 \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} \varepsilon_i$$

Sum of mean square errors

$$= \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

$$\therefore MSE = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} (y_i - B_0 - B_1 x_i)^2$$

$$\therefore \frac{\partial (MSE)}{\partial B_0} = 0$$

(For MSE to be minimum its differential should be 0)

$$\therefore -2 \sum_{i=1}^{n} (y_i - B_1 x_i - B_0) = 0$$

$$\therefore 2 \sum_{i=1}^{n} \varepsilon_i = 0 \quad (\varepsilon_i = y_i - B_1 x_i - B_0)$$

$$\sum_{i=1}^{n} \varepsilon_i = 0$$

$$\sum_{i=1}^{n} y_i = n\beta_0 + \beta_1 \sum_{i=1} x_i + 0$$

$$\frac{1}{n}\sum_{i=1}^{n} y_i = \beta_0 + \beta_1 \left(\frac{\sum_{i=1} x_i}{n} \right)$$

$$\bar{y} = \beta_0 + \beta_1 \bar{x} \qquad \left(\begin{array}{l} \Sigma y_i/n = \bar{y} \\ \Sigma x_i/n = \bar{x} \end{array} \right)$$

* The least square fit line always passes through the point $(\bar{x}, \bar{y})$

Thus Proved

b) Let us take random variables A, B, C.

∴ We are given that A is highly correlated with C and B is also highly correlated with C

∴ $\rho_{AC} > 0$, $\rho_{BC} > 0$

We need to show that A and B will not necessarily be highly correlated.

We will find the value of correlation coefficient of A and B (ρ_{AB}) using partial correlation

$$\rho_{AB.C} = \frac{\rho_{AB} - \rho_{AC}\rho_{CB}}{\sqrt{1-\rho_{AC}^2}\sqrt{1-\rho_{CB}^2}}$$

$$\therefore \rho_{AB} = \rho_{AC}\rho_{CB} + \rho_{AB.C}\left(\sqrt{1-\rho_{AC}^2}\sqrt{1-\rho_{CB}^2}\right)$$

∴ We can see that $\rho_{AB.C}$ lies between the range of -1 and 1

$$\rho_{AB} \leq \rho_{AC}\rho_{CB} + \sqrt{1-\rho_{AC}^2}\sqrt{1-\rho_{CB}^2}$$

$$\rho_{AB} \geq \rho_{AC}\rho_{CB} - \sqrt{1-\rho_{AC}^2}\sqrt{1-\rho_{CB}^2}$$

Let $\rho_{AC} = 0.8$, $\rho_{CB} = 0.8$

$\therefore \rho_{AC}\rho_{CB} = 0.64$

$$\sqrt{1-\rho_{AC}^2}\sqrt{1-\rho_{CB}^2} = 0.36$$

$\therefore 0.28 \leq \rho_{AB} \leq 1$

$\therefore \rho_{AB} \in [0.28, 1]$

$\therefore$ We can see that A and B need not be highly correlated as value can reach 0.28

$\therefore$ Thus Proved

c) We need to prove the weak law of large numbers

Let us take n independant and identically distributed Random variables and let each have an expected value of μ.

Let S_n be the sum of these RV's

$$\therefore S_n = X_1 + X_2 \ldots X_n$$

$\therefore$ We need to perove that

$$\lim_{n\to\infty}\left(P\left(\left|\frac{S_n}{n} - \mu\right| \geq \varepsilon\right)\right) \to 0$$

for any $\varepsilon > 0$

$$S_n = \sum_{i=1}^{n} X_i$$

$$\overline{S}_n = S_n / n$$

$$\therefore \bar{\xi}_n = \frac{X_1 + X_2 + \ldots X_n}{n}$$

$$\therefore \bar{\xi}_n = \frac{n\mu}{n} = \mu$$

Now, we will use the Chebyshev Inequality to perove the weak law of large numbers

$$P\left(|\bar{\xi}_n - \mu| \geq \varepsilon\right) \leq \frac{Var(\bar{\xi}_n)}{\varepsilon^2}$$

where $Var(\bar{\xi}_n)$ denotes variance of $\bar{\xi}_n$

$$Var(\bar{\xi}_n) = Var\left(\frac{X_1 + X_2 \ldots X_n}{n}\right)$$

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

$$= \frac{1}{n^2}\left(n\, Var(X_i)\right) \quad (X_i \text{ are i.i.d RV})$$

$$= \frac{n\, Var(X_i)}{n^2}$$

Let $Var(X_i) = \sigma^2$

$$\therefore Var(\bar{\xi}_n) = \frac{\sigma^2}{n}$$

$$\therefore P(|\bar{x}_n - \mu| \geq \varepsilon) \leq \frac{\sigma^2/n}{\varepsilon^2}$$

$$\therefore \lim_{n\to\infty} P(|\bar{x}_n - \mu| \geq \varepsilon) \leq \lim_{n\to\infty} \frac{\sigma^2}{n\varepsilon^2}$$

$$\therefore \lim_{n\to\infty} P(|\bar{x}_n - \mu| \geq \varepsilon) \leq 0 \quad \left(\text{as } \lim_{n\to\infty} \frac{\sigma^2}{n\varepsilon^2} = 0\right)$$

We have proved the weak law of large numbers

We will now provide a pseudo code to illustrate the weak law of large numbers
We will use gaussian distribution with mean 0 and standard deviation of 5

```
function wlln():
  sum=0
  x=[]
  y=[]
  for i=10 to 100000:
    x.append(i)
    sum=sum+random.gauss(0,5)
    y.append(sum/i)
  plot(x,y)
```

Here random.gauss(0,5) will give us a random gaussian number with mean 0 and standard deviation 5.
We will be able to see in graph that the value of mean of RV's approaches 0 as the number of random gaussian numbers increases.
∴ This illustrates the weak law of large numbers

d) We need to derive the Maximum A posteriori solution for linear regression

$$y = \beta x + \varepsilon$$

Here ε is a normal distribution with mean 0 and variance σ^2

$$\varepsilon \sim \text{Normal}(0, \sigma^2)$$

$$y|x \sim \text{Normal}(\beta x, \sigma^2)$$

$$p(y_i|x_i) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i - \beta x_i)^2}{2\sigma^2}}$$

Now we can see that

$$\theta = \underset{\theta}{\operatorname{argmax}}\, p(\beta | y_1, x_1, y_2, x_2 \ldots y_n, x_n)$$

$$\theta = \underset{\theta}{\operatorname{argmax}}\, \frac{p(y_1, x_1 \ldots y_n, x_n | \beta)\, p(\beta)}{p(y_1, x_1 \ldots y_n, x_n)}$$

(Bayes theorem)

$$\left(P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \right)$$

My NoteBook

$\therefore$ argmax $P(y_1, x_1 \ldots y_n, x_n \mid w) P(w)$

(denominator doesn't contain w)

Let us assume θ to be a normal distribution

$\theta \sim N(\vec{0}, \sigma_2^2)$

$\therefore P(\theta) = \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{-\frac{1}{2}\left(\frac{\theta}{\sigma_2}\right)^2}$

$P(\theta) = \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{-\frac{\theta^2}{2\sigma_2^2}}$

$\therefore P(y_1, x_1 \ldots y_n, x_n \mid \beta) = \prod_{i=1}^{n} P(y_i, x_i \mid \beta)$

(They are i i d)

$P(y_1, x_1 \dots y_n, x_n \mid \beta)$

$= \prod_{i=1}^{n} \left(P(y_i \mid x_i, \beta) \, P(x_i \mid \beta) \right)$

$\theta = \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{n} P(y_i \mid x_i, \beta) \, P(x_i \mid \beta) P(\beta)$

$\theta = \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{n} \left(P(y_i \mid x_i, \beta) \, P(x_i) \right) P(\beta)$

(x_i is independant of β)

$= \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{n} \left(\log(P(y_i \mid x_i, \beta)) + \log P(\beta) \right)$

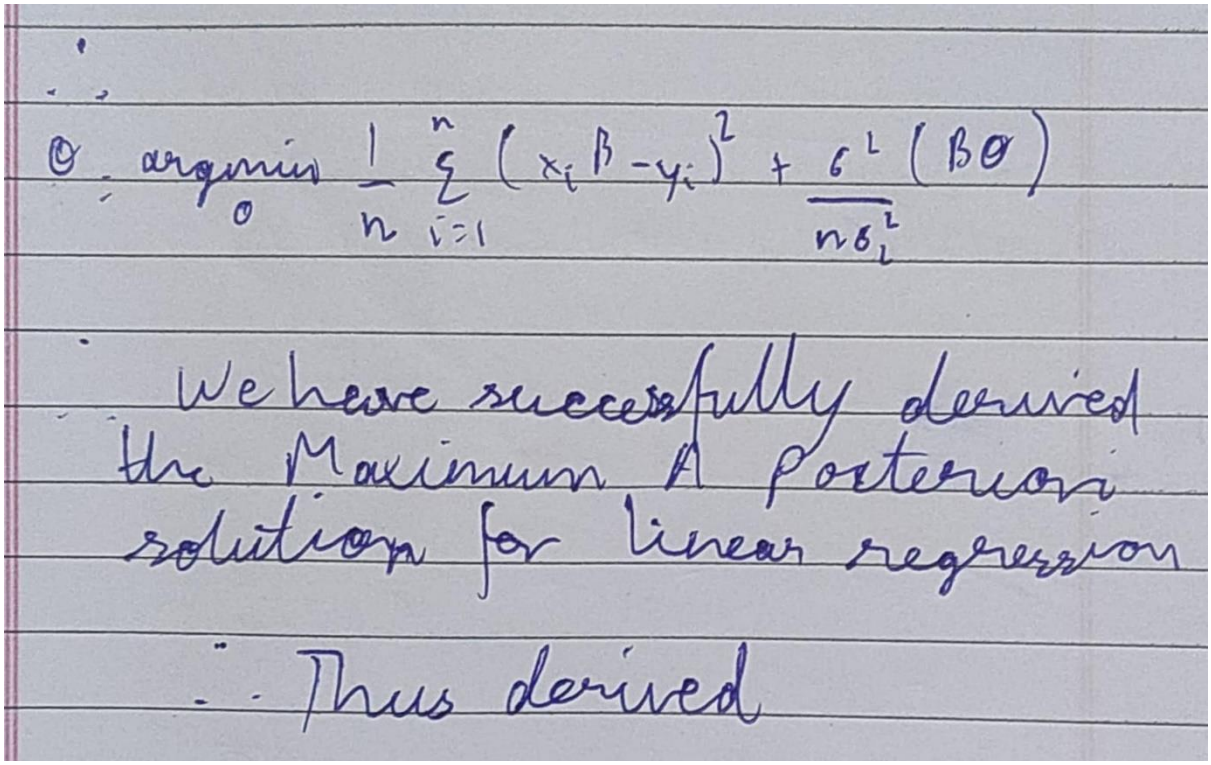
(As $P(x_i)$ is constant)

$= \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{n} \left(\log\left(\frac{1}{\sqrt{2\pi\sigma^2}} \right) + \log\left(e^{\frac{-(y_i - x_i\beta)^2}{2\sigma_i^2}} \right) \right)$

$+ \log\left(\frac{-\theta^2}{2\sigma_0^2} \right) + \log\left(\frac{1}{\sqrt{2\pi\sigma_0^2}} \right)$

$= \underset{\theta}{\operatorname{argmin}} \frac{1}{2\sigma^2} \sum_{i=1}^{n} (x_i\beta - y_i)^2 + B\theta$

($\frac{1}{\sqrt{2\pi\sigma_i^2}}$ is constant so it can be ignored safely)



2.

a.

In this problem, we have implemented k fold cross validation. We have implemented it for values of k being 2, 3, 4, and 5.

Explanation of Code:

- First, we have used some functions to get a basic idea about our data. We have used head() to see the first five rows, dtypes to see the type of the data in each column, describe() to get various information about our data, info() to understand data like number of rows, null values, etc, isnull().sum() t check for null values, shape to see dimensions of the dataframe, duplicated().sum() to see the number of duplicate rows. Then, we use iloc function to separate the data into input and output columns, and then we use head function to see the first 5 rows.
- First, we loop through all the values of k (2 to 5). After that, we first separate the data into training and testing sets, and then we standardize the data, and then we call the function for gradient descent on it. Then, we use the output to find the root mean square error for every fold, and finally, after calculating it, we add it to the average_error variable that is storing our average error across all the folds for this particular value of k.
- At the end of each iteration of k, before we go to the next value of k, we print the average error for that particular value of k.

Results:

```
average error for k= 2 is 8.88962794325466
average error for k= 3 is 8.859713461305743
average error for k= 4 is 8.819017844323731
average error for k= 5 is 8.717990441837383
```

Result:

1. The optimal value of k is 5. As we increase the number of folds, the average error decreases.
2. This is because as the number of folds increase, the amount of training data also increases, and the accuracy of the model increases.

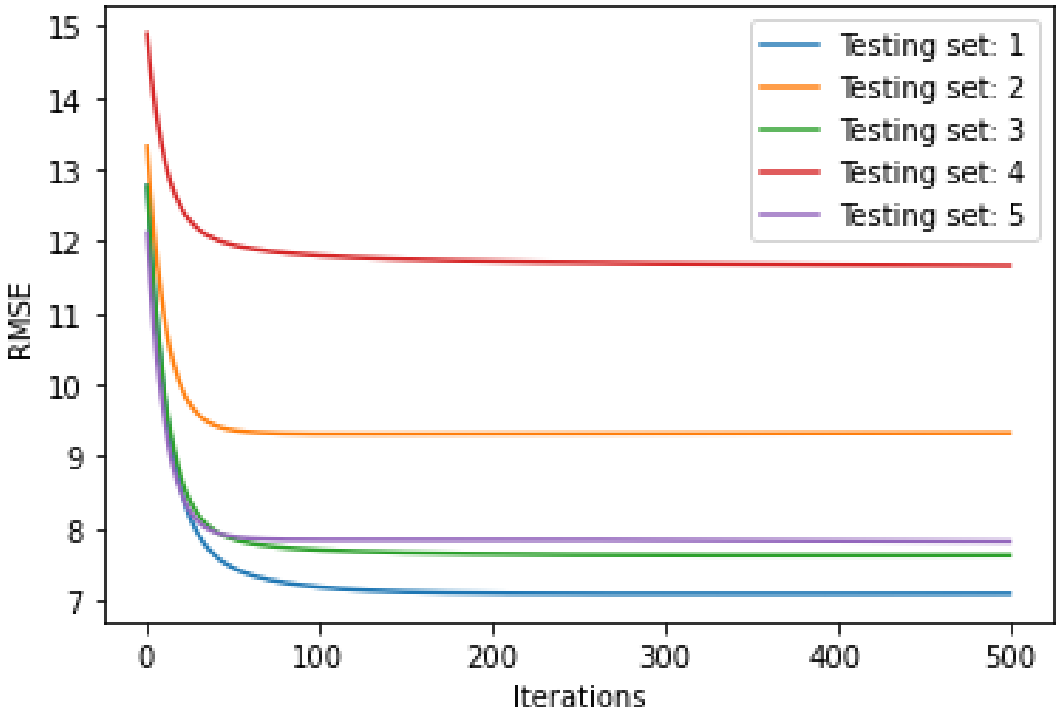
b.

From the last part, we have found the optimal value of k to be 5. Therefore, we will plot the graph of RMSE vs Iteration graph

Explanation of Code:

- First, we create two lists that will be there for storing the x coordinates, and the y coordinates so that we can plot them later on. After that, we first separate the data into training and testing sets, and then we standardize the data, and then we perform gradient descent on it. Then, we calculate the loss value and add that to the list of y coordinates.
- Finally, we plot the graph for RMSE vs Iterations.

Graph:



Result:

Results:

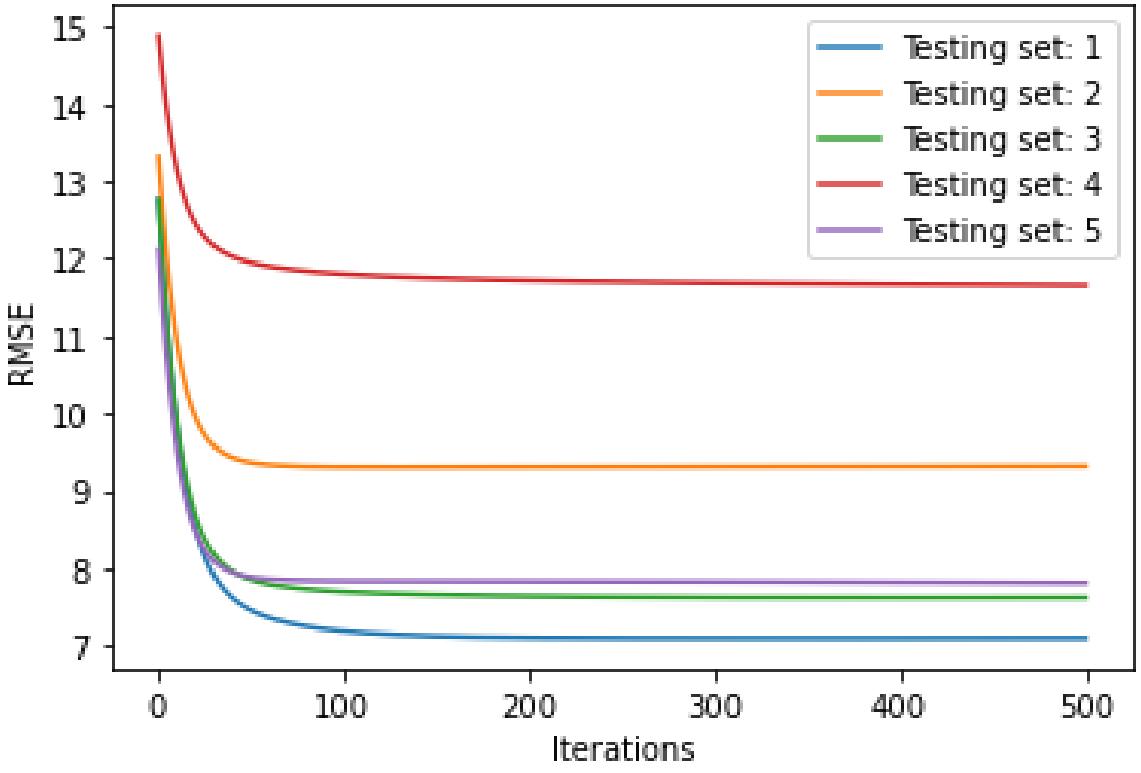
1. The RMSE decreases as the number of iterations increases.
2. This is because as the number of iterations increases, we approach closer and closer to the minima of the loss function.
3. We can see that the RMSE value is inversely proportional to the number of iterations.

c.

Explanation of Code:

- First, we will plot the Lasso regularization implementation. This is being implemented because it helps prevent overfitting of our data.
- First, we separate the data into training and testing sets. Here, in LASSO implementation, we create an extra variable d which stands for deviation. After that, we standardise the data, and then perform gradient descent. For LASSO implementation, we add an absolute value as a penalty term to the loss function. This is done to reduce overfitting. Then, we update the value of d and weights. After that, we calculate the loss and then add the value to the coordinates so that we can plot it. Finally, we add the value to our average_error variable as well. This variable stores the average error.
- Finally, we plot the RMSE vs Iterations graph and then show the average error.

Graph:



Results:

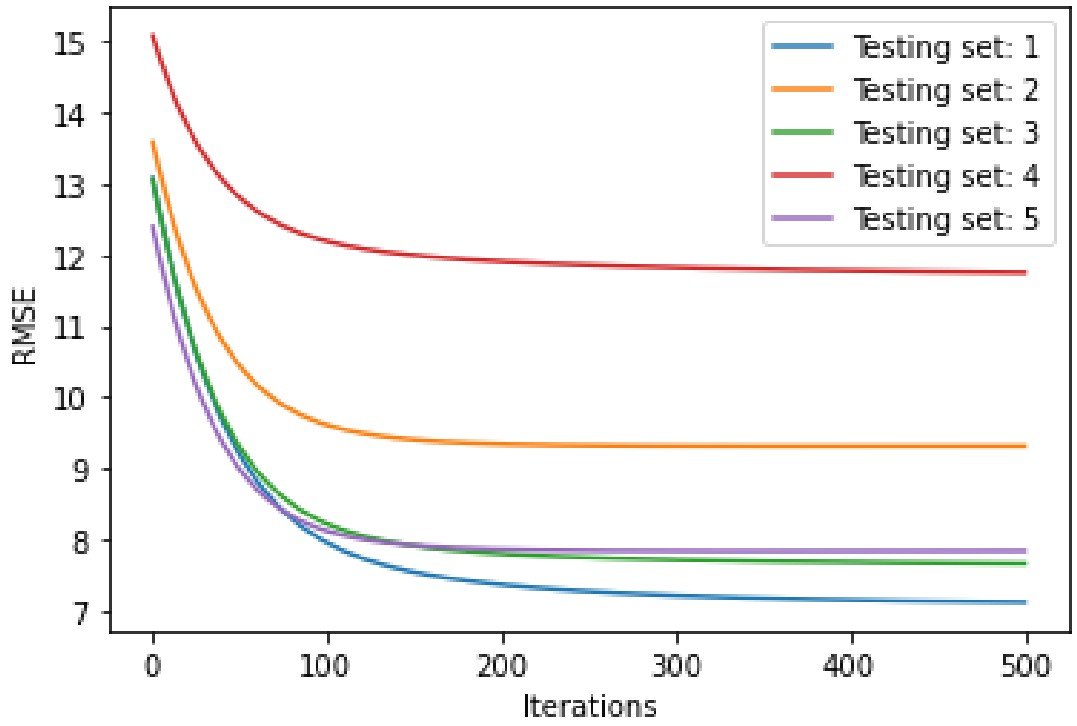
```
Average error:  8.704643056926518
```

Results:

1. The RMSE decreases as the number of iterations increases.
2. We can see that the RMSE value is inversely proportional to the number of iterations.
3. The average error is less than what it was without regularization.

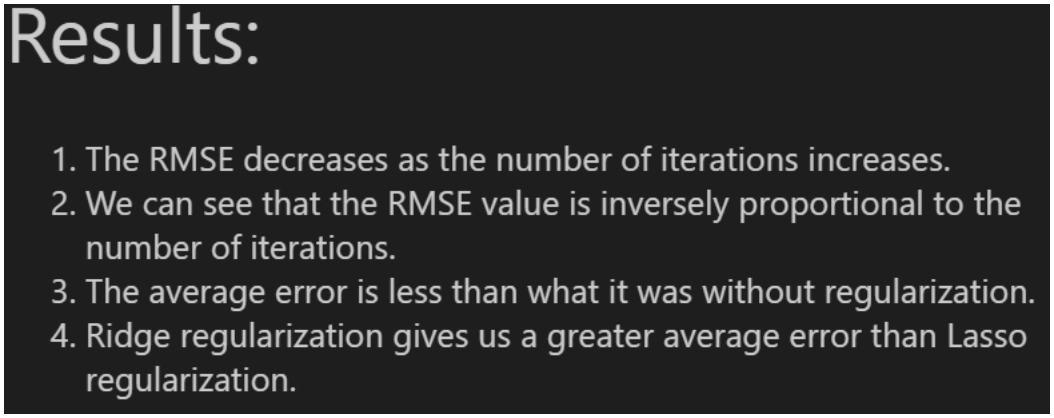
- Then, we will plot the Ridge regularization implementation. This is being implemented because it helps prevent overfitting of our data.
- First, we separate the data into training and testing sets. We create an extra variable d which stands for deviation. After that, we standardise the data, and then perform gradient descent. Similarly, to LASSO implementation, a penalty is applied to the loss function in ridge regularization. Then, we update the value of d and weights. After that, we calculate the loss and then add the value to the coordinates so that we can plot it. Finally, we add the value to our average_error variable as well. This variable stores the average error.
- Then, we plot the RMSE vs Iterations graph and show the average error.

Graph:



Result:

```
Average error:  8.738246761640388
```



d.

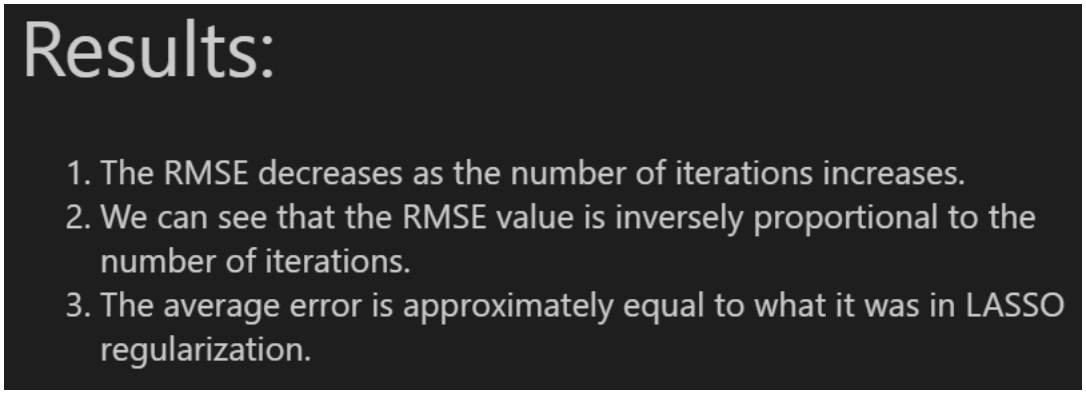
Explanation of Code:

We implement the normal equation to get the optimal parameters directly for each fold, and then we show the average error.

- First, we split the data into training and testing sets. After that we standardise the data and convert the data into numpy arrays so that we can use numpy functions for matrix operations. We use the normal equation which gives us the value of the weights, and then use that to calculate the loss value for each fold. We then add it to the average_error value, which is used to store the average error.
- Finally, we show the average error.

Result:

```
Average error:  8.70407793548543
```



3.

a.

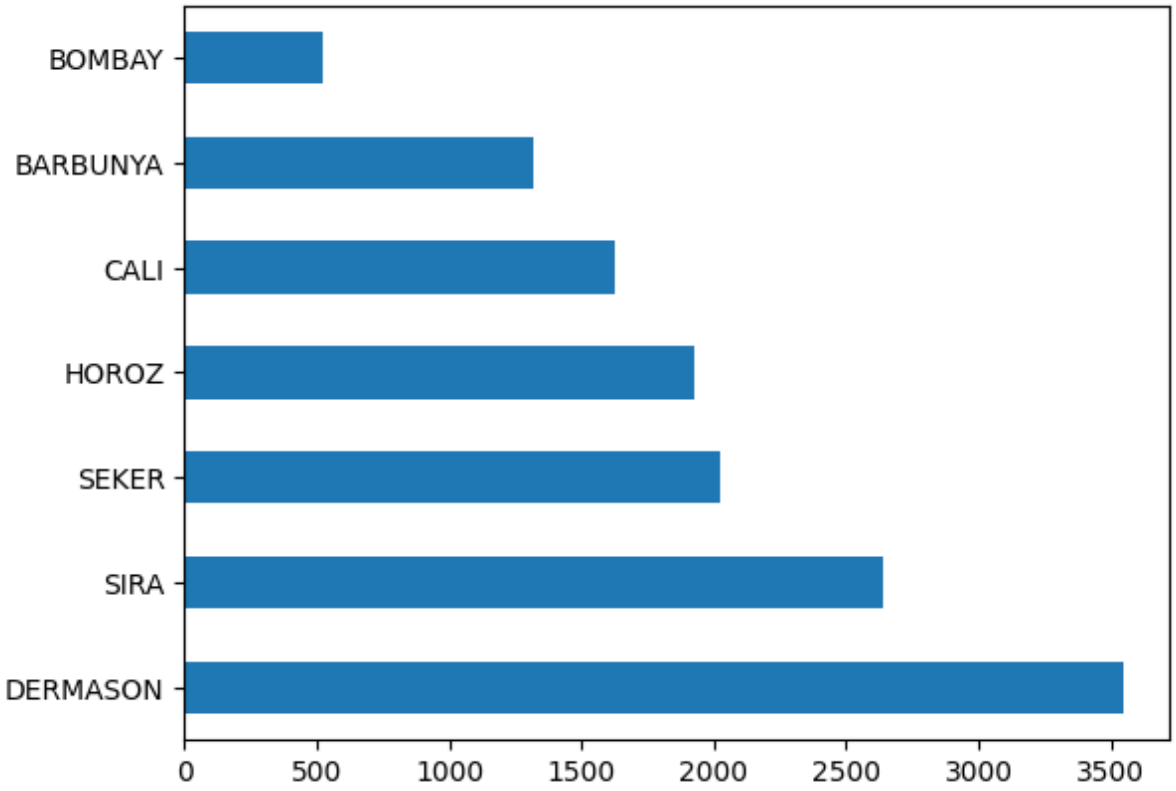
Explanation of Code:

- First, we have imported the necessary libraries. We are using numpy, matplotlib, pandas, and seaborn
- Then we read the data using pd.read_excel(). This gives us a dataframe.
- Then we run a few basic commands like describe, head, shape, to get a basic overview of the data.

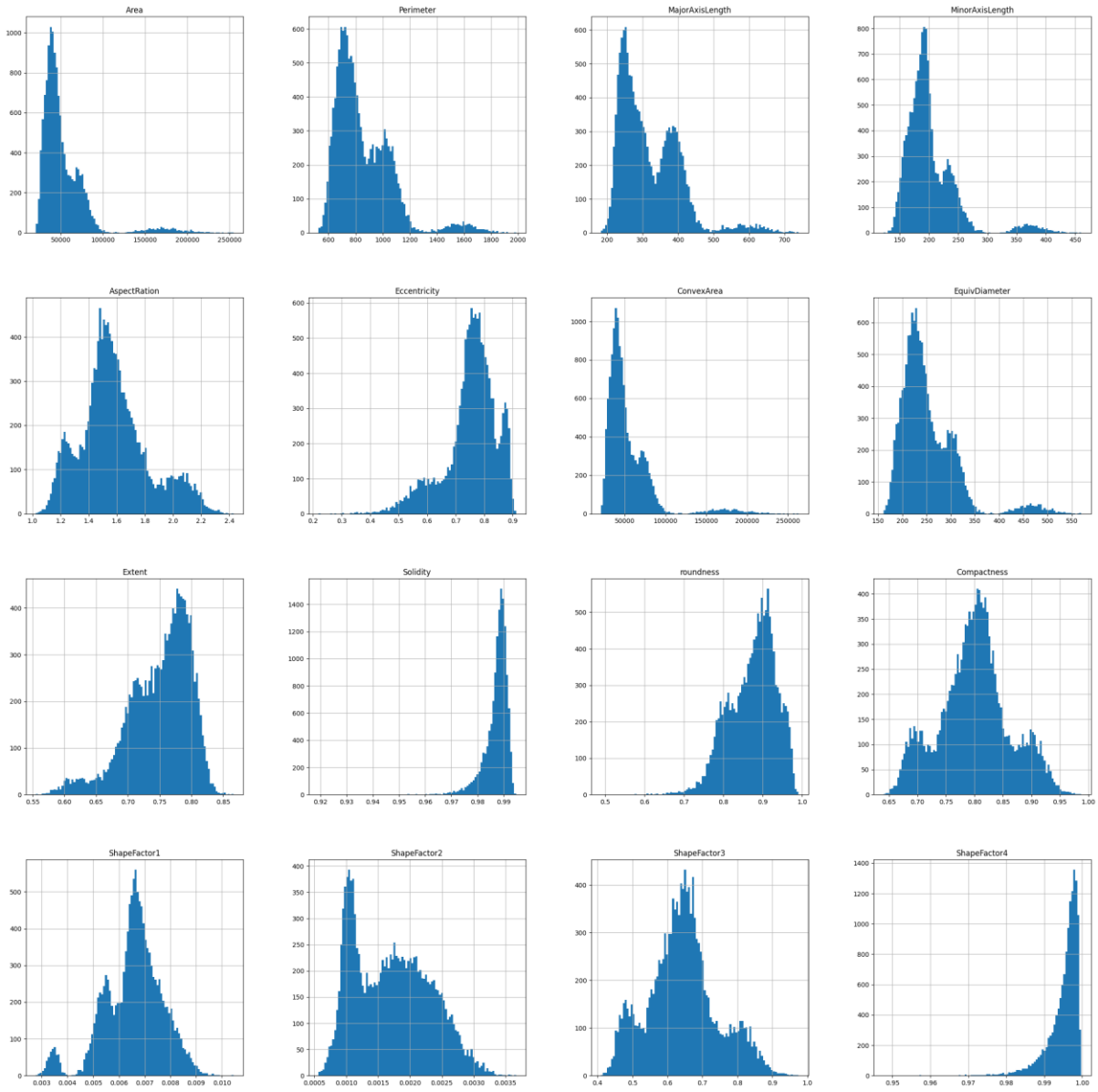
- Then we plot a horizontal bar graph of the data which shows us the class distribution.
- Then we plot a histogram of the 16 columns of the data.

Results:

Class distribution:



Histogram:



ANALYSIS:

1. We can see that the perimeter of the beans is directly proportional to the area

2. The maximum number of the beans have an area of around 40000

3. We can see that the largest number of beans have a class of DERMASON and the smallest number of beans have a class of BOMBAY. The frequency of beans increases in the order of Bombay, Barbunya, Cali, Horoz, Seker, Sira, Dermason.

b.

Explanation of the Code:

- First, we perform a few basic functions on the data to understand more about the columns like the number of rows, number of columns, types of the data, average, etc.
- The functions we run are info(), describe(), shape, duplicated().sum(), isnull().sum(), dtypes, corr()

- Then we plot histograms, scatter plots, box plots, and heatmaps to visually understand the data.

Results of the data:

Info function:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13611 entries, 0 to 13610
Data columns (total 17 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   Area             13611 non-null  int64  
 1   Perimeter        13611 non-null  float64
 2   MajorAxisLength  13611 non-null  float64
 3   MinorAxisLength  13611 non-null  float64
 4   AspectRation     13611 non-null  float64
 5   Eccentricity     13611 non-null  float64
 6   ConvexArea       13611 non-null  int64  
 7   EquivDiameter    13611 non-null  float64
 8   Extent           13611 non-null  float64
 9   Solidity         13611 non-null  float64
 10  roundness        13611 non-null  float64
 11  Compactness      13611 non-null  float64
 12  ShapeFactor1     13611 non-null  float64
 13  ShapeFactor2     13611 non-null  float64
 14  ShapeFactor3     13611 non-null  float64
 15  ShapeFactor4     13611 non-null  float64
 16  Class            13611 non-null  object 
dtypes: float64(14), int64(2), object(1)
memory usage: 1.8+ MB
```

Describe function:

	Area	Perimeter	MajorAxisLength	MinorAxisLength	AspectRation	Eccentricity	ConvexArea	EquivDiameter	Extent	Solidity
count	13611.000000	13611.000000	13611.000000	13611.000000	13611.000000	13611.000000	13611.000000	13611.000000	13611.000000	13611.000000
mean	53048.284549	855.283459	320.141867	202.270714	1.583242	0.750895	53768.200206	253.064220	0.749733	0.987143
std	29324.095717	214.289696	85.694186	44.970091	0.246678	0.092002	29774.915817	59.177120	0.049086	0.004660
min	20420.000000	524.736000	183.601165	122.512653	1.024868	0.218951	20684.000000	161.243764	0.555315	0.919246
25%	36328.000000	703.523500	253.303633	175.848170	1.432307	0.715928	36714.500000	215.068003	0.718634	0.985670
50%	44652.000000	794.941000	296.883367	192.431733	1.551124	0.764441	45178.000000	238.438026	0.759859	0.988283
75%	61332.000000	977.213000	376.495012	217.031741	1.707109	0.810466	62294.000000	279.446467	0.786851	0.990013
max	254616.000000	1985.370000	738.860153	460.198497	2.430306	0.911423	263261.000000	569.374358	0.866195	0.994677

Shape:

```
(13611, 17)
```

```
df1.duplicated().sum()
```

✓ 0.1s

```
68
```

```
df1.isnull().sum()
```

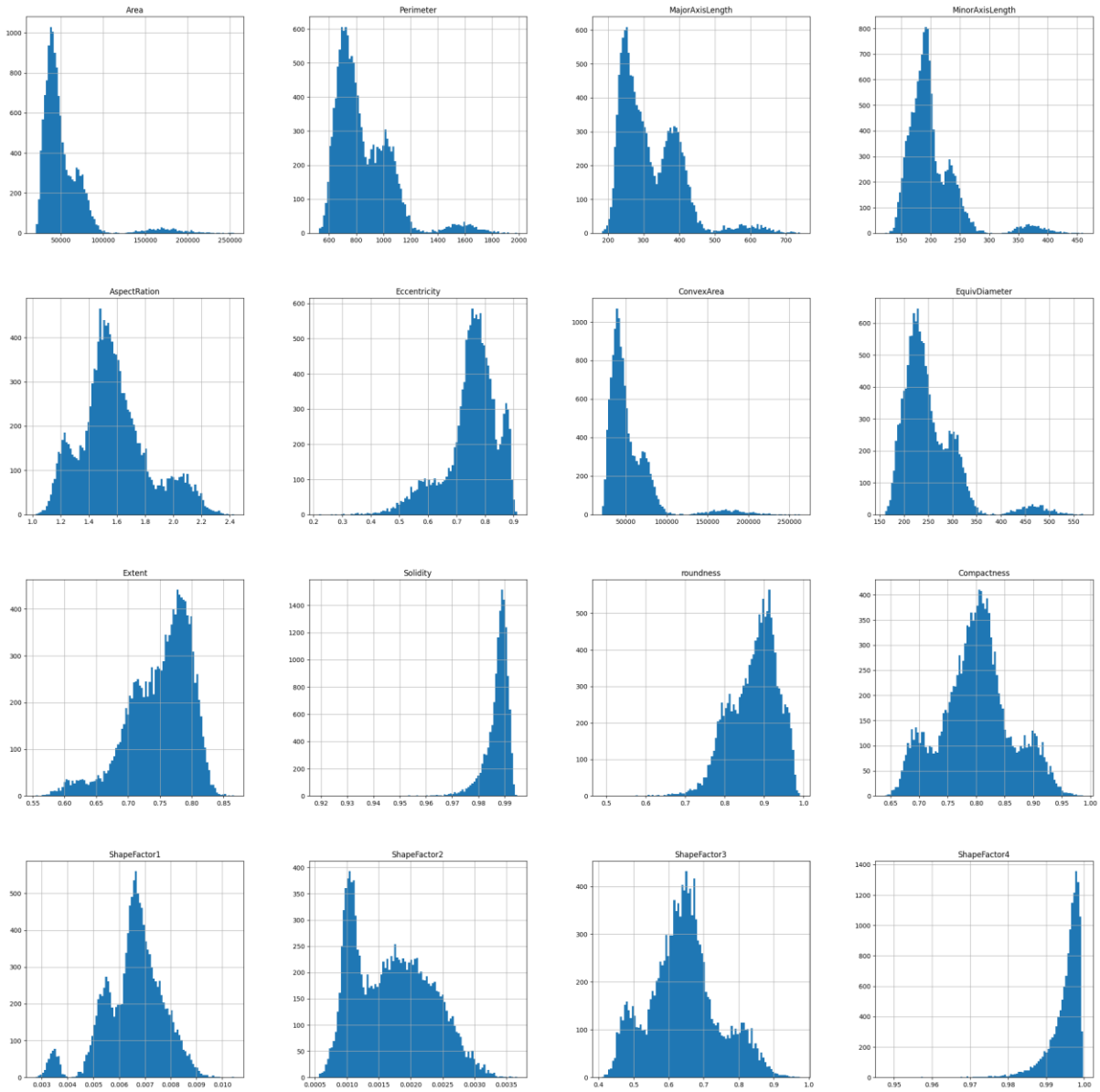
✓ 0.1s

```
Area               0
Perimeter          0
MajorAxisLength    0
MinorAxisLength    0
AspectRation       0
Eccentricity       0
ConvexArea         0
EquivDiameter      0
Extent             0
Solidity           0
roundness          0
Compactness        0
ShapeFactor1       0
ShapeFactor2       0
ShapeFactor3       0
ShapeFactor4       0
Class              0
dtype: int64
```

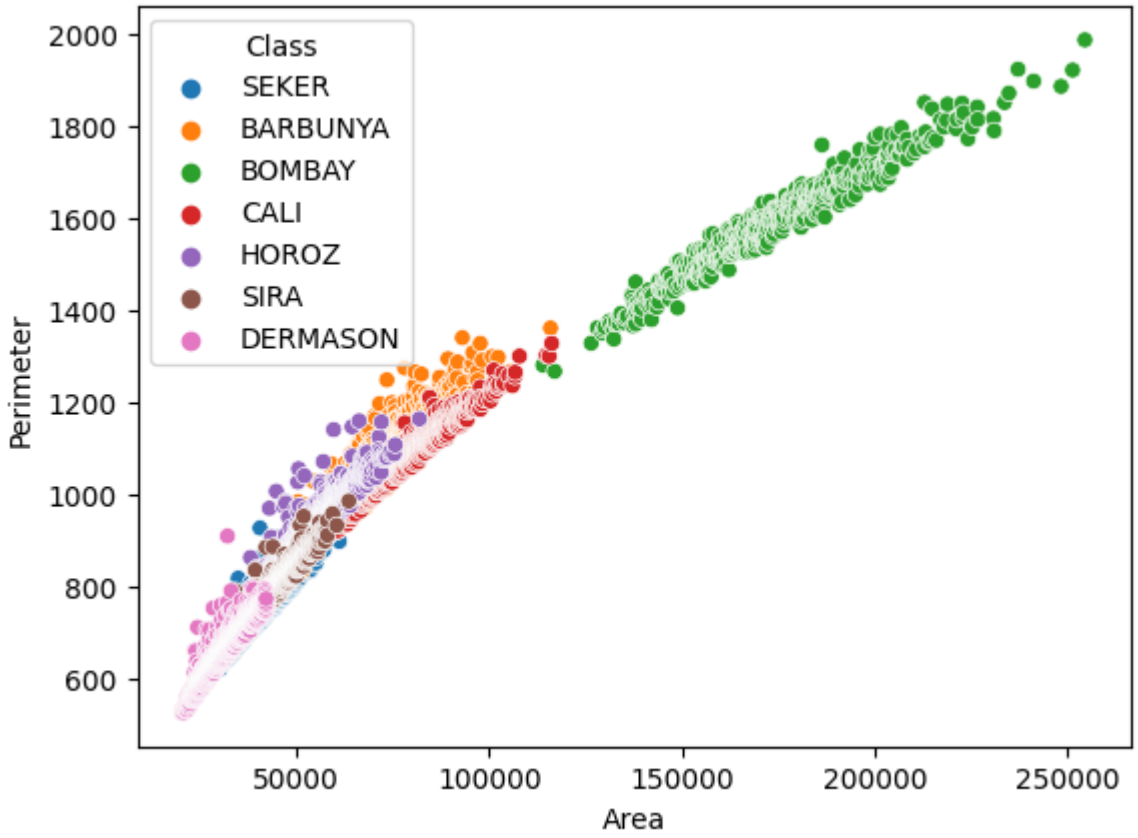
```
df1.dtypes
```
✓ 0.7s

```
Area                int64
Perimeter         float64
MajorAxisLength   float64
MinorAxisLength   float64
AspectRation      float64
Eccentricity      float64
ConvexArea          int64
EquivDiameter     float64
Extent            float64
Solidity          float64
roundness         float64
Compactness       float64
ShapeFactor1      float64
ShapeFactor2      float64
ShapeFactor3      float64
ShapeFactor4      float64
Class              object
dtype: object
```

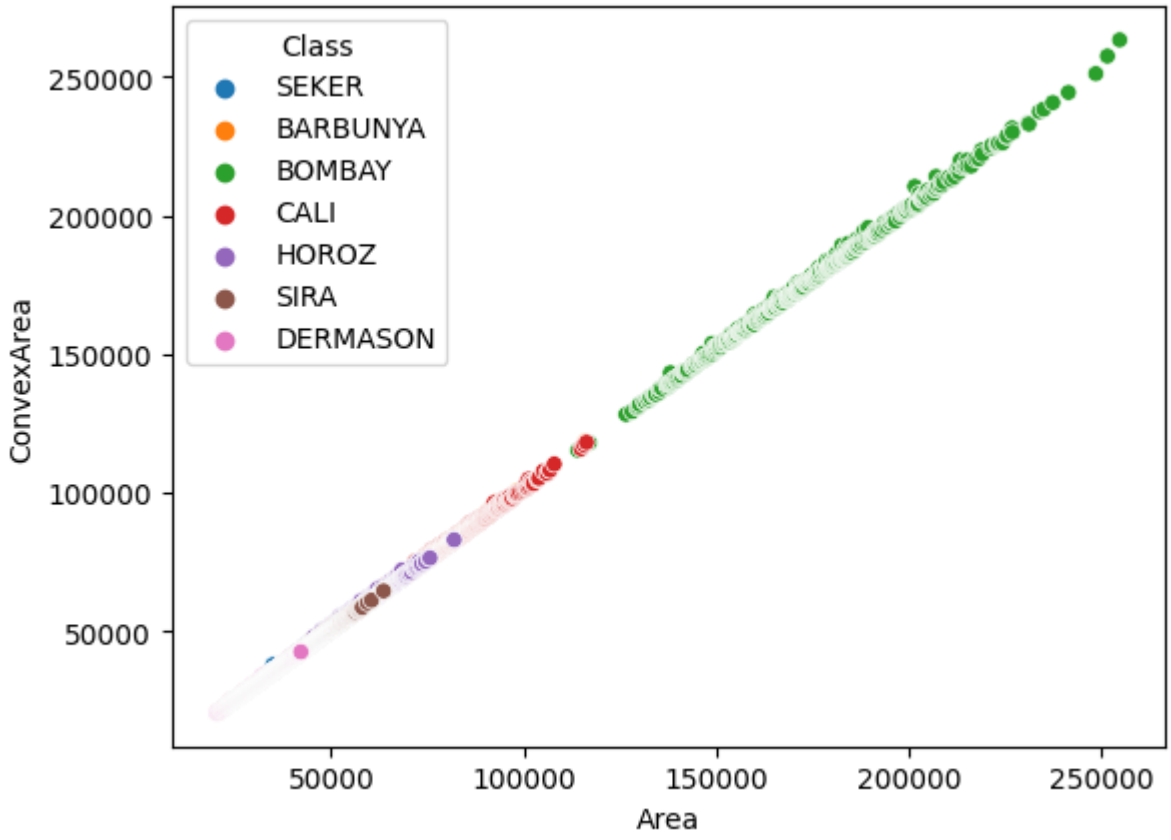
Histograms:



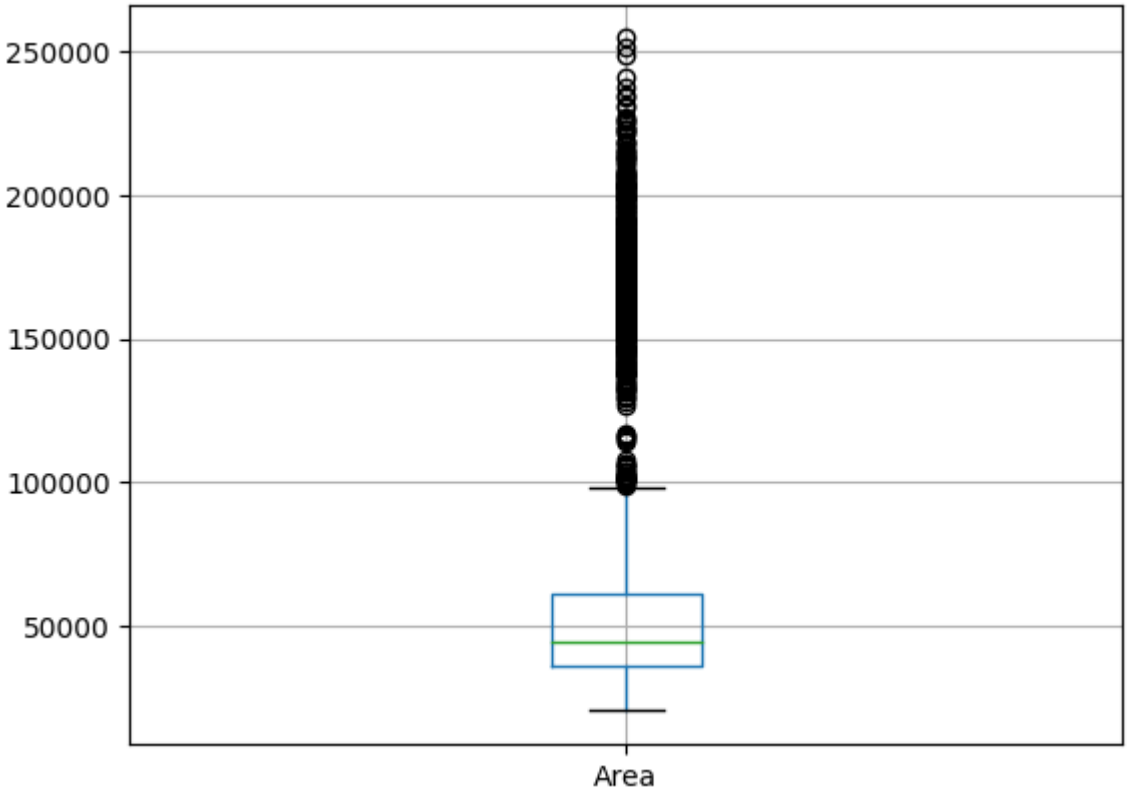
Area vs perimeter scatter plot:

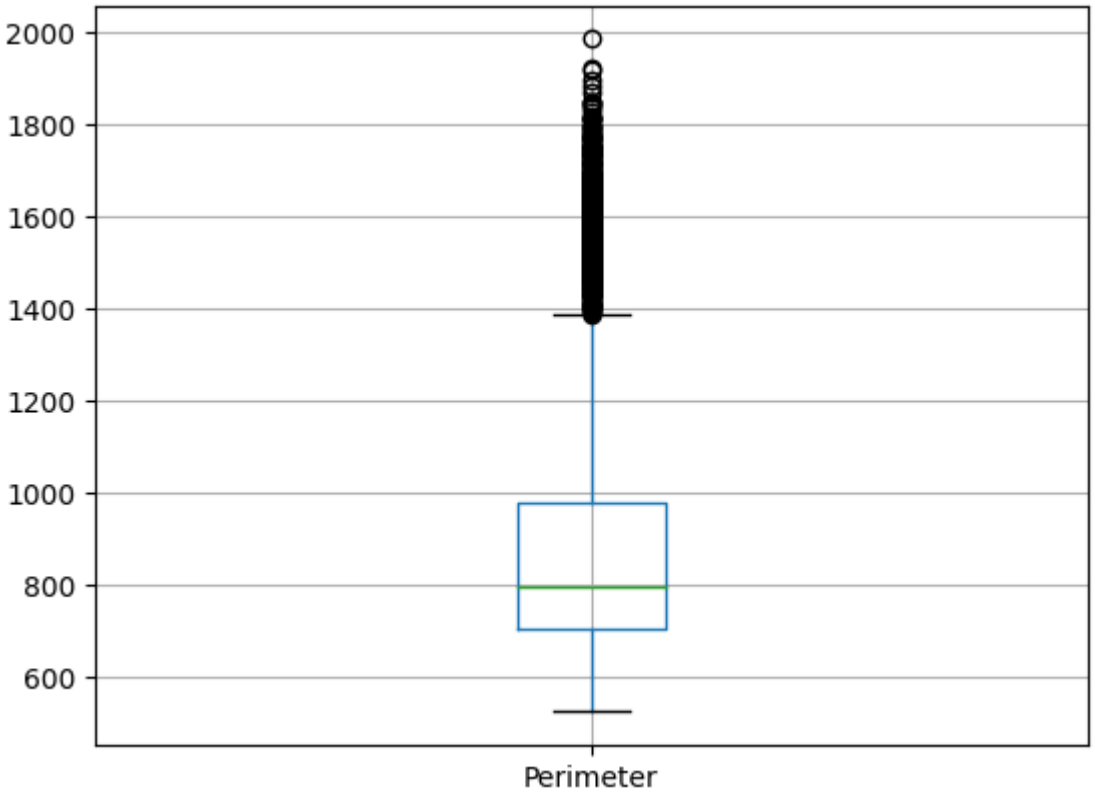


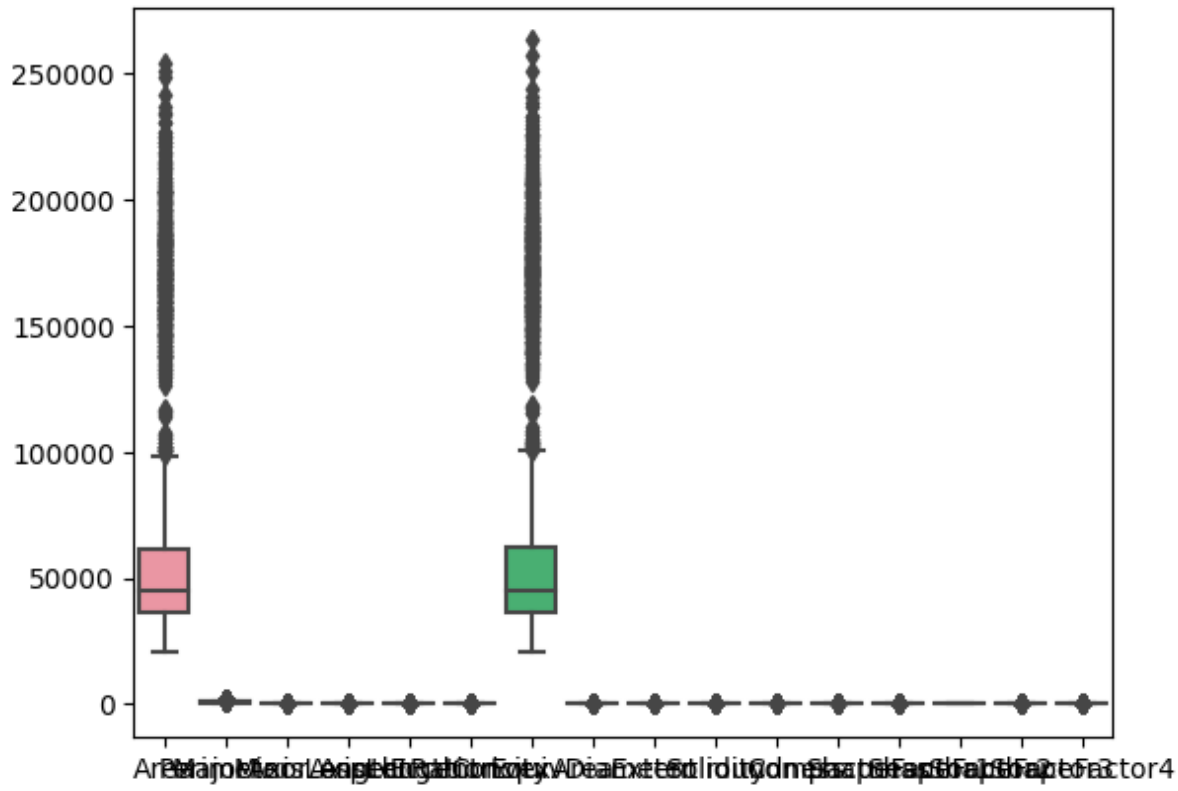
Area Vs ConvexArea scatterplot:



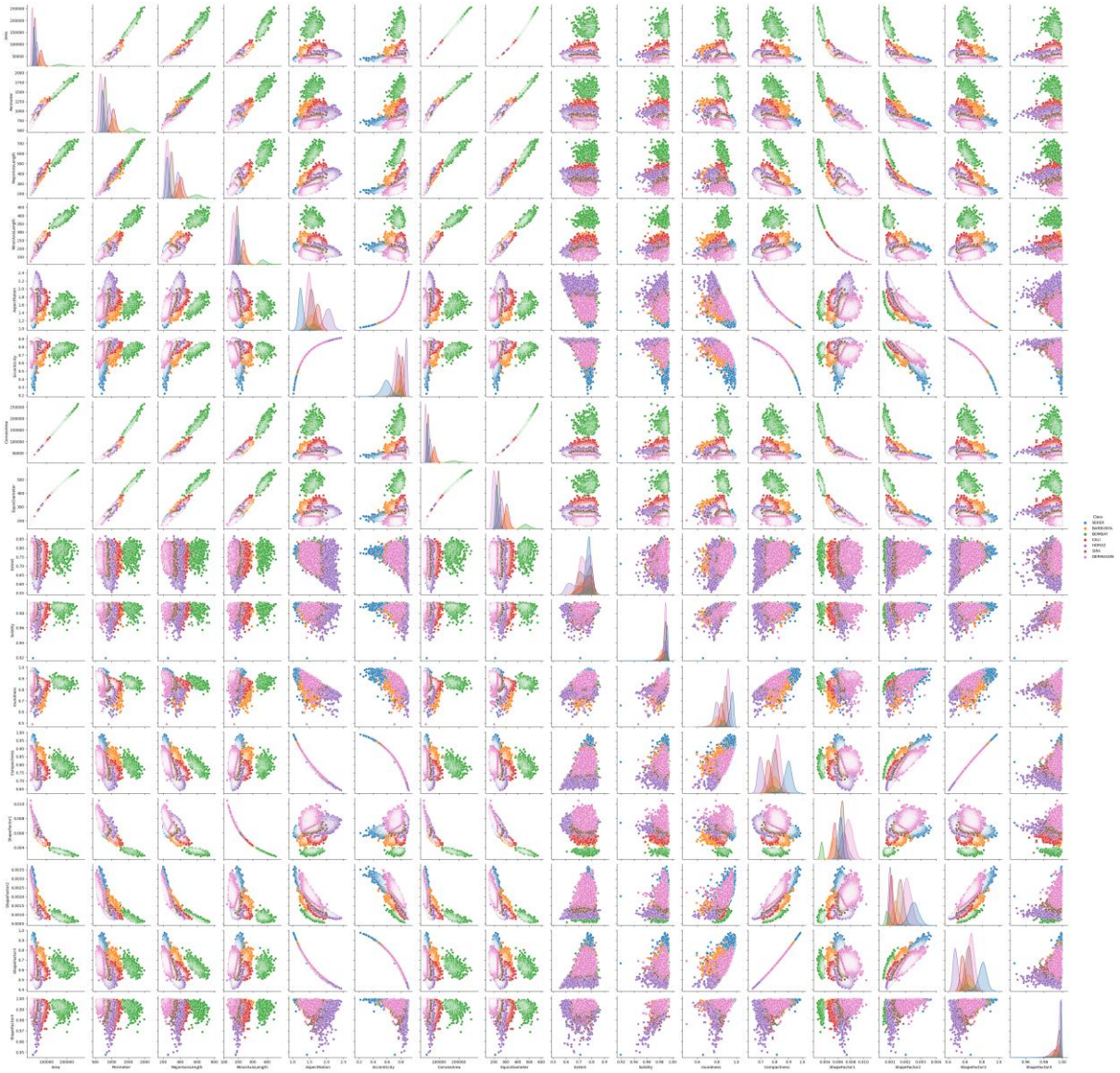
Boxplots:


250000
200000
150000
100000
50000
Area


2000
1800
1600
1400
1200
1000
800
600
Perimeter



Pair Plot:



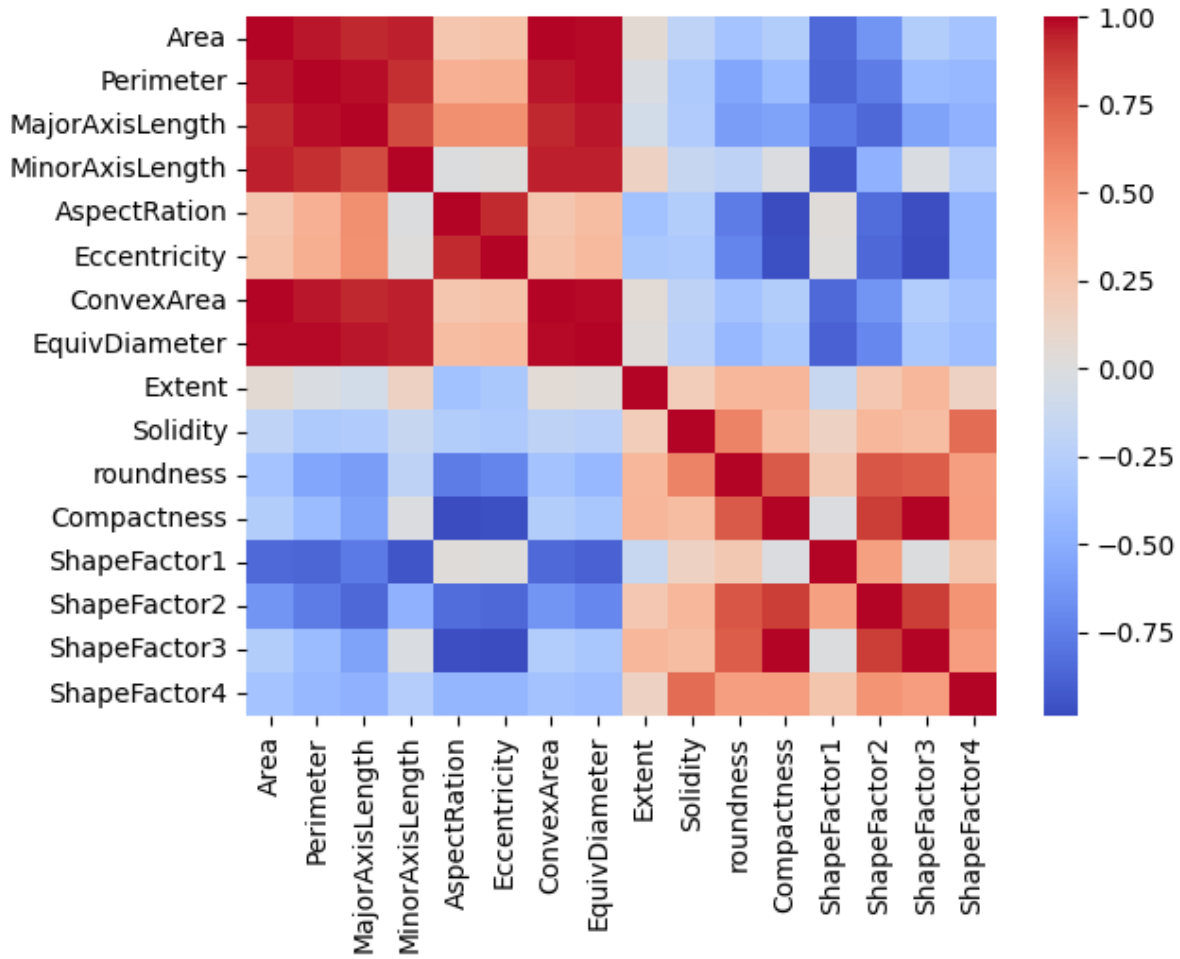
Correlation Matrix:

```
df1.corr()
```

✓ ✓ 0.5s Python

	Area	Perimeter	MajorAxisLength	MinorAxisLength	AspectRation	Eccentricity	ConvexArea	EquivDiameter	Extent	Solidity	roundnes
Area	1.000000	0.966722	0.931834	0.951602	0.241735	0.267481	0.999939	0.984968	0.054345	-0.196585	-0.35753
Perimeter	0.966722	1.000000	0.977338	0.913179	0.385276	0.391066	0.967689	0.991380	-0.021160	-0.303970	-0.54764
MajorAxisLength	0.931834	0.977338	1.000000	0.826052	0.550335	0.541972	0.932607	0.961733	-0.078062	-0.284302	-0.59635
MinorAxisLength	0.951602	0.913179	0.826052	1.000000	-0.009161	0.019574	0.951339	0.948539	0.145957	-0.155831	-0.21034
AspectRation	0.241735	0.385276	0.550335	-0.009161	1.000000	0.924293	0.243301	0.303647	-0.370184	-0.267754	-0.76697
Eccentricity	0.267481	0.391066	0.541972	0.019574	0.924293	1.000000	0.269255	0.318667	-0.319362	-0.297592	-0.72227
ConvexArea	0.999939	0.967689	0.932607	0.951339	0.243301	0.269255	1.000000	0.985226	0.052564	-0.206191	-0.36208
EquivDiameter	0.984968	0.991380	0.961733	0.948539	0.303647	0.318667	0.985226	1.000000	0.028383	-0.231648	-0.43594
Extent	0.054345	-0.021160	-0.078062	0.145957	-0.370184	-0.319362	0.052564	0.028383	1.000000	0.191389	0.34441
Solidity	-0.196585	-0.303970	-0.284302	-0.155831	-0.267754	-0.297592	-0.206191	-0.231648	0.191389	1.000000	0.60715
roundness	-0.357530	-0.547647	-0.596358	-0.210344	-0.766979	-0.722272	-0.362083	-0.435945	0.344411	0.607150	1.00000
Compactness	-0.268067	-0.406857	-0.568377	-0.015066	-0.987687	-0.970313	-0.269922	-0.327650	0.354212	0.303766	0.76808
ShapeFactor1	-0.847958	-0.864623	-0.773609	-0.947204	0.024593	0.019920	-0.847950	-0.892741	-0.141616	0.153388	0.23027
ShapeFactor2	-0.639291	-0.767592	-0.859238	-0.471347	-0.837841	-0.860141	-0.640862	-0.713069	0.237956	0.343559	0.78282
ShapeFactor3	-0.272145	-0.408435	-0.568185	-0.019326	-0.978592	-0.981058	-0.274024	-0.330389	0.347624	0.307662	0.76312
ShapeFactor4	-0.355721	-0.429310	-0.482527	-0.263749	-0.449264	-0.449354	-0.362049	-0.392512	0.148502	0.702163	0.47214

Heat Map:



Insights into the Data:

1. We can see that the sum of the duplicate values is 68. This means that there are 68 duplicate values in the dataset.
2. We can see that there are no null values in the dataset. Therefore, we have confirmed that there are no missing values in the dataset.
3. Most of the columns in the dataset are either int or float. The only column that is not int or float is the class column. There are 14 columns that are of float type, 2 columns that are of int type, and 1 column that is of object type
4. We can see that the correlation between the area and other categories like perimeter, major axis length is very high. This means that these categories are highly correlated. On the other hand correlation between the area and the shape factors, extent, solidity is very low. This means that these categories are negatively correlated. In the heatmap, we have chosen a coolwarm color map. This means that elements with a positive correlation are red and elements with a negative correlation are blue.
5. We can see that the area is directly proportional to the perimeter
6. We can see that area and perimeter have a lot of outliers through the boxplot
7. We can see that area is directly proportional to convex are. Area and Convex Area also have a very high correlation.
8. We can see that the Bombay beans have largest area and perimeter and the dermason beans have smallest area and perimeter

c.

Explanation of Code:

- First, we import the necessary libraries. We import TSNE and StandardScaler
- We use head(4) to print out the first four rows in the dataframe.
- First, we take another variable and assign it to the Class column (target column) of the dataframe. Then, we assign the rest of the dataframe (input columns)to another variable.
- Then we declare our TSNE model.
- We use fit transform to fit the input columns to the TSNE model
- Then we use np.vstack() and pd.Dataframe() to create a new variable and assign it to a new variable. This is done because using a dataframe will help us in plotting the result. The dataframe has three columns, Dim_1, Dim_2, and the target column which is Class

- Then we plot a scatterplot of the data.

Results:

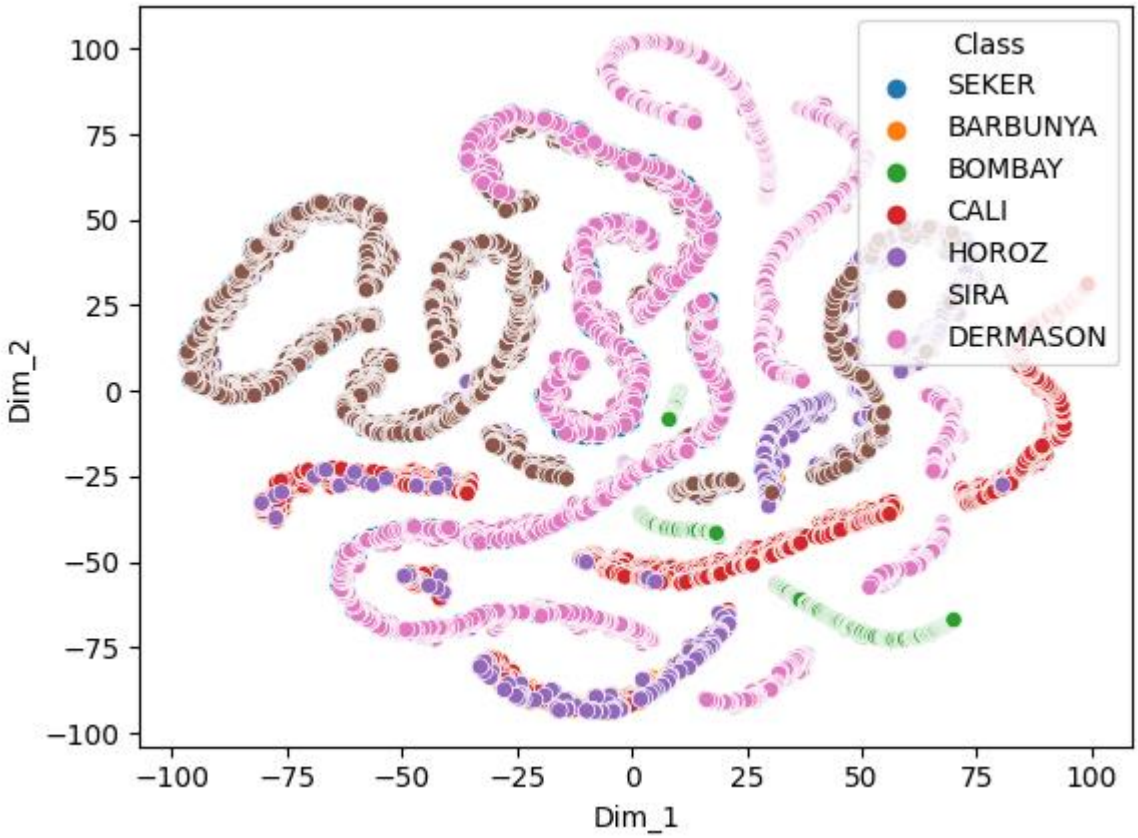
Head function:

	Area	Perimeter	MajorAxisLength	MinorAxisLength	AspectRation	\
0	28395	610.291	208.178117	173.888747	1.197191	
1	28734	638.018	200.524796	182.734419	1.097356	
2	29380	624.110	212.826130	175.931143	1.209713	
3	30008	645.884	210.557999	182.516516	1.153638	

	Eccentricity	ConvexArea	EquivDiameter	Extent	Solidity	roundness	\
0	0.549812	28715	190.141097	0.763923	0.988856	0.958027	
1	0.411785	29172	191.272750	0.783968	0.984986	0.887034	
2	0.562727	29690	193.410904	0.778113	0.989559	0.947849	
3	0.498616	30724	195.467062	0.782681	0.976696	0.903936	

	Compactness	ShapeFactor1	ShapeFactor2	ShapeFactor3	ShapeFactor4	Class
0	0.913358	0.007332	0.003147	0.834222	0.998724	SEKER
1	0.953861	0.006979	0.003564	0.909851	0.998430	SEKER
2	0.908774	0.007244	0.003048	0.825871	0.999066	SEKER
3	0.928329	0.007017	0.003215	0.861794	0.994199	SEKER

Tsne scatterplot:



Comments on Separability of Data

We can see that the data is not linearly separable as there is a lot of overlap between the classes of the beans and the classes are not separable by a straight line or a hyperplane

d.

Explanation of Code:

- I use the Gaussian Naïve Bayes and the Multinomial Naive Bayes
- First, we import the libraries, we use train test split, accuracy score, precision score, recall score, GaussianNB, MultinomialNB
- First, we use the Gausian Naïve Bayes. We use train test split to split the data into training and testing sets. After that, we use StandardScaler to standardize the data. Then we declare the GaussianNB model, and fit it to our data. Then we use predict function to predict the output of the data. Then we print out the accuracy score, precision score, and recall score.
- Then, we use the Multinomial Naïve Bayes. We use train test split to split the data into training and testing sets. After that we declare the MultinomialNB model, and fit it to our data. Then we use predict function to predict the output of the data. Then we print out the accuracy score, precision score, and recall score.

Results:

Gaussian Naïve Bayes:

```
Accuracy: 0.8968049944913699
Precision: 0.8974148189793245
Recall: 0.8968049944913699
```

Multinomial Naïve Bayes:

```
Accuracy: 0.786265148733015
Precision: 0.7876641107304538
Recall: 0.786265148733015
```

Comments on the results:

```
Comments on results of gaussian naive bayes implementation
1. The accuracy of the gaussian naive bayes implementation 0.8968049944913699
2. The recall of the gaussian naive bayes implementation 0.8968049944913699
3. The precision of the gaussian naive bayes implementation 0.8974148189793245
Comments on results of multinomial naive bayes implementation
1. The accuracy of the multinomial naive bayes implementation
0.786265148733015
2. The recall of the multinomial naive bayes implementation 0.786265148733015
3. The precision of the multinomial naive bayes implementation
0.7876641107304538
Comments on the differences between the two implementations of Naive Bayes
1. The accuracy of the Multinomial naive bayes implementation is lower than
the accuracy of the gaussian naive bayes implementation
2. The recall of the Multinomial naive bayes implementation is lower than the
recall of the gaussian naive bayes implementation
3. The precision of the Multinomial naive bayes implementation is lower than
the precision of the gaussian naive bayes implementation
4. Overall, Gaussian Naive Bayes implementation is better than the Multinomial
Naive Bayes implementation
```

e.

Explanation of the Code:

- We import the libraries of PCA, f1 score, and logistic regression
- First, we use train test split to split the data into training and testing sets. After that, we declare a logistic regression model with max iterations as 10000. This number is chosen to ensure that the model can converge.
- Then we declare the StandardScaler() and standardise the data.
- Then we declare PCA model that will reduce the number of features to 4, and use it on the input data. Then we fit the logistic regression model with the input data, and make the predictions. Finally, we print various features like variance, accuracy score, precision score, recall score, f1 score.
- We repeat the same steps with different PCA models with the number of features being 6, 8, 10, and 12.

Results:

```
Number of components = 4 Variance: 0.9501988041227656 Accuracy: 0.8894601542416453 Precision: 0.8888329430312921 Recall: 0.8894601542416453 F1:
0.8889938558614282
Number of components = 6 Variance: 0.9891986430964116 Accuracy: 0.9243481454278369 Precision: 0.9248812148305942 Recall: 0.9243481454278369 F1:
0.9243511803525162
Number of components = 8 Variance: 0.9993093340631204 Accuracy: 0.9280205655526992 Precision: 0.9286837346038485 Recall: 0.9280205655526992 F1:
0.9280765809256262
Number of components = 10 Variance: 0.999908342576265 Accuracy: 0.9280205655526992 Precision: 0.9286837346038485 Recall: 0.9280205655526992 F1:
0.9280765809256262
Number of components = 12 Variance: 0.9999897278778122 Accuracy: 0.9280205655526992 Precision: 0.9286837346038485 Recall: 0.9280205655526992 F1:
0.9280765809256262
```

Results:

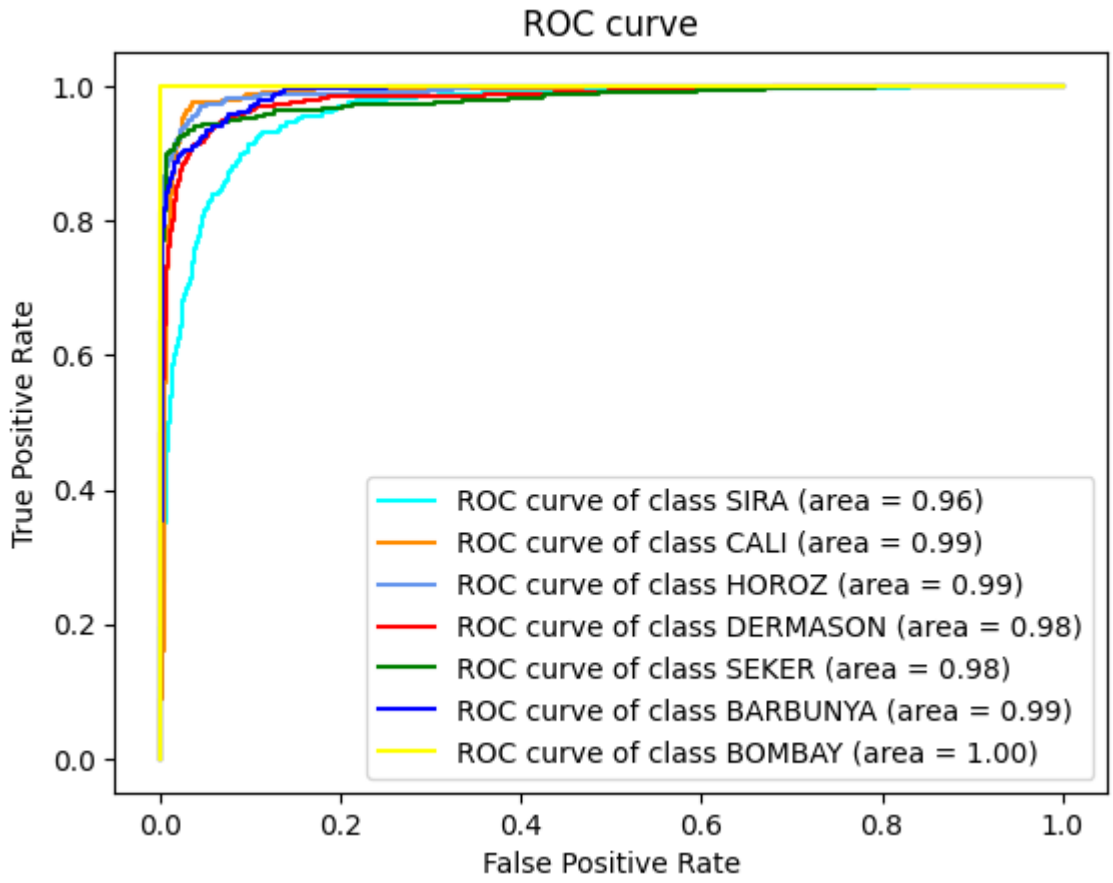
1. We have run the PCA algorithm for different number of components and we have calculated the accuracy, precision, recall and f1 score for each of the components.
2. We can see that the variance increases as the number of components increases.
3. Accuracy, precision, recall, and f1 scores first increase but for the number of components greater than 6, they stay at the same value.

f.

Explanation of the Code:

- First, we import the libraries for roc_auc_score, and for roc_curve and for auc
- In the code, first we create a list of the colours for each of the class we will plot. We create a list of the classes so that we can iterate through it. Then we declare a Gaussian Naïve Bayes model.
- Then we iterate through all the classes, and create a copy of the dataframe. We alter this copy so that the particular class we are on during the iteration is labelled as 1, and the rest of the classes are labelled as 0. After that we use train test split to split the data into training and testing sets. Then we run PCA on the data in order to reduce the number of features to 6. Then we fit the GaussianNB model to our data and then use predict_proba to predict the probability estimates for the output. Then we use roc_curve function to get all the coordinates for False Positive Rate, True positive rate. We use roc_auc_score, to get the area under the curve, which is the auc scoure. Then we plot this data.
- We repeat this process for all the classes

Result:



Comments on the output:

1. As we can see, the roc score for all the models is greater than 0.5. This means that the models are better than random guessing.
2. All the models have roc score greater than 0.9. This means that the models are excellent.
3. The roc score is highest for BOMBAY beans and lowest for SIRA beans.

g.

Explanation of the code:

- We import the library for Logistic Regression. All other required libraries have already been imported in previous cells of notebook.
- We create a logistic regression model, with C =1000, penalty=l2,max_iter=10000, and random state=0
- We split the data into training and testing sets using train test split.
- Then we run PCA to reduce the number of features to 10, and fit it to our data.
- Then we create a StandardScaler and use it to standardise our data.

- Then we fit the data to our logistic regression model and predict the output of the testing set using predict() function.
- Then we output the accuracy score, precision score, recall score.

Results:

```
Accuracy: 0.9313257436650753
Precision: 0.9320292062423539
Recall: 0.9313257436650753
```

Comments on the results:

1. We have used a logistic regression model with a C value of 1000, max_iter of 1000, penalty of none and random state of 0
2. The accuracy of the logistic regression model is 0.9313257436650753
3. The recall of the logistic regression model is 0.9313257436650753
4. The precision of the logistic regression model is 0.9320292062423539

Comparison of results with Naive Bayes models:

1. The accuracy of the logistic regression model is higher than the accuracy of the multinomial naive bayes implementation
2. The recall of the logistic regression model is higher than the recall of the multinomial naive bayes implementation
3. The precision of the logistic regression model is higher than the precision of the multinomial naive bayes implementation
4. Overall, the logistic regression model is better than the multinomial naive bayes implementation
5. The accuracy of the logistic regression model is higher than the accuracy of the gaussian naive bayes implementation
6. The recall of the logistic regression model is higher than the recall of the gaussian naive bayes implementation
7. The precision of the logistic regression model is higher than the precision of the gaussian naive bayes implementation
8. Overall, the logistic regression model is better than the gaussian naive bayes implementation
9. The logistic regression model has given us better performance than both the Naive Bayes models.