Machine Learning (CSE343/ECE343)

Assignment - 1

Report

Name: Khushdev Pandit
Roll no.: 2020211

Section - A

(Theoretical)

Q1

ML - Assignment - 1

Date
DELTA Pg No.

Section - A

Answer 1 (b) → If two Random Variables are highly correlated with third Random Variable, this doesn't convey that those two Random variable will also be high correlated.

→ If A and B are highly correlated with RV C.

→ Then, A and B need not be highly correlated, thus the above statement doesn't convey anything about high correlation between A and B.

Example 1 : Let, H and W be the random variable to denote height and weight of the person. Also, F denote the Body fat percentage of the person.

∴ Clearly, H and W are highly correlated as taller people tend to have higher weight.

Also, F and W are highly correlated as people with high fat percentage tend to have more weight.

∴ Both, H and F are highly correlated to W.

But, H and F needn't be highly correlated, because people with high body fat percentage may or may not have taller height. and vice-versa. Clearly, there is no dependency and no high correlation between F and H.

Answer 1(d) MAP gives us the solution that maximizes the posterior probability.

∴ We need to find model parameters, w that is most likely the given data.

∴ We use Baye's theorem to derive MAP solution for Linear regression:

$$P(w|D) = P(D|w) \times P(w) / P(D)$$

[D → data distribution with labels]

→ We assume gaussian distribution of weights, w:

$$\therefore w \sim N\left(0, \frac{I}{\alpha}\right) = N(\mu_x, C_x)$$

∴ Since w is a vector and has a Gaussian Distribution, so, w is a Gaussian Random vector.

→ So, parameter has a mean of zero vector and its covariance matrix is $\frac{I}{\alpha}$, a diagonal matrix with scalar values.

∴ Mean of w = zero vector = $\bar{0} = \mu$

∴ Covariance matrix of w = $C_x = \frac{I}{\alpha}$

where α is a scalar.

I is identity matrix.

→ Finding the Prior Probability of Gaussian vector, w:

We know that PDF of a Gaussian vector, w is given by:

Date
DELTA Pg No.

$$\Rightarrow f_W(w) = \frac{1}{\sqrt{(2\pi)^N \det[C_x]}} \exp\left(-\frac{1}{2}(w-u)^T C_x^{-1}(w-u)\right)$$

(N → size of Gaussian vector)

∴ By substituting ~~data~~ various values:

$$\therefore f_W(w) = \frac{1}{\sqrt{(2\pi)^N \left(\frac{1}{\alpha}\right)^N}} \exp\left(-\frac{1}{2}(w-\bar{0})^T \left(\frac{I}{\alpha}\right)^{-1}(w-\bar{0})\right)$$

$$\therefore f_W(w) = \frac{(\alpha)^{N/2}}{(2\pi)^{N/2}} \exp\left(-\frac{1}{2} \cdot w^T(\alpha I) w\right)$$

$$\therefore f_W(w) = \frac{(\alpha)^{N/2}}{(2\pi)^{N/2}} \exp\left(-\frac{\alpha}{2} w^T w\right) \quad \text{---- (i)}$$

⟹ Finding the likelihood Probability: $P(D|w)$:

∴ Output of classifier, $y_i = w^T x_i + \epsilon$.

As, ϵ is a Noise $\sim N(0, \sigma^2)$.

$$\therefore y_i \sim N(w^T x, \sigma^2) \qquad [w^T x \in R]$$

$$\therefore f(y_i|x_i; w) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{(y_i - w^T x)^2}{2\sigma^2}\right).$$

∴ Because all the y_i's $(y_1, y_2, \dots y_n)$ are independent so, our likelihood probability, $P(D|w)$ is given by:

$$\therefore P(D|w) = f(y_1 \dots y_n | x_1 \dots x_n; w) = \prod_{i=1}^{N} f(y_i|x_i; w).$$

$$P(D|w) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - w^T x)^2}{2\sigma^2}\right) \quad \text{---- (ii)}$$

→ $P(D|w)$ is the probabity of likelihood of our given Data distribution with labels, D given a parameter w.

Date
DELTA Pg No.

⟶ In MAP, given Data Distribution with labels, D we want to such a parameter w, that maximizes our ~~likelihood probability~~ ~~posterior~~ posterior probability $P(w|D)$:

∴ Posterior probability $P(w|D)$, is related to likelihood probability, $P(D|w)$ by bayes theorem:

$$\therefore P(w|D) = \frac{P(D|w) \times P(w)}{P(D)}$$

where • $P(w)$ → Prior probability of ~~vecto~~ Gaussian vector w.
• $P(D|w)$ → likelihood probability of Data.
• $P(D)$ → Data distribution's probability which is a constant.

$$\therefore \log P(w|D) = \log P(D|w) + \log P(w) - \log P(D)$$

Since, we want to maximize our posterior probabity, $P(w|D)$ over w.

$$\therefore w^* = \underset{w}{\operatorname{argmax}} \log P(w|D).$$

$$\therefore w^* = \underset{w}{\operatorname{argmax}} \left[\log P(D|w) + \log P(w) - \log P(D)\right]$$

→ Since $P(D)$ is independent of w, so it won't affect maximizing RHS, so we drop it.

$$w^* = \underset{w}{\operatorname{argmax}} \left[\log P(D|w) + \log P(w)\right]$$

∴ Substituting equation (i) & (ii) into above equations:

$$\therefore w^* = \underset{w}{\operatorname{argmax}} \left[\log \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - w^T x)^2}{2\sigma^2}\right) + \log \frac{(\alpha)^{N/2}}{(2\pi)^{N/2}} \exp\left(-\frac{\alpha w^T w}{2}\right) \right]$$

$$\therefore w^* = \underset{w}{\operatorname{argmax}} \left[\sum_{i=1}^{N} \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - w^T x)^2}{2\sigma^2}\right) + \log(\alpha)^{N/2} - \log(2\pi)^{N/2} - \frac{\alpha}{2} w^T w \right]$$

$$\therefore w^* = \underset{w}{\operatorname{argmax}} \left[\sum_{i=1}^{N} \left[\log \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{(y_i - w^T x)^2}{2\sigma^2} \right] + \log(\alpha)^{N/2} - \log(2\pi)^{N/2} - \frac{\alpha}{2} w^T w \right]$$

∴ Now, we remove terms that doesn't depend on w, as they won't affect the maximizing of posterior probability.

$$w^* = \underset{w}{\operatorname{argmax}} \left[\frac{-1}{\sigma^2} \sum_{i=1}^{N} \frac{(y_i - w^T x)^2}{2} - \frac{\alpha w^T w}{2} \right]$$

∴ As, negative of maximization is same as minimization, so we replace argmax by argmin, Also we remove σ^2 in denominator as it is constant.

$$\therefore w^* = \underset{w}{\operatorname{argmin}} \left[\sum_{i=1}^{N} \frac{(y_i - w^T x)^2}{2} - \frac{\alpha w^T w}{2} \right]$$

→ This is the required MAP solution for linear regression.

→ As one more step, since the summation is similar to L_2 loss (Squared error cost function), we take its average over N samples, again this won't affect minimization part.

$$\therefore W^* = \underset{W}{\operatorname{argmin}} \left[\frac{1}{N} \sum_{i=1}^{N} \frac{(y_i - W^T x)^2}{2} - \alpha \frac{W^T W}{2} \right]$$

→ This is the MAP solution for linear regression with prior Gaussian distribution of weights, w.

Answer 1(a) → In simple linear regression, there is only one input, $x \in R$. Let the training Data sample be $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots\ldots, (x_n, y_n)\}$.

$$\therefore \bar{x} = \text{mean of independent variable} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Also, $y_i = \theta_0 + \theta_1 x_i$ (in Linear Regression).

$$\therefore \bar{y} = \text{Mean of dependent variable} = \frac{1}{n} \sum_{i=1}^{n} y_i$$

Now, we simplify $\bar{y}$:

$$\bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i = \frac{1}{n} \sum_{i=1}^{n} (\theta_0 + \theta_1 x_i)$$

$$\bar{y} = \frac{1}{n} \sum_{i=1}^{n} \theta_0 + \frac{1}{n} \sum_{i=1}^{n} \theta_1 x_i$$

$$\bar{y} = \frac{n\theta_0}{n} + \frac{\theta_1}{n} \sum_{i=1}^{n} x_i$$

$$\bar{y} = \theta_0 + \frac{\theta_1}{n} \sum_{i=1}^{n} x_i$$

But $\frac{1}{n}\sum_{i=1}^{n} x_i = \bar{x}$

$$\therefore \quad \bar{y} = \theta_0 + \theta_1 \bar{x}$$

This equation is analogous to '$y_i = \theta_0 + \theta_1 x_i$' Linear Regression equation.

→ This signifies that, when input data is $\bar{x}$, predicted output ~~linear~~ by Linear Regression model is $\bar{y}$.

→ Furthermore, '$y_i = \theta_0 + \theta_1 x_i$' is equation of straight line for Linear Regression.

So, '$\bar{y} = \theta_0 + \theta_1 \bar{x}$' satisfies the least square Linear Regression line.

→ So, $(\bar{x}, \bar{y})$ passes through the least square fit line of Linear Regression.

Answer 1 (c) → Weak Law of large numbers states, "the Mean of a ~~seeq~~ Sequence of i.i.d. Random Variables converges in probability to the Expected value of Random Variable, as the no. of sequence goes to infinity".

Proof:

⇒ Let, $X_1, X_2, X_3, \ldots, X_n$ be independent and identically distributed Random variables. (i.i.d.), and each X_i has expected value, $E[X_i] = \mu$ and variance $Var[X_i] = \sigma^2$.

⇒ ∵ The ~~sum~~ mean of ~~seeq~~ sequence of iid RVs, X_i's is given by Sample mean, $M_n(X)$.

$$\therefore M_n(X) = \frac{1}{n}\sum_{i=1}^{n} X_i$$

⇒ Expected value of Sample Mean = $E[M_n(X)]$.

$$\therefore E[M_n(X)] = E\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right]$$

$$\therefore E[M_n(X)] = \frac{1}{n}\sum_{i=1}^{n} E[X_i]$$ (because ~~Expectation~~ Expectation is a linear operator)

$$\therefore E[M_n(X)] = \frac{1}{n}\sum_{i=1}^{n} \mu = \frac{n\mu}{n} = \mu$$

⇒ Variance of Sample Mean = $Var[M_n(X)]$

$$\therefore Var[M_n(X)] = Var\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right]$$

Since all X_i's are independent of Random Variable.

$$\therefore Var[M_n(X)] = \sum_{i=1}^{n} Var\left[\frac{X_i}{n}\right]$$

$\therefore$ We know $Var[aY] = a^2 Var[Y]$

$\therefore Var[M_n(X)] = \sum_{i=1}^{n} \frac{1}{n^2} Var[X_i] = \frac{1}{n^2} \sum_{i=1}^{n} Var[X_i]$

$\therefore Var[M_n(X)] = \frac{1}{n^2} \sum_{i=1}^{n} \sigma^2$

$\therefore Var[M_n(X)] = \frac{\sigma^2}{n}$

$\Rightarrow$ Now, we will use Chebyshev's Inequality to prove LLN :

Chebyshev Inequality says :

$$P[|Y - E[Y]| \geq k] \leq \frac{Var[Y]}{k^2} \quad \forall\, k>0$$

$\therefore$ With sample mean :

$$\Rightarrow P[|M_n(X) - E[M_n(X)]| \geq k] \leq \frac{Var[M_n(X)]}{k} \quad ; \forall\, k>0$$

$$\therefore P[|M_n(X) - \mu| \geq k] \leq \frac{\sigma^2}{nk} \quad ; \forall\, k>0$$

as the no. of sequence tends to infinity ($n \to \infty$)

$$\therefore \lim_{n\to\infty} P[|M_n(X) - \mu| \geq k] \leq \frac{\sigma^2}{nk} \quad ; \forall\, k>0$$

$$\Rightarrow \lim_{n\to\infty} P[|M_n(X) - \mu| \geq k] = 0 \quad ; \forall\, k>0$$

$\therefore$ Since, μ = Expected value of $X_i = E[X]$.

$\therefore$ The above equation tells that as the no. of samples tends to infinity, the Probability with which Sample mean, $M_n(X)$ is outside the range $[\mu - k, \mu + k]$

is 0 (even for very small k, $k \approx 0$, but $k > 0$).

Hence, Sample mean, $M_n(X)$ converges to Expected value, μ, in probability.

$$\therefore \quad M_n(X) \xrightarrow[n \to \infty]{P()} \mu$$

Assignment-1

Section-A, Q1 (c)

Program to illustrate Weak law of Large numbers for Gaussian Distribution

I have chosen a Gaussian Random variable with mean = 10 and std. deviation = 5.

Therefore, Random Variable ~ N(mean=10, std_deviation=5)

Since, we used a Gaussian Distribution with mean = 10 and std. deviation = 5. So, the Expected value of Gaussian Distribution is 10.

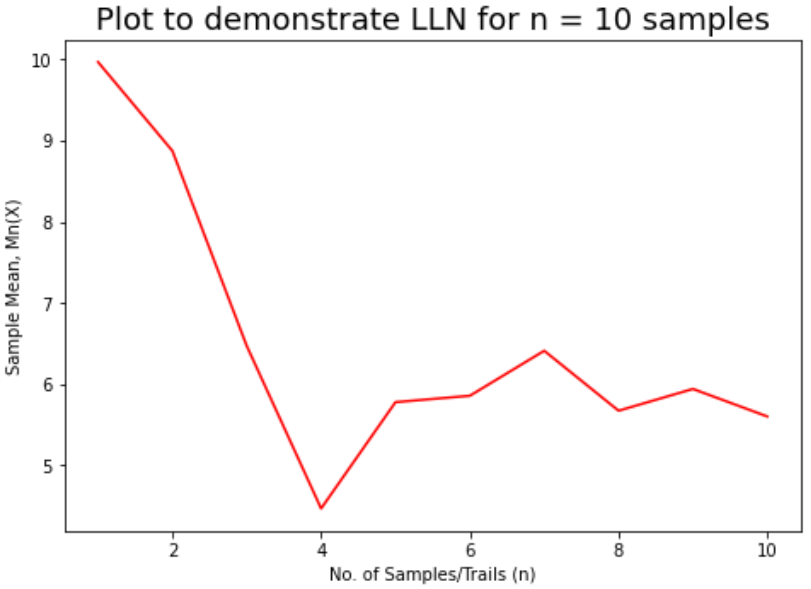
In [ ]:
```
#importing necessary libraries
import numpy as np
import matplotlib.pyplot as plt
```

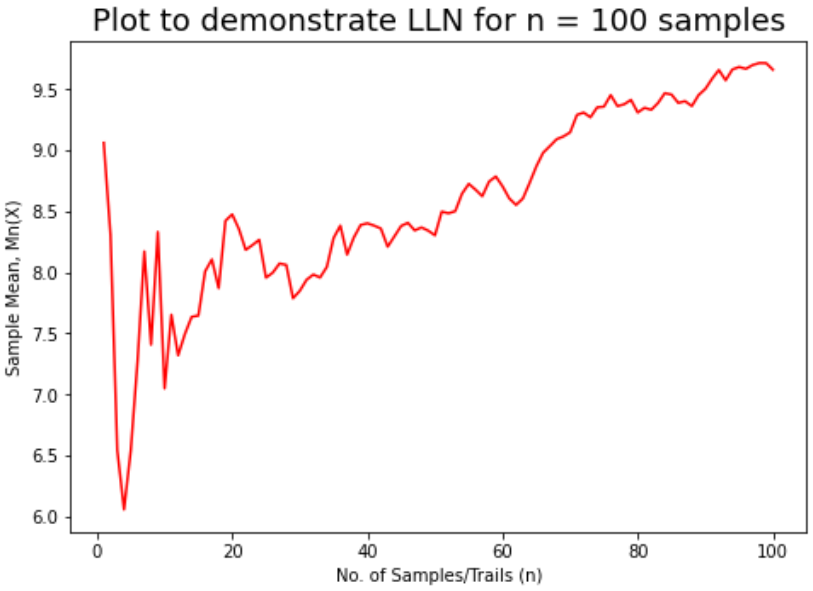
In [ ]:
```
# Program to demonstrate Weak law of Large numbers (LLN)

class GaussianLLN:
    def __init__(self, n) -> None:
        self.n = n

    # I have verified LLN for a Gaussian Distribution ~ N(mean=10, std_deviation=5)
    # Create a Gaussian RV Data with 'n' samples
    def createGaussianData(self):
        self.gaussianData = 5 * np.random.randn(self.n) + 10


    # Calculating the Sample Mean, Mn(X) for the Gaussian Distributions with n samples
    def calculateSampleMean(self):
        self.sampleMean = np.zeros(self.n)

        prefixSum = 0
        for i in range(self.n):
            prefixSum += self.gaussianData[i]
            self.sampleMean[i] = prefixSum / (i + 1)


    # Plotting the Sample Mean, Mn(X) for the Gaussian Distributions with n samples/trails
    def plotData(self):
        fig = plt.figure()
        axes = fig.add_axes([0, 0, 1, 1])

        # x-axis (no. of trails, n)
        x = np.arange(1, self.n + 1)

        # Adding information to the graph
        axes.plot(x, self.sampleMean, 'r-')
        axes.set_xlabel('No. of Samples/Trails (n)')
        axes.set_ylabel('Sample Mean, Mn(X)')
        axes.set_title(f'Plot to demonstrate LLN for n = {self.n} samples', fontsize=18);


    # Function to illustrate the Weak Law of Large Numbers
    def illustrate(self):
        self.createGaussianData()          # Create our Gaussian Data
        self.calculateSampleMean()         # Compute the Sample Mean of our Gaussian Data
        self.plotData()                    # Plot the Sample Mean of our Gaussian Data
```

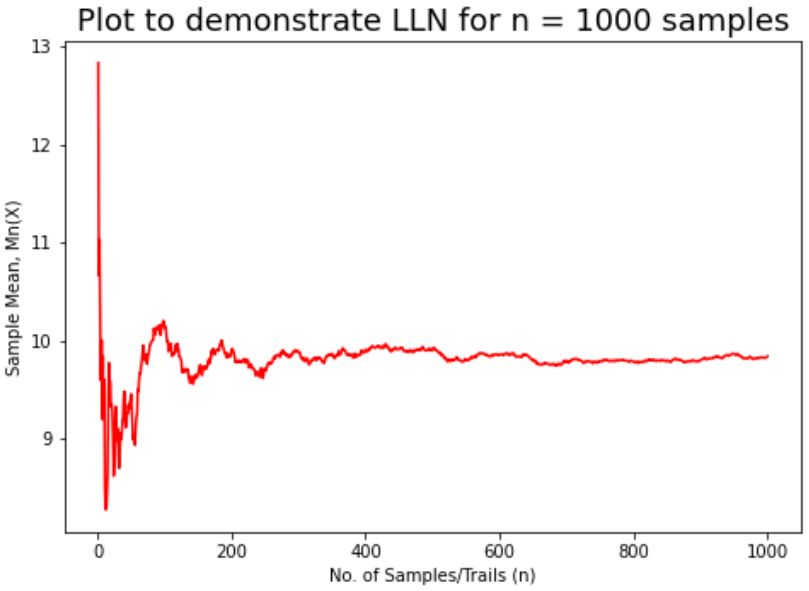
In [ ]:
```
# "No. of trails (n)" for which we will illustrate Law of Large Numbers
no_of_sample = [10, 100, 1000, 10000, 100000, 1000000, 10000000]
```

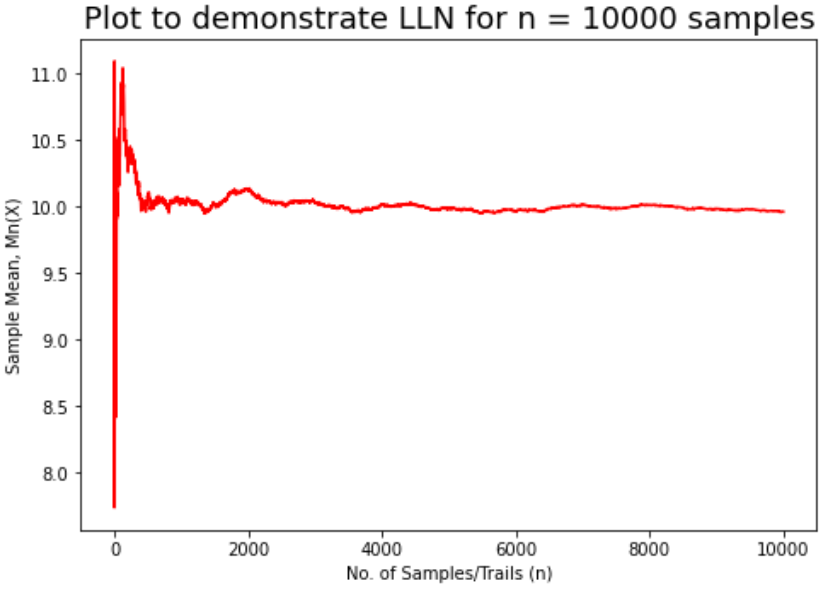
```
print('''Plots to verify the Law of Large numbers for Gaussian Distribution ~ N(mean=10, std_deviation=5)
for various number of samples.''')

# Looping through all the values of no. of trails (n), we basically increase the no. of trails (n) by 10 everytime
for n in no_of_sample:
    lln = GaussianLLN(n)
    lln.illustrate()
```

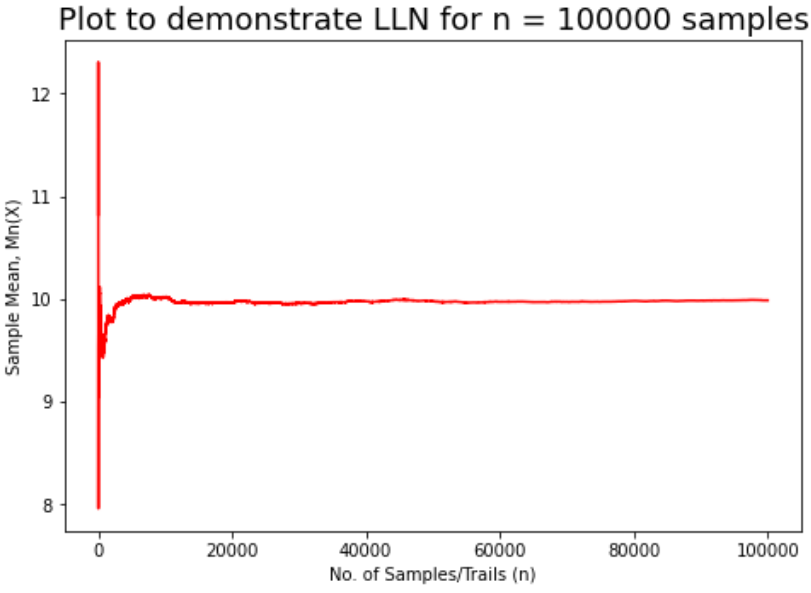
Plots to verify the Law of Large numbers for Gaussian Distribution ~ N(mean=10, std_deviation=5)
for various number of samples.

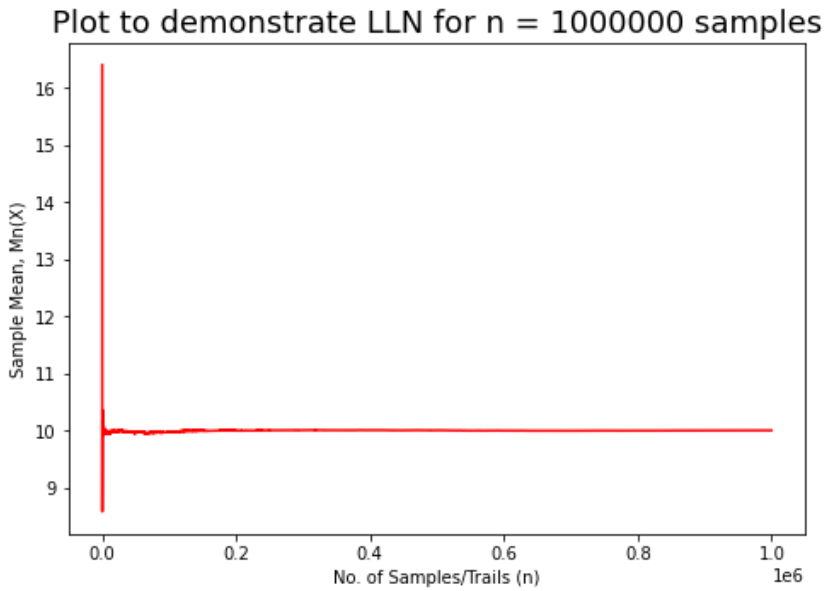


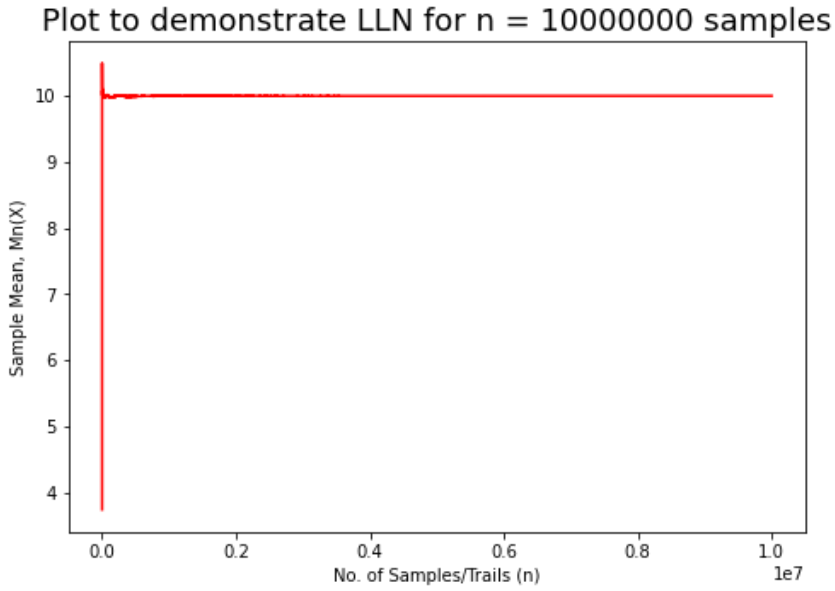



Plot to demonstrate LLN for n = 1000 samples
Sample Mean, Mn(X)
No. of Samples/Trails (n)
13
12
11
10
9
0
200
400
600
800
1000


Plot to demonstrate LLN for n = 10000 samples
Sample Mean, Mn(X)
No. of Samples/Trails (n)
11.0
10.5
10.0
9.5
9.0
8.5
8.0
0
2000
4000
6000
8000
10000


Plot to demonstrate LLN for n = 100000 samples
Sample Mean, Mn(X)
No. of Samples/Trails (n)
12
11
10
9
8
0
20000
40000
60000
80000
100000





From these plots we can see that, as the no. of samples/trails becomes large and large, the Sample mean, Mn(X) becomes equal to the Expected mean, which is 10.

Thus, all these plots demonstrate the Weak Law of Large numbers for a Gaussian Random Variable.

Section - B

(Scratch Implementation)

Q2

Section-B

Importing Required libraries

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Reading Data

In [ ]:
```
df = pd.read_csv('Real Estate.csv')
```

Adding a columns of 1's into the Housing DataFrame

In [ ]:
```
df['X7 Constant'] = np.ones(len(df))
```

Information about Housing DataFrame

In [ ]:
```
df.info()
```

Out[ ]:
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 414 entries, 0 to 413
Data columns (total 9 columns):
 #   Column                                  Non-Null Count  Dtype  
---  ------                                  --------------  -----  
 0   No                                      414 non-null    int64  
 1   X1 transaction date                     414 non-null    float64
 2   X2 house age                            414 non-null    float64
 3   X3 distance to the nearest MRT station  414 non-null    float64
 4   X4 number of convenience stores         414 non-null    int64  
 5   X5 latitude                             414 non-null    float64
 6   X6 longitude                            414 non-null    float64
 7   Y house price of unit area              414 non-null    float64
 8   X7 Constant                             414 non-null    float64
dtypes: float64(7), int64(2)
memory usage: 29.2 KB
```

In [ ]:
```
df.head()
```

Out[ ]:

	No	X1 transaction date	X2 house age	X3 distance to the nearest MRT station	X4 number of convenience stores	X5 latitude	X6 longitude	Y house price of unit area	X7 Constant
0	1	2012.917	32.0	84.87882	10	24.98298	121.54024	37.9	1.0
1	2	2012.917	19.5	306.59470	9	24.98034	121.53951	42.2	1.0
2	3	2013.583	13.3	561.98450	5	24.98746	121.54391	47.3	1.0
3	4	2013.500	13.3	561.98450	5	24.98746	121.54391	54.8	1.0
4	5	2012.833	5.0	390.56840	5	24.97937	121.54245	43.1	1.0

Features/columns present in Housing DataFrame

In [ ]:
```
df.columns
```

Out[ ]:
```
Index(['No', 'X1 transaction date', 'X2 house age',
       'X3 distance to the nearest MRT station',
       'X4 number of convenience stores', 'X5 latitude', 'X6 longitude',
       'Y house price of unit area', 'X7 Constant'],
      dtype='object')
```

Solution for Q2 (a)

Created a Custom Class for "Linear Regression Model"

The Linear Regression model has three functions:

1) predict(): Given a Dataset x (it can be Training or Testing Data), it will predict the House price, h(x) by using its Model Parameters, w

2) costFunction(): Given the Dataset (x) and correct output label of House price (y), it will determine the Cost function of the model in the prediction of House price, h(x) and correct output price (y)

3) calculateRMSE(): Given the Testing/Training/Validation Dataset (x), we calculate the Root Mean Square Error (RMSE) in the predicted House price output, h(x) and correct house price, y

4) fit(): Train the model on the Training Dataset, using Gradient Descent Algorithm

Created a custom class for "Gradient Descent Algorithm"

Gradient Descent has two functions:

1) runGradientDesecent(): To run Gradient Desecent Algorithm on its Linear Regression Model

2) getDerivativeOfCostFunction(): Compute the Derivative of the Cost function of Linear Regression Model, which is used in Gradient Descent Algorithm

In [ ]:

```
# Linear Regression Model
class LinearRegressionModel:
    def __init__(self, n) -> None:
        # Intitializing the Linear Regression Model with model parameters (or weights) w
        self.parameters = np.zeros(n)

    # Test the model over the Housing Dataset, x. It basically computes House price
    # prediction, h(x) using its model parameters, w by matrix multiplication
    def predict(self, x):
        y_test = np.dot(x, self.parameters)
        return y_test

    # Calculate the Cost function, J(h) of the model in the prediction of House price, h(x)
    def costFunction(self, x, y):
        m = len(x)
        prediction_error = self.predict(x) - y
        cost_function_value = np.dot(prediction_error.T, prediction_error) / (2*m)
        return cost_function_value

    # Root Mean Square Error is Square root of twice of Cost function
    def calculateRMSE(self, x, y):
        return np.sqrt(2 * self.costFunction(x, y))

    # Train the Model on the Testing Dataset (x_train,y_train)
    def fit(self, x_train, y_train, x_test, y_test):
        algorithm = GradientDescent(0.03, 1000)
        return algorithm.runGradientDesecent(self, x_train, y_train, x_test, y_test)


# Gradient Descent
class GradientDescent:
    def __init__(self, learningRate, epochs) -> None:
        self.alpha = learningRate      # Learning parameter of the the Gradient Descent Algorithm
        self.epochs = epochs

    # Run Gradient Desecent Algorithm on given model
    def runGradientDesecent(self, model: LinearRegressionModel, x_train, y_train, x_test, y_test):
        # Store all the values of RMSE, to be used for plotting
```

```
        training_rmse = []
        validation_rmse = []

        # Run gradient descent algorithm, over multiple itertaions/epochs
        for epoch in range(self.epochs):
            # θj  =  θj  -  α * dJ(θ)/dθj
            model.parameters = model.parameters - self.alpha * self.getDerivativeOfCostFunction(
                model, x_train, y_train)

            # Add the Training RMSE & Validation RMSE
            training_rmse.append(model.calculateRMSE(x_train, y_train))
            validation_rmse.append(model.calculateRMSE(x_test, y_test))

        return training_rmse,validation_rmse

    # Compute the Derivative of the Cost function of Linear Regression Model, which is used in
    # Gradient Descent Algorithm
    def getDerivativeOfCostFunction(self, model: LinearRegressionModel, x_train, y_train):
        prediction_error = model.predict(x_train) - y_train
        derivative_of_cost_function = np.dot(x_train.T, prediction_error) / len(x_train)
        return derivative_of_cost_function
```

Creates the Dataset of Training and Testing set for each value of 'k' in "K-Fold cross-validation"

In [ ]:

```
# Function to get the Training and Testing Data for each iteration of 'k' in K-Fold cross-validation

def getTrainAndTestData(fold, fold_size):
    # Calculate the Training and Testing Data inputs and outputs
    x_train = df.drop(np.arange((fold - 1) * fold_size, fold * fold_size), axis=0)
    x_test = df.iloc[(fold - 1) * fold_size : fold * fold_size]
    y_train = x_train['Y house price of unit area']
    y_test = x_test['Y house price of unit area']

    # Drroping Output labels from Training & Testing Dataset
    x_train = x_train.drop(['Y house price of unit area', 'No'], axis=1)
    x_test = x_test.drop(['Y house price of unit area', 'No'], axis=1)

    # Renaming Indices (starting from 0)
    x_train.index = np.arange(len(x_train))
    y_train.index = np.arange(len(y_train))
    x_test.index = np.arange(len(x_test))
    y_test.index = np.arange(len(y_test))

    # Normalizing Training Data: For each feature/columns in Dataset,
    # we normalize their values to make their ranges in similar scale.
    # Operation performed:
    # feature = (feature - feature.mean()) / feature.std()
    for i in range(len(x_train.columns)):
        feature = x_train.columns[i]
        x_train[feature] = (x_train[feature] - x_train[feature].mean()) / (x_train[feature].std())
    x_train['X7 Constant'] = np.ones(len(x_train))

    # Normalizing Test Data similar to I did in Training Data
    for i in range(len(x_test.columns)):
        feature = x_test.columns[i]
        x_test[feature] = (x_test[feature] - x_test[feature].mean()) / x_test[feature].std()
    x_test['X7 Constant'] = np.ones(len(x_test))

    return x_train, y_train, x_test, y_test
```

Conclusion after performing K-Fold cross-validation on the Given Dataset

In [ ]:

```
def performKFoldCrossValidation(k):
    fold_size = len(df) // k        # size of Dataset in each fold of size 'k'
    train_error = 0                 # Training error
    test_error = 0                  # Testing error
```

```
    # For each value of 'k', creates 'k' different Training Dataset
    for fold in range(1,k+1):
        # Training and Testing set for the given value of 'k-fold'
        x_train, y_train, x_test, y_test = getTrainAndTestData(fold, fold_size)

        # initialize a Linear regression model
        linear_reg_model = LinearRegressionModel(len(x_train.columns))

        # Train the Model
        linear_reg_model.fit(x_train, y_train, x_test, y_test)

        # Add the Training and Testing error, and take average at the end
        train_error += linear_reg_model.calculateRMSE(x_train, y_train)
        test_error += linear_reg_model.calculateRMSE(x_test, y_test)

    print(f"Mean Training RMSE with k = {k} is:  {(train_error/k):.8f}")
    print(f"Mean Testing RMSE with k = {k} is:  {(test_error/k):.8f}")
    print()


# Perform K-Fold cross-validation for the follwing values of 'k'
for k in [2,3,4,5]:
    performKFoldCrossValidation(k)

print("Since the Root Mean Square Error for all values of 'k' in K-Fold cross-validation is nearly same.")
print("Though, the Mean RMSE for Training Data is less for 'k' = 5")
print("Also the size of Dataset is medium size, so the value of 'k' chosen is : \"k = 5\"")
```

```
Mean Training RMSE with k = 2 is:  8.71070805
Mean Testing RMSE with k = 2 is:  8.81579086

Mean Training RMSE with k = 3 is:  8.74807329
Mean Testing RMSE with k = 3 is:  8.85197431

Mean Training RMSE with k = 4 is:  8.75051335
Mean Testing RMSE with k = 4 is:  8.78306862

Mean Training RMSE with k = 5 is:  8.76087827
Mean Testing RMSE with k = 5 is:  8.76428813

Since the Root Mean Square Error for all values of 'k' in K-Fold cross-validation is nearly same.
Though, the Mean RMSE for Training Data is less for 'k' = 5
Also the size of Dataset is medium size, so the value of 'k' chosen is : "k = 5"
```

Solution for Q2 (b)

Plotting the Training and Validation RMSE while Training the model, using Gradient Descent Algorithm

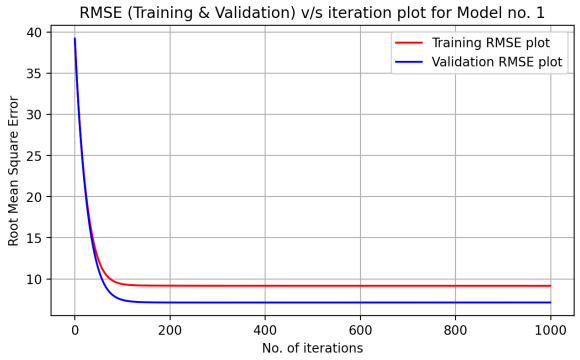
The optimal value of K chosen for 'K-Fold cross-validation' is : 5

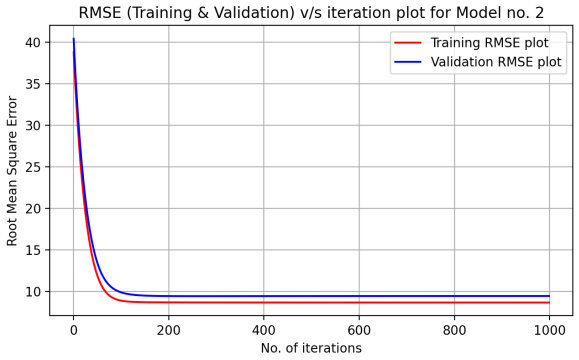
In [ ]:

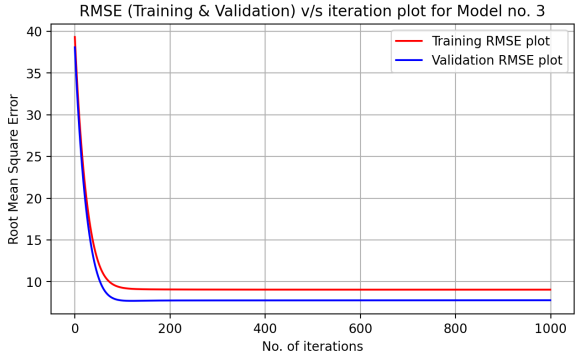
```
def plot_RMSE_for_Training_and_Validation(k):
    fold_size = len(df) // k        # size of Dataset in each fold of size 'k'

    fig, axes = plt.subplots(3, 2, dpi=200, figsize=(17,16))
    fig.subplots_adjust(wspace=0.3, hspace=0.4)
    i = 0

    # Creates 'k' different Training & Validation Dataset
    for fold in range(1, k+1):
        # Training and Validation set for the each fold
        x_train, y_train, x_valid, y_valid = getTrainAndTestData(fold, fold_size)

        # initialize a Linear regression model
        linear_reg_model = LinearRegressionModel(len(x_train.columns))

        # Train Model and run gradient descent algorithm it, to find its Model parameters, w
```

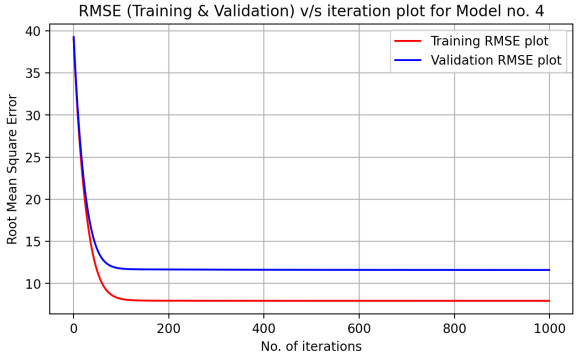
```
        train_rmse, validation_rmse = linear_reg_model.fit(x_train, y_train, x_valid, y_valid)

        # RMSE of Training & Validation set for each model trained
        print(f"After Training the Model number {fold}...")
        print(f"The RMSE Training Error : {linear_reg_model.calculateRMSE(x_train, y_train)}")
        print(f"The RMSE Validation Error : {linear_reg_model.calculateRMSE(x_valid, y_valid)}")
        print()

        # Plotting Training and Validation RMSE
        axes[i//2][i%2].plot(train_rmse, 'r-',label='Training RMSE plot')
        axes[i//2][i%2].plot(validation_rmse, 'b-',label='Validation RMSE plot')
        axes[i//2][i%2].set_title(f"RMSE (Training & Validation) v/s iteration plot for Model no. {fold}")
        axes[i//2][i%2].set_xlabel(f'No. of iterations', )
        axes[i//2][i%2].set_ylabel('Root Mean Square Error')
        axes[i//2][i%2].legend()
        axes[i//2][i%2].grid(True)
        i += 1

# optimal value of K for "K-Fold cross-validation" chosen is 5
k = 5
print(f"The optimal value of K chosen for 'K-Fold cross-validation' is : {k} \n")
plot_RMSE_for_Training_and_Validation(k=k)

print("The plots of the RMSE V/s iteration graph for Training & Testing Data (for optimal k) : ")
```

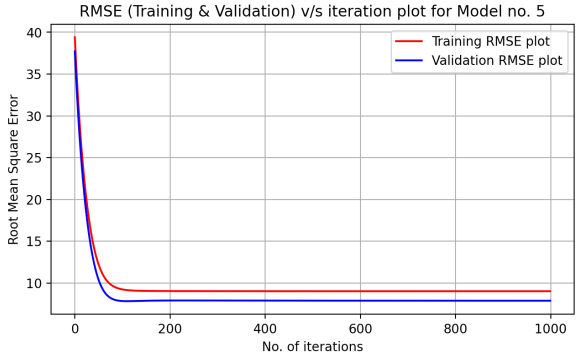
```
The optimal value of K chosen for 'K-Fold cross-validation' is : 5

After Training the Model number 1...
The RMSE Training Error : 9.158868786909448
The RMSE Validation Error : 7.130756380189358

After Training the Model number 2...
The RMSE Training Error : 8.649390753469541
The RMSE Validation Error : 9.436091713007146

After Training the Model number 3...
The RMSE Training Error : 9.044423511032385
The RMSE Validation Error : 7.775777429609902

After Training the Model number 4...
The RMSE Training Error : 7.937365188991557
The RMSE Validation Error : 11.602826730406399

After Training the Model number 5...
The RMSE Training Error : 9.014343087786454
The RMSE Validation Error : 7.8759883965731134

The plots of the RMSE V/s iteration graph for Training & Testing Data (for optimal k) :
```

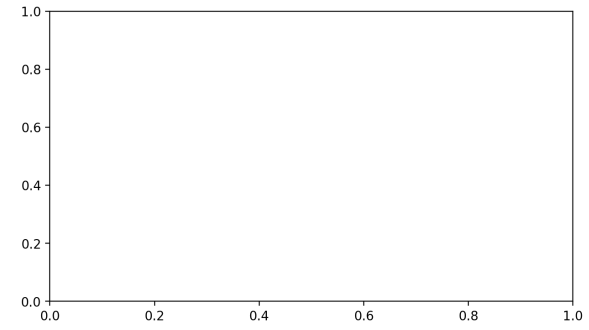












Observations in the plots of the RMSE V/s iteration graph for Training & Testing Data:

1) The value of RMSE for both Training & Testing Data decreases with the iterations. This indicates that RMSE is converging to global minimum, with increase in iterations.

2) The plot of the RMSE V/s iteration graph for Training & Testing Data shows the similar behaviour for all the 'k' models trained.

Verifying the Results using Skicit Learn's Linear Regression model

In [ ]:
```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
```

In [ ]:
```
print("Implementing Linear Regression using K-fold Cross Validation, with k=5.\n")

Y = df['Y house price of unit area']
X = df.drop(['No', 'X7 Constant', 'Y house price of unit area'], axis=1)
k = 5
mean_rmse = 0

for i in range(1,k+1):
    x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=1/k)

    model = LinearRegression()
```

```
    model.fit(x_train, y_train)

    y_pred = model.predict(x_test)
    error = y_pred - y_test

    rmse = np.sqrt(np.dot(error.T, error) / len(y_test))
    mean_rmse += rmse
    print(f"Model-{i} in {k}-fold regularization, RMSE: {rmse}")

print(f"\nK-fold regularization (k={k}), Mean RMSE: {mean_rmse/k}")
```

```
Implementing Linear Regression using K-fold Cross Validation, with k=5.

Model-1 in 5-fold regularization, RMSE: 11.648281377815865
Model-2 in 5-fold regularization, RMSE: 8.772565640249331
Model-3 in 5-fold regularization, RMSE: 9.958118147263686
Model-4 in 5-fold regularization, RMSE: 7.7704577793838805
Model-5 in 5-fold regularization, RMSE: 11.284082192928121

K-fold regularization (k=5), Mean RMSE: 9.886701027528176
```

As, the Mean RMSE is same nearly same for both "My implementation of Linear Regression" and "Scikit Learn implementation of Linear Regression", for optimal value of k = 5 in K-fold Cross validation.

I can say that my model is quite good in estimating House price.

Solution for Q2 (c)

Implementing Ridge Regression from scratch

In code, Ridge Regularization for Linear Regression Model is inherited from the earlier Linear Regression class. Only neccessary methods (for Cost function, RMSE calculation, Training model) have been overridden.

Similarly, Ridge Regularization for Gradient Descent is inherited from the earlier Gradient Descent class. Only one neccessary method (for calculating Derivative of Cost function) is overridden.

In [ ]:

```
# Linear Model for Ridge Regression
class LinearRegressionModel_Ridge(LinearRegressionModel):
    def __init__(self, n, λ) -> None:
        super().__init__(n)
        self.λ = λ          # Lambda parameter for Ridge Linear Regression

    # Only these methods will change in Ridge Regularization
    # Calculate the Cost function, J(h) of the model in the prediction of House price, h(x)
    def costFunction(self, x, y):
        # Penaly term in Ridge Regression
        penalty_cost_function = self.λ * np.dot(self.parameters.T, self.parameters)
        return super().costFunction(x, y) + penalty_cost_function

    # Root Mean Square Error same as the Square root of twice of Cost function
    def calculateRMSE(self, x, y):
        return np.sqrt(2 * super().costFunction(x, y))

    # Train the Model on the Testing Dataset (x_train,y_train)
    def fit(self, x_train, y_train, x_test, y_test):
        algorithm = GradientDescent_Ridge(0.03, 1000)
        return algorithm.runGradientDesecent(self, x_train, y_train, x_test, y_test)


# Gradient Descent for Ridge Linear Regression
class GradientDescent_Ridge(GradientDescent):
    def __init__(self, learningRate, epochs) -> None:
        super().__init__(learningRate, epochs)

    # Compute the Derivative of the Cost function of Linear Regression Model, which is used in
    # Gradient Descent Algorithm. Only this method will change in Ridge Regularization
```

```
    def getDerivativeOfCostFunction(self, model: LinearRegressionModel_Ridge, x_train, y_train):
        penalty_term_in_derivative = 2 * model.λ * model.parameters
        return super().getDerivativeOfCostFunction(model, x_train, y_train) + penalty_term_in_derivative
```

Conclusion after performing Ridge Regression for various values of Hyperparameter (λ).

Finally, the Best value of λ is reported as follows for optimal value of k in K-fold cross Validation

In [ ]:

```
def plot__Training_and_Validation_RMSE_for_Ridge_Regression(k, λ):
    print("********** Tuning Parameter, λ in Ridge Regression:", λ, "**********")
    fold_size = len(df) // k        # size of Dataset in each fold of size 'k'

    global train_rmse, validation_rmse
    train_rmse, validation_rmse = [], []

    # Creates 'k' different Training & Validation Dataset
    for fold in range(1, k+1):
        # Training and Validation set for the each fold
        x_train, y_train, x_valid, y_valid = getTrainAndTestData(fold, fold_size)

        # initialize a Linear regression model
        ridge_reg_model = LinearRegressionModel_Ridge(len(x_train.columns), λ)

        # Train Model and run gradient descent algorithm it, to find its Model parameters, w
        __train_rmse, __validation_rmse = ridge_reg_model.fit(x_train, y_train, x_valid, y_valid)
        train_rmse.append(__train_rmse)
        validation_rmse.append(__validation_rmse)

        # RMSE of Training & Validation set for each model trained
        print(f"After Training the Model number {fold}...")
        print(f"The RMSE Training Error : {ridge_reg_model.calculateRMSE(x_train, y_train)}")
        print(f"The RMSE Validation Error : {ridge_reg_model.calculateRMSE(x_valid, y_valid)}")
        print()


# RMSE for Training and Validation set
train_rmse, validation_rmse = [], []

# optimal value of K for "K-Fold cross-validation" chosen is 5
k = 5
λ = [1, 0.1, 0.01]
for lambda_ in λ:
    plot__Training_and_Validation_RMSE_for_Ridge_Regression(k=k, λ=lambda_)


print("\nConclusion:")
print(f"So, after the tuning various values of Regularization parameter, λ, the best chosen value is {λ[-1]}.")
print("Reasons:")
print("1) The value of λ which gives least RMSE and the least variance in all the models is: ", λ[-1])
print(f"'2) This value of λ={λ[-1]} is chosen beacuse, the model was already performing Best and
  wasn't overfitting the Data. Hence, no penalty should be added. So, the value of Tuning Parameter, λ
  must be low.'")

print("\nPlots for RMSE (Training + Validation) v/s iterations, for best value of Regularization parameter, λ")

# Subplot for RMSE for Best value of Hyperparamter, λ
fig, axes = plt.subplots(3, 2, dpi=200, figsize=(17,16))
fig.subplots_adjust(wspace=0.3, hspace=0.4)
for i in range(k):
    axes[i//2][i%2].plot(train_rmse[i], 'r-',label='Training RMSE plot')
    axes[i//2][i%2].plot(validation_rmse[i], 'b-',label='Validation RMSE plot')
    axes[i//2][i%2].set_title(f"'Ridge Regression (λ={λ[-1]}) plot for Model no. {i+1}
RMSE (Training & Validation) v/s iteration plot'")
    axes[i//2][i%2].set_xlabel(f'No. of iterations')
    axes[i//2][i%2].set_ylabel('Root Mean Square Error')
    axes[i//2][i%2].legend()
    axes[i//2][i%2].grid(True)
```

********** Tuning Parameter, λ in Ridge Regression: 1 **********
After Training the Model number 1...
The RMSE Training Error : 27.332823712660744
The RMSE Validation Error : 27.197269268239435

After Training the Model number 2...
The RMSE Training Error : 27.026315334698577
The RMSE Validation Error : 28.693096040762143

After Training the Model number 3...
The RMSE Training Error : 27.476841221392387
The RMSE Validation Error : 26.09130551822521

After Training the Model number 4...
The RMSE Training Error : 27.064903184351007
The RMSE Validation Error : 27.82298280882618

After Training the Model number 5...
The RMSE Training Error : 27.50303216949731
The RMSE Validation Error : 25.795163301190392

********** Tuning Parameter, λ in Ridge Regression: 0.1 **********
After Training the Model number 1...
The RMSE Training Error : 11.195828595755215
The RMSE Validation Error : 9.883825389593284

After Training the Model number 2...
The RMSE Training Error : 10.762004767095602
The RMSE Validation Error : 12.218027865752596

After Training the Model number 3...
The RMSE Training Error : 11.135887325937077
The RMSE Validation Error : 9.401385326185974

After Training the Model number 4...
The RMSE Training Error : 10.231342563181908
The RMSE Validation Error : 13.05090924526742

After Training the Model number 5...
The RMSE Training Error : 11.108255641855747
The RMSE Validation Error : 9.327289600662287

********** Tuning Parameter, λ in Ridge Regression: 0.01 **********
After Training the Model number 1...
The RMSE Training Error : 9.191071821724323
The RMSE Validation Error : 7.215242358705414

After Training the Model number 2...
The RMSE Training Error : 8.683236112354278
The RMSE Validation Error : 9.583863911388478

After Training the Model number 3...
The RMSE Training Error : 9.07816847209956
The RMSE Validation Error : 7.696637162094075

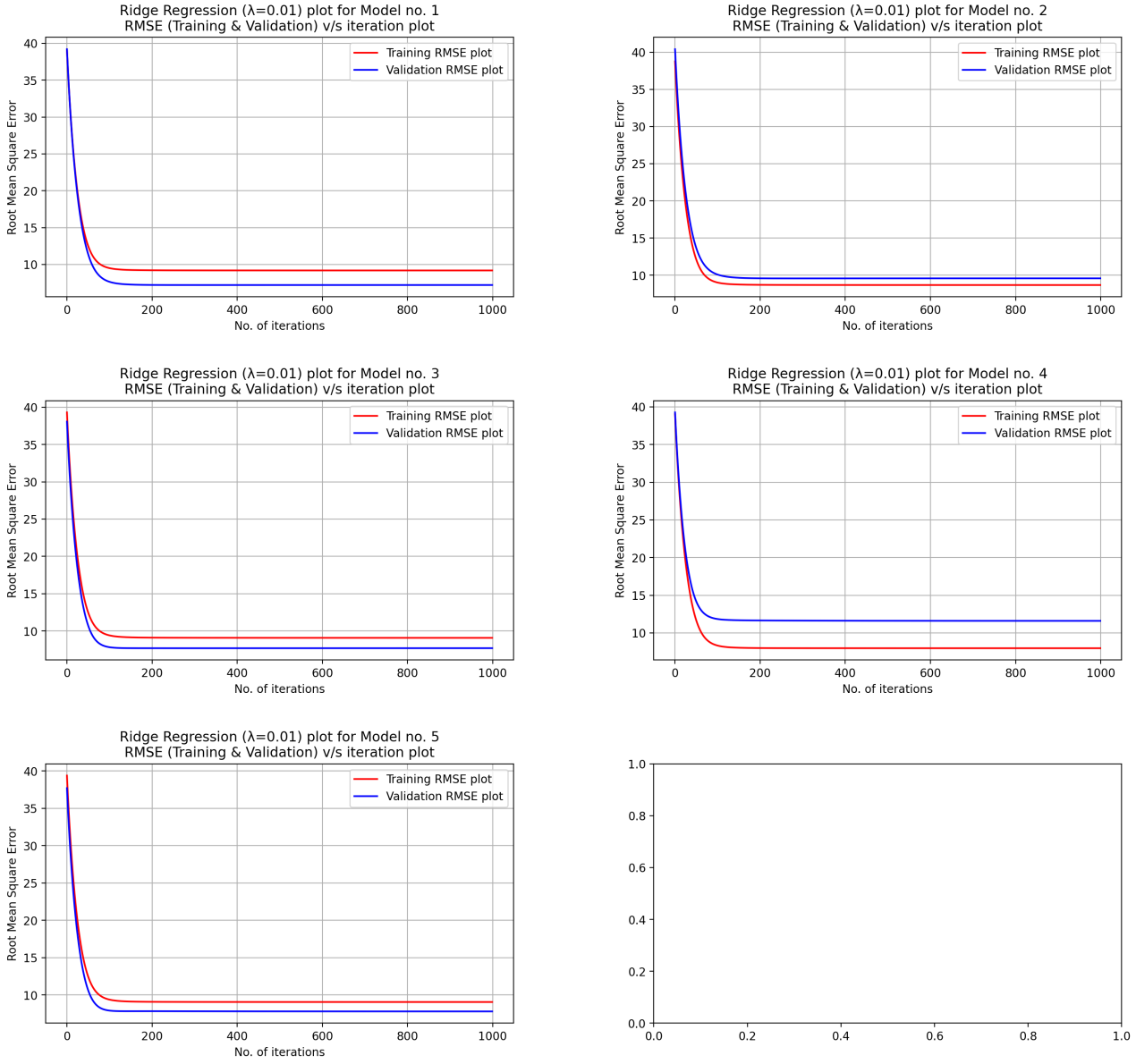
After Training the Model number 4...
The RMSE Training Error : 7.974578685817553
The RMSE Validation Error : 11.603500438151714

After Training the Model number 5...
The RMSE Training Error : 9.047654482861134
The RMSE Validation Error : 7.787475196306616

Conclusion:
So, after the tuning various values of Regularization parameter, λ, the best chosen value is 0.01.
Reasons:
1) The value of λ which gives least RMSE and the least variance in all the models is: 0.01
2) This value of λ=0.01 is chosen beacuse, the model was already performing Best and
   wasn't overfitting the Data. Hence, no penalty should be added. So, the value of Tuning Parameter, λ

must be low.

Plots for RMSE (Training + Validation) v/s iterations, for best value of Regularization parameter, λ



Observations in the plots of the RMSE V/s iteration graph for Training & Testing Data:

1) The value of RMSE for both Training & Testing Data decreases with the iterations. This indicates that RMSE is converging to global minimum, with increase in iterations.

2) The plot of the RMSE V/s iteration graph for Training & Testing Data shows the similar behaviour for all the 'k' models trained.

Implementing LASSO Regression from scratch

In code, LASSO Regularization for Linear Regression Model is inherited from the earlier Linear Regression class. Only neccessary methods (for Cost function, RMSE calculation, Training model) have been overridden.

Similarly, LASSO Regularization for Gradient Descent is inherited from the earlier Gradient Descent class. Only one neccessary method (for calculating Derivative of Cost function) is overridden.

In [ ]:

```
# Linear Model for LASSO Regression
class LinearRegressionModel_LASSO(LinearRegressionModel):
    def __init__(self, n, λ) -> None:
        super().__init__(n)
```

```
        self.λ = λ          # Lambda parameter for LASSO Linear Regression
        self.parameters = np.ones(n)

    # Calculate the Cost function, J(h) of the model in the prediction of House price, h(x)
    def costFunction(self, x, y):
        # Penaly term in LASSO Regression
        penalty_cost_function = self.λ * np.sum(np.abs(self.parameters))
        return super().costFunction(x, y) + penalty_cost_function

    # Root Mean Square Error is Square root of twice of Cost function
    def calculateRMSE(self, x, y):
        return np.sqrt(2 * super().costFunction(x, y))

    # Train the Model on the Testing Dataset (x_train,y_train)
    def fit(self, x_train, y_train, x_test, y_test):
        algorithm = GradientDescent_LASSO(0.03, 1000)
        return algorithm.runGradientDesecent(self, x_train, y_train, x_test, y_test)


# Gradient Descent for LASSO Linear Regression
class GradientDescent_LASSO(GradientDescent):
    def __init__(self, learningRate, epochs) -> None:
        super().__init__(learningRate, epochs)

    # Compute the Derivative of the Cost function of Linear Regression Model, which is used in
    # Gradient Descent Algorithm. Only this method will change in LASSO Regularization
    def getDerivativeOfCostFunction(self, model: LinearRegressionModel_LASSO, x_train, y_train):
        # Prediction error in training the model (y[i] - h(x[i]))
        prediction_error = model.predict(x_train) - y_train

        # Change in Parameter
        change_in_parameter = np.dot(x_train.T, prediction_error) / len(x_train)

        # Computing new model parameter, by just implementing the Theoretical formulaes
        for j in range(len(model.parameters)):
            # Implementing thr Theory formulae
            if model.parameters[j] > model.λ:
                change_in_parameter[j] -= model.λ
            elif model.parameters[j] < -model.λ:
                change_in_parameter[j] += model.λ
            else:
                change_in_parameter[j] = 0
        return change_in_parameter
```

Conclusion after performing LASSO Regression for various values of Hyperparameter (λ).

Finally, the Best value of λ is reported as follows for optimal value of k in K-fold cross Validation

In [ ]:

```
def plot__Training_and_Validation_RMSE_for_LASSO_Regression(k, λ):
    print("******************** Tuning Parameter, λ in LASSO Regression:", λ, "************************")
    fold_size = len(df) // k        # size of Dataset in each fold of size 'k'

    global train_rmse, validation_rmse
    train_rmse, validation_rmse = [], []

    # Creates 'k' different Training & Validation Dataset
    for fold in range(1, k+1):
        # Training and Validation set for the each fold
        x_train, y_train, x_valid, y_valid = getTrainAndTestData(fold, fold_size)

        # initialize a Linear regression model
        ridge_reg_model = LinearRegressionModel_LASSO(len(x_train.columns), λ)

        # Train Model and run gradient descent algorithm it, to find its Model parameters, w
        __train_rmse, __validation_rmse = ridge_reg_model.fit(x_train, y_train, x_valid, y_valid)
        train_rmse.append(__train_rmse)
        validation_rmse.append(__validation_rmse)

        # RMSE of Training & Validation set for each model trained
```

```
        print(f"After Training the Model number {fold}...")
        print(f"The RMSE Training Error : {ridge_reg_model.calculateRMSE(x_train, y_train)}")
        print(f"The RMSE Validation Error : {ridge_reg_model.calculateRMSE(x_valid, y_valid)}")
        print()


# RMSE for Training and Validation set
train_rmse, validation_rmse = [], []

# optimal value of K for "K-Fold cross-validation" chosen is 5
k = 5
λ = [1, 0.1, 0.01]
for lambda_ in λ:
    plot__Training_and_Validation_RMSE_for_LASSO_Regression(k=k, λ=lambda_)

print("\nConclusion:")
print(f"So, after the tuning various values of Regularization parameter, λ, the best chosen value is {λ[-1]}.")
print("Reasons:")
print("1) The value of λ which gives least RMSE and the least variance in all the models is: ", λ[-1])
print(f"'2) This value of λ={λ[-1]} is chosen beacuse, the model was already performing Best and
   wasn't overfitting the Data. Hence, no penalty should be added. So, the value of Tuning Parameter, λ
   must be low.'")


# Subplot for RMSE for Best value of Hyperparamter, λ
fig, axes = plt.subplots(3, 2, dpi=200, figsize=(17,16))
fig.subplots_adjust(wspace=0.3, hspace=0.4)
for i in range(k):
    axes[i//2][i%2].plot(train_rmse[i], 'r-',label='Training RMSE plot')
    axes[i//2][i%2].plot(validation_rmse[i], 'b-',label='Validation RMSE plot')
    axes[i//2][i%2].set_title(f"'LASSO Regression (λ={λ[-1]}) plot for Model no. {i+1}
RMSE (Training & Validation) v/s iteration plot'")
    axes[i//2][i%2].set_xlabel(f'No. of iterations')
    axes[i//2][i%2].set_ylabel('Root Mean Square Error')
    axes[i//2][i%2].legend()
    axes[i//2][i%2].grid(True)
```

********** Tuning Parameter, λ in LASSO Regression: 1 **********
After Training the Model number 1...
The RMSE Training Error : 39.13880265128961
The RMSE Validation Error : 39.29489369386367

After Training the Model number 2...
The RMSE Training Error : 38.82551015677518
The RMSE Validation Error : 40.51609770408358

After Training the Model number 3...
The RMSE Training Error : 39.43242639668804
The RMSE Validation Error : 38.073565077494536

After Training the Model number 4...
The RMSE Training Error : 39.119057687581254
The RMSE Validation Error : 39.378111275625926

After Training the Model number 5...
The RMSE Training Error : 39.49572449292626
The RMSE Validation Error : 37.826540867962635

********** Tuning Parameter, λ in LASSO Regression: 0.1 **********
After Training the Model number 1...
The RMSE Training Error : 10.061229127673114
The RMSE Validation Error : 8.358058996168797

After Training the Model number 2...
The RMSE Training Error : 9.678930198387672
The RMSE Validation Error : 9.988550875347684

After Training the Model number 3...
The RMSE Training Error : 9.981923300925903
The RMSE Validation Error : 8.622619748066045

After Training the Model number 4...
The RMSE Training Error : 8.951785245217811
The RMSE Validation Error : 12.445301946154688

After Training the Model number 5...
The RMSE Training Error : 9.95252192619632
The RMSE Validation Error : 8.998357625316057

********** Tuning Parameter, λ in LASSO Regression: 0.01 **********
After Training the Model number 1...
The RMSE Training Error : 9.161539668851143
The RMSE Validation Error : 7.11185529487142

After Training the Model number 2...
The RMSE Training Error : 8.651360242829623
The RMSE Validation Error : 9.431219265310768

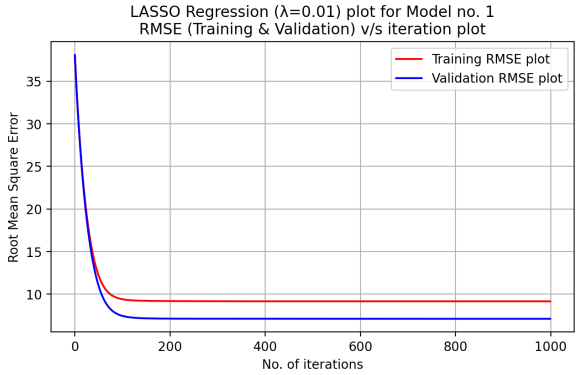
After Training the Model number 3...
The RMSE Training Error : 9.04658075684748
The RMSE Validation Error : 7.772669925764314

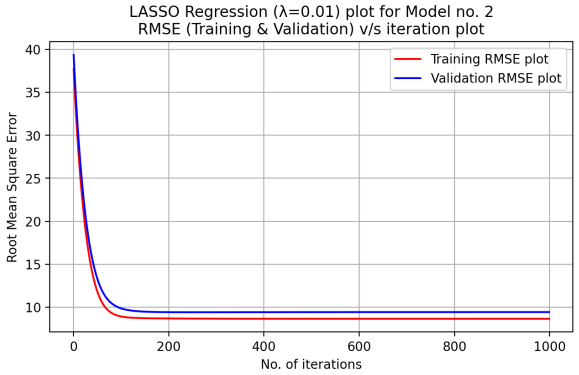
After Training the Model number 4...
The RMSE Training Error : 7.937431947598227
The RMSE Validation Error : 11.604964124605749

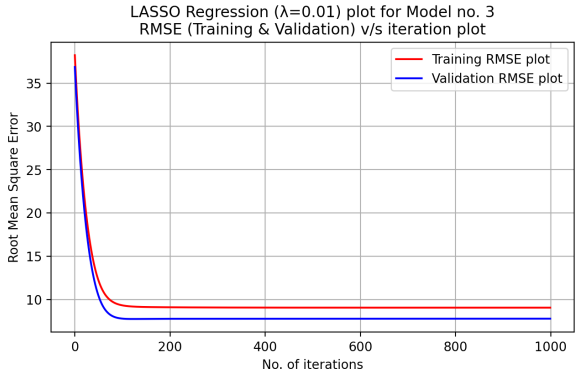
After Training the Model number 5...
The RMSE Training Error : 9.015540177112097
The RMSE Validation Error : 7.877516997035123

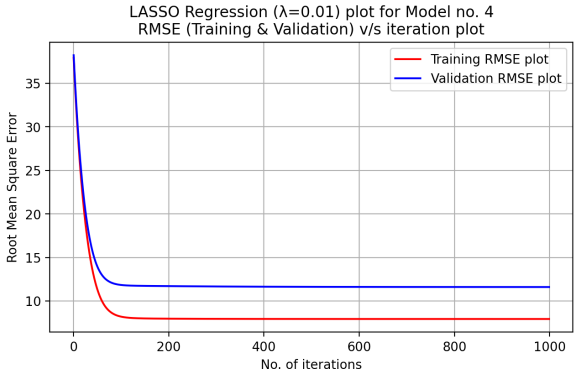
Conclusion:
So, after the tuning various values of Regularization parameter, λ, the best chosen value is 0.01.
Reasons:
1) The value of λ which gives least RMSE and the least variance in all the models is: 0.01
2) This value of λ=0.01 is chosen beacuse, the model was already performing Best and

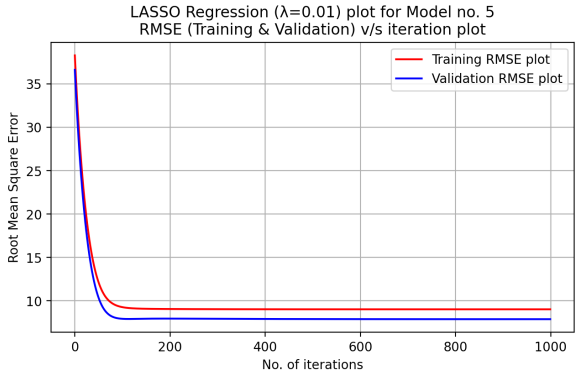
wasn't overfitting the Data. Hence, no penalty should be added. So, the value of Tuning Parameter, λ must be low.

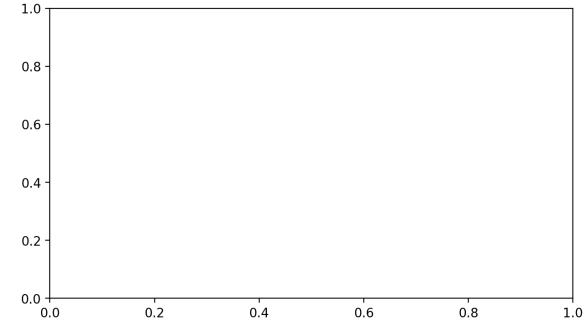












Observations in the plots of the RMSE V/s iteration graph for Training & Testing Data:

1) The value of RMSE for both Training & Testing Data decreases with the iterations. This indicates that RMSE is converging to global minimum, with increase in iterations.

2) The plot of the RMSE V/s iteration graph for Training & Testing Data shows the similar behaviour for all the 'k' models trained.

Solution for Q2 (d)

Normal Equation:

Given Training Dataset, x and its output label y.

Its model parameters, w are given by:

w = inverse((X' *X)) (X' * Y)*

where X' --> Transpose of X

In [ ]:

```
# Optimal value of k is 5
k = 5
print(f"The optimal value of K chosen for 'K-Fold cross-validation' is : {k} \n")

# Stores the average of the Validation RMSE error for all the 'k' models
avg_validation_rmse = 0

# Creates 'k' different Training & Validation Dataset, and Train & Validate on them
for fold in range(1,k+1):
    # Training and Validation set for the each fold
    x_train, y_train, x_valid, y_valid = getTrainAndTestData(fold, len(df) // k)

    # Model parameters, w in normal equation is given by: w = inverse((X.T * X)) * (X.T * Y)
    model_parameters = np.dot(np.linalg.inv(np.dot(x_train.T, x_train)), np.dot(x_train.T, y_train))

    # Predicted value of output of Validation set = h(x)
    y_valid_predicted = np.dot(x_valid, model_parameters)

    # Predicted error = difference of predicted value and actual value  = h(x) - y_valid
    prediction_error = y_valid_predicted - y_valid

    # RMSE = (sum_over_all_samples(square_of_prediction_error))^0.5
    RMSE = np.sqrt(np.dot(prediction_error.T, prediction_error) / len(y_valid))

    print(f"Training Model number {fold}...")
    print(f"The RMSE Validation Error : {(RMSE):.4f}")
    print()

    # Sum up all the RMSE (to take average later)
    avg_validation_rmse += RMSE

print(f"The average of RMSE Validation Error for all models is: {(avg_validation_rmse / k):.4f}")
```

```
The optimal value of K chosen for 'K-Fold cross-validation' is : 5

Training Model number 1...
The RMSE Validation Error : 7.1312

Training Model number 2...
The RMSE Validation Error : 9.4368

Training Model number 3...
The RMSE Validation Error : 7.7777

Training Model number 4...
The RMSE Validation Error : 11.6010

Training Model number 5...
The RMSE Validation Error : 7.8752

The average of RMSE Validation Error for all models is: 8.7644
```

Section - C

(Algorithm implementation using packages)

Q3

Section-C

In [ ]:
```
import numpy as np
from matplotlib import pyplot as plt
import pandas as pd
import warnings
warnings.filterwarnings("ignore")
```

In [ ]:
```
df = pd.read_excel("Dry_Bean_Dataset.xlsx", sheet_name='Dry_Beans_Dataset')
```

In [ ]:
```
df.head()
```

Out[ ]:

	Area	Perimeter	MajorAxisLength	MinorAxisLength	AspectRation	Eccentricity	ConvexArea	EquivDiameter	Extent
0	28395	610.291	208.178117	173.888747	1.197191	0.549812	28715	190.141097	0.763923
1	28734	638.018	200.524796	182.734419	1.097356	0.411785	29172	191.272750	0.783968
2	29380	624.110	212.826130	175.931143	1.209713	0.562727	29690	193.410904	0.778113
3	30008	645.884	210.557999	182.516516	1.153638	0.498616	30724	195.467062	0.782681
4	30140	620.134	201.847882	190.279279	1.060798	0.333680	30417	195.896503	0.773098

In [ ]:
```
df.info()
```

```
<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
```

In [ ]:
```
df.columns
```

Out[ ]:
```
Index(['Area', 'Perimeter', 'MajorAxisLength', 'MinorAxisLength',
       'AspectRation', 'Eccentricity', 'ConvexArea', 'EquivDiameter', 'Extent',
       'Solidity', 'roundness', 'Compactness', 'ShapeFactor1', 'ShapeFactor2',
       'ShapeFactor3', 'ShapeFactor4', 'Class'],
      dtype='object')
```

In [ ]:
```
df.describe()
```

Out[ ]:

	Area	Perimeter	MajorAxisLength	MinorAxisLength	AspectRation	Eccentricity	ConvexArea	Equiv
count	13611.000000	13611.000000	13611.000000	13611.000000	13611.000000	13611.000000	13611.000000	136
mean	53048.284549	855.283459	320.141867	202.270714	1.583242	0.750895	53768.200206	2
std	29324.095717	214.289696	85.694186	44.970091	0.246678	0.092002	29774.915817	
min	20420.000000	524.736000	183.601165	122.512653	1.024868	0.218951	20684.000000	1
25%	36328.000000	703.523500	253.303633	175.848170	1.432307	0.715928	36714.500000	2
50%	44652.000000	794.941000	296.883367	192.431733	1.551124	0.764441	45178.000000	2
75%	61332.000000	977.213000	376.495012	217.031741	1.707109	0.810466	62294.000000	2
max	254616.000000	1985.370000	738.860153	460.198497	2.430306	0.911423	263261.000000	5

In [ ]:
```
df.shape # 17 columns and 13611 rows
```

Out[ ]:
```
(13611, 17)
```

In [ ]:
```
df['Class'].value_counts()
```

Out[ ]:
```
DERMASON    3546
SIRA        2636
SEKER       2027
HOROZ       1928
CALI        1630
BARBUNYA    1322
BOMBAY       522
Name: Class, dtype: int64
```

In [ ]:
```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt
```

Solution for Q3 (a)

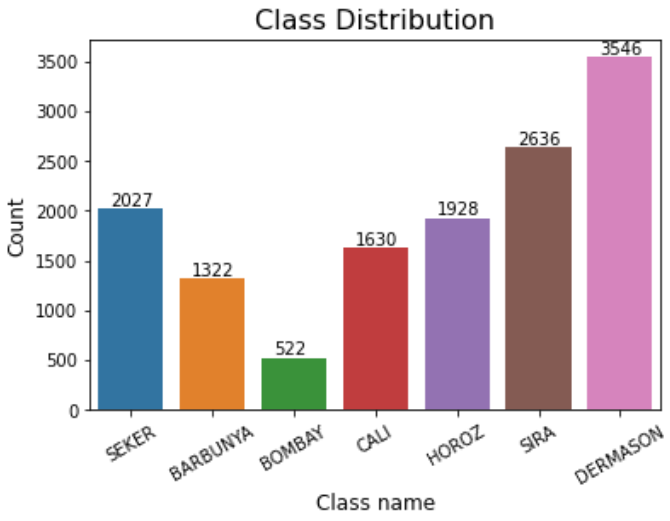
Class distribution:

There are 7 categories in 'Class'

The count of each Class is shown as count in Bar plot

Beans with belonging to class 'DERMASON' and 'BOMBAY' occurs most and least frequently in this Dataset respectively.

In [ ]:
```
axes = sns.countplot(x=df['Class'])
axes.set_title("Class Distribution", fontsize=16)
axes.set_xlabel("Class name", fontsize=12)
axes.set_ylabel("Count", fontsize=12)
axes.set_xticklabels(axes.get_xticklabels(), rotation=30)

for bar in axes.patches:
    axes.annotate(bar.get_height(), xy=(bar.get_x()+0.15, bar.get_height()+30))

plt.show()
```



Solution for Q3 (b)

Performing EDA on Dataset

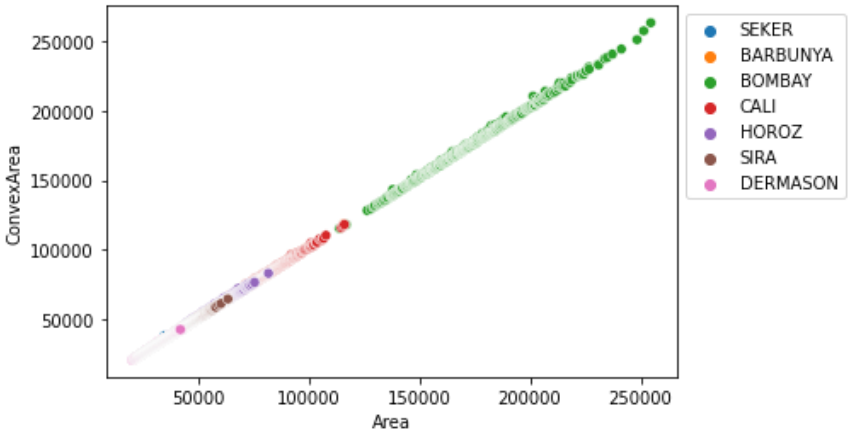
In [ ]:
```
df.corr()
```

Out[ ]:

	Area	Perimeter	MajorAxisLength	MinorAxisLength	AspectRation	Eccentricity	ConvexArea	EquivI
Area	1.000000	0.966722	0.931834	0.951602	0.241735	0.267481	0.999939	
Perimeter	0.966722	1.000000	0.977338	0.913179	0.385276	0.391066	0.967689	
MajorAxisLength	0.931834	0.977338	1.000000	0.826052	0.550335	0.541972	0.932607	
MinorAxisLength	0.951602	0.913179	0.826052	1.000000	-0.009161	0.019574	0.951339	
AspectRation	0.241735	0.385276	0.550335	-0.009161	1.000000	0.924293	0.243301	
Eccentricity	0.267481	0.391066	0.541972	0.019574	0.924293	1.000000	0.269255	
ConvexArea	0.999939	0.967689	0.932607	0.951339	0.243301	0.269255	1.000000	
EquivDiameter	0.984968	0.991380	0.961733	0.948539	0.303647	0.318667	0.985226	
Extent	0.054345	-0.021160	-0.078062	0.145957	-0.370184	-0.319362	0.052564	
Solidity	-0.196585	-0.303970	-0.284302	-0.155831	-0.267754	-0.297592	-0.206191	-
roundness	-0.357530	-0.547647	-0.596358	-0.210344	-0.766979	-0.722272	-0.362083	-
Compactness	-0.268067	-0.406857	-0.568377	-0.015066	-0.987687	-0.970313	-0.269922	-
ShapeFactor1	-0.847958	-0.864623	-0.773609	-0.947204	0.024593	0.019920	-0.847950	-
ShapeFactor2	-0.639291	-0.767592	-0.859238	-0.471347	-0.837841	-0.860141	-0.640862	-
ShapeFactor3	-0.272145	-0.408435	-0.568185	-0.019326	-0.978592	-0.981058	-0.274024	-
ShapeFactor4	-0.355721	-0.429310	-0.482527	-0.263749	-0.449264	-0.449354	-0.362049	-

Scatter Plots on features 'Area' & 'ConvexArea', denoting correlation between them

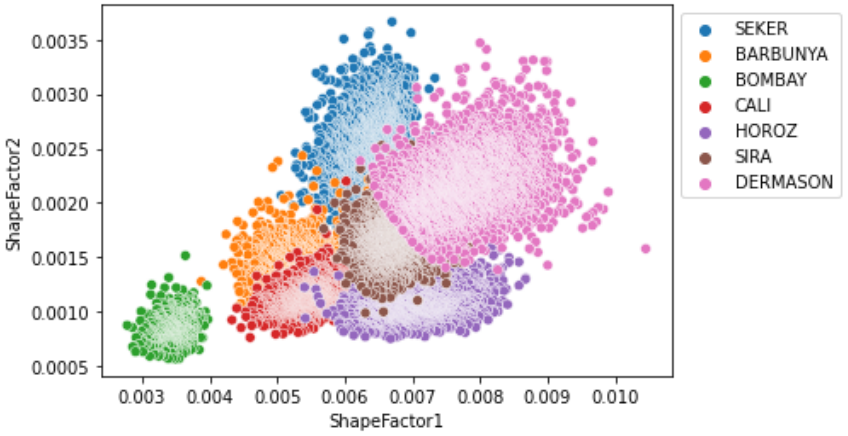
In [ ]:
```
sns.scatterplot(x='Area', y='ConvexArea',hue='Class', data=df,marker='o')
plt.legend(bbox_to_anchor=(1, 1), loc=2)
plt.show()
```



Scatter Plots on features 'ShapeFactor1' & 'ShapeFactor2', denoting correlation between them

In [ ]:

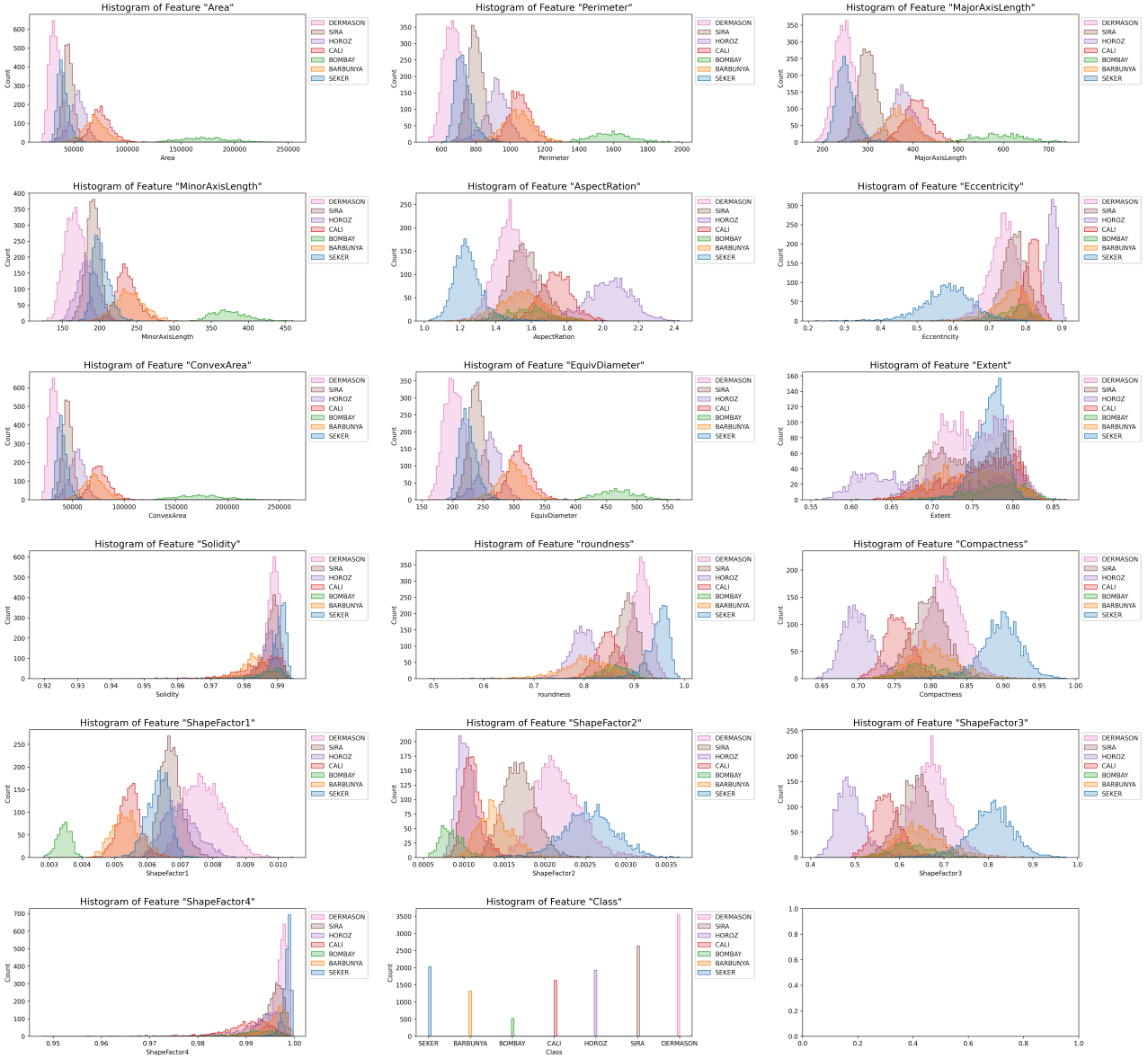
```
sns.scatterplot(x='ShapeFactor1', y='ShapeFactor2',hue='Class', data=df,marker='o')
plt.legend(bbox_to_anchor=(1, 1), loc=2)
plt.show()
```



Histograms on all the features, over every class

In [ ]:

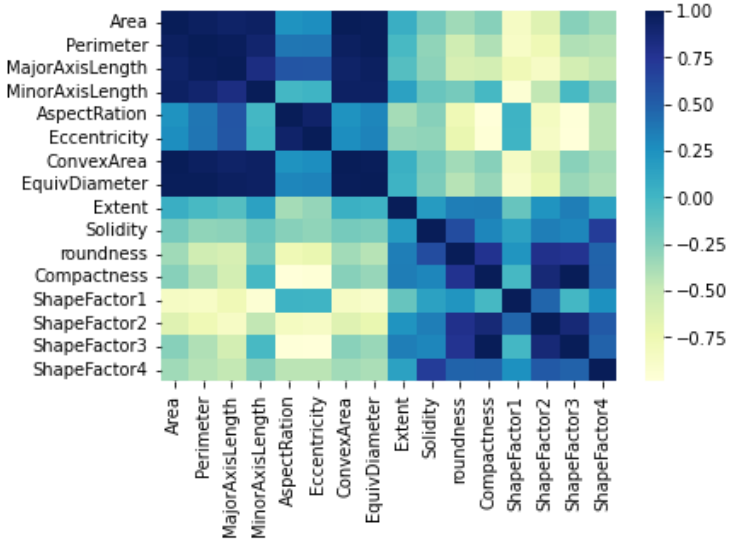
```
fig, axes = plt.subplots(6, 3, figsize=(30,30),dpi=250)
fig.subplots_adjust(wspace=0.4, hspace=0.4)
column = 0

for i in range(len(axes)):
    for j in range(len(axes[i])):
        if (column == len(df.columns)):
            break
        sns.histplot(x=df.columns[column], data=df, hue='Class', bins=100,
                     ax=axes[i,j], element='step', discrete=False)
        axes[i,j].set_title(f'Histogram of Feature "{df.columns[column]}"', fontsize=15)
        axes[i,j].legend(df['Class'].unique()[::-1], bbox_to_anchor=(1, 1), loc=2)
        column += 1
```



Heatmap (illustrating correlation between various features)

In [ ]:

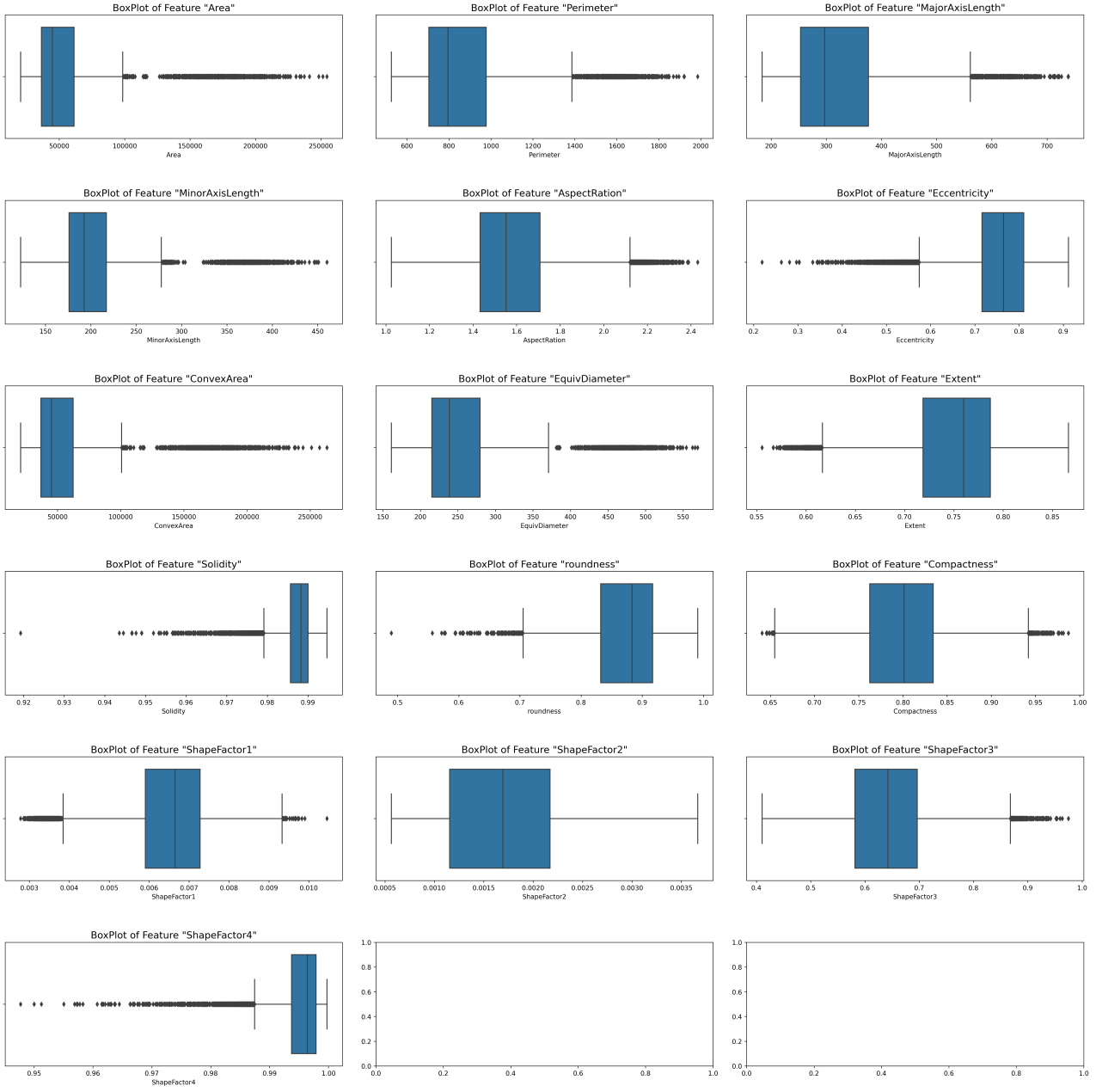
```
sns.heatmap(df.corr(method='pearson'), cmap='YlGnBu')
plt.show()
```



Boxplots on all the features (denoting the outliers in each feature)

In [ ]:

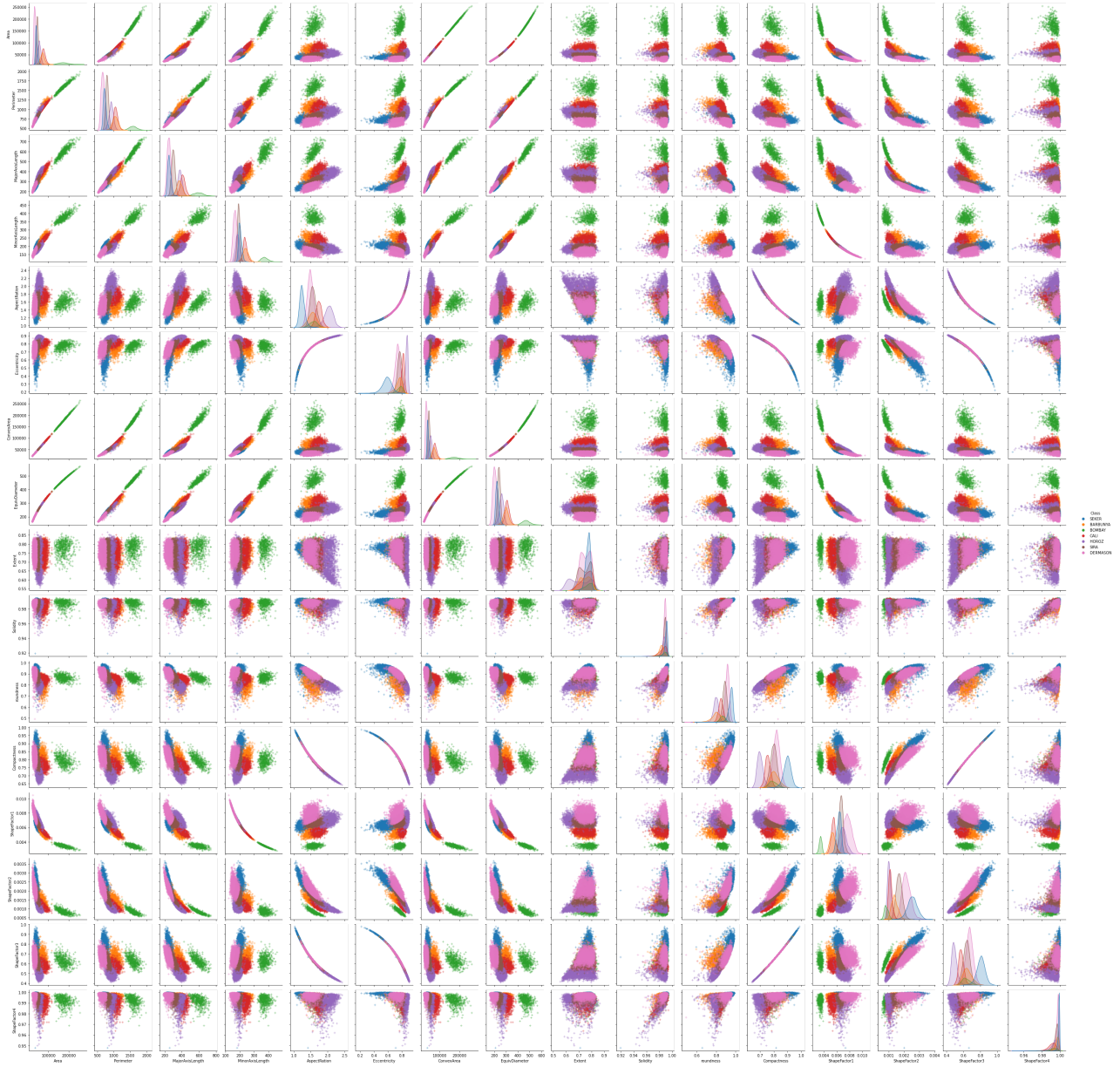
```
fig, axes = plt.subplots(6, 3, figsize=(30,30),dpi=250)
fig.subplots_adjust(wspace=0.1, hspace=0.5)
column = 0

for i in range(len(axes)):
    for j in range(len(axes[i])):
        if (column == len(df.columns)-1):
            break
        sns.boxplot(x=df.columns[column], data=df, ax=axes[i,j])   # outlier
        axes[i,j].set_title(f'BoxPlot of Feature "{df.columns[column]}"', fontsize=15)
        column += 1
```



Pairplot on the entire Dataset (illustrating correlation b/w every pair of features)

In [ ]:

```
sns.pairplot(data=df, hue='Class', markers='+');
```

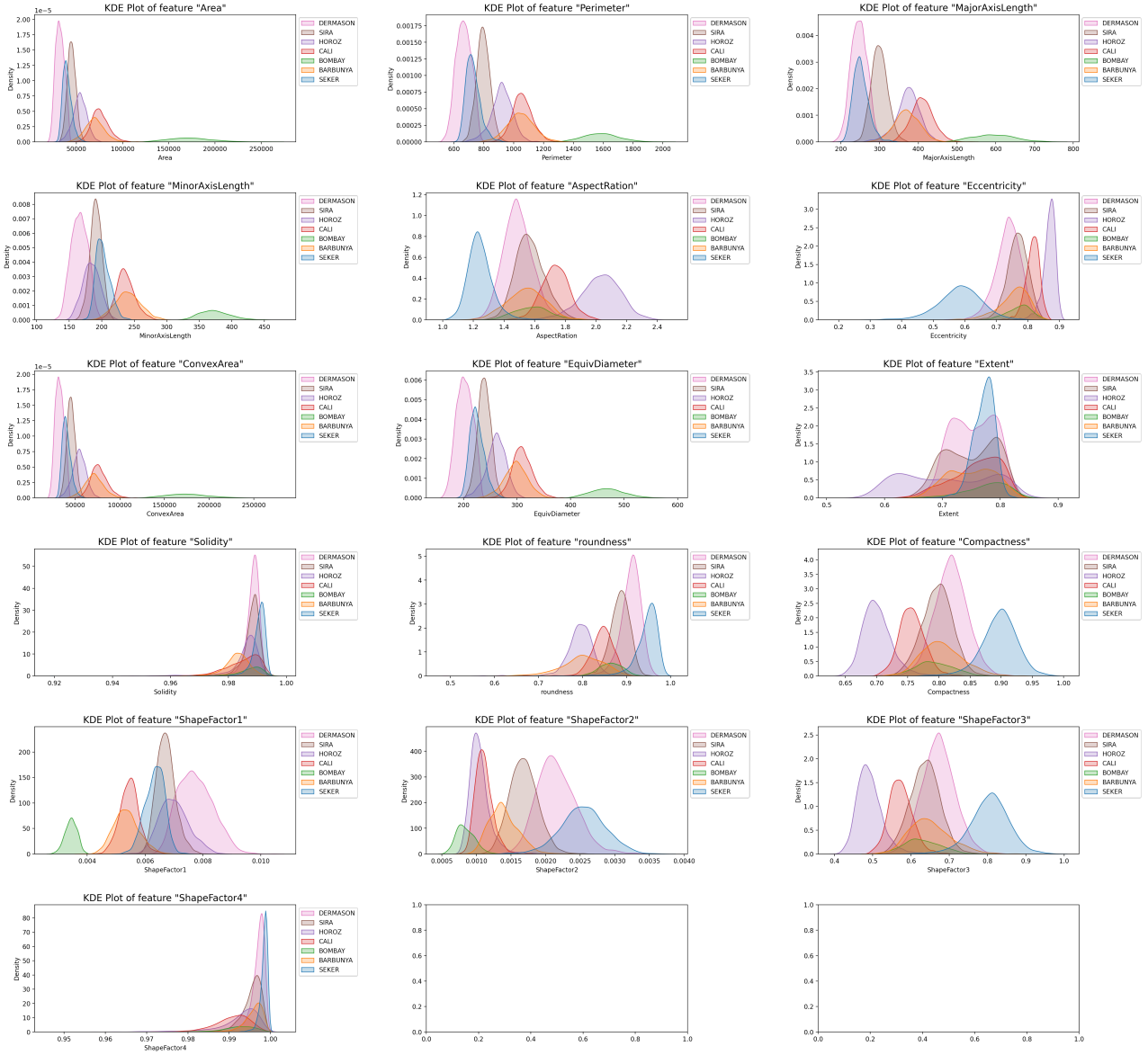


KDE plot for every feature (denoting the densities of features over various classes)

It provides same information as Hitograms, except count information

In [ ]:

```
fig, axes = plt.subplots(6, 3, figsize=(30,30),dpi=200)
fig.subplots_adjust(wspace=0.5, hspace=0.4)
column = 0

for i in range(len(axes)):
    for j in range(len(axes[i])):
        if (column == len(df.columns)-1):
            break
        sns.kdeplot(data=df, x=df.columns[column], hue='Class', ax=axes[i,j], fill=True)
        axes[i,j].set_title(f'KDE Plot of feature "{df.columns[column]}"', fontsize=15)
        axes[i,j].legend(df['Class'].unique()[::-1], bbox_to_anchor=(1, 1), loc=2)
        column += 1
```



Checking Missing values in Dataset

In [ ]:

```
print("Count of null values in each feature/column:")
df.isnull().sum()
```

Out[ ]:

```
Count of null values in each feature/column:
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
```

Several insights on this dataset are:

1) Beans with belonging to class 'DERMASON' and 'BOMBAY' occurs most and least frequently in this Dataset respectively.

2) Features 'Area' and 'ConvexArea' have very high positive correlation, as shown by scatterplot. So, the presence of both these features together will be redundant. So, either of these can be dropped.

3) All the features related to size, area & dimensions (area, perimeter, majoraxislength, minoraxislength, convexarea, equivdiameter) have high positive correlation with each other, as shown by Heatmap.

4) All the features related to Shape factors (Shapefactor 1,2,3,4, roundness, compactness) have negative correlation with the features related to size, area & dimensions.

5) From histogram plots, we can say 'BOMBAY' Class beans have greatest Size, Area & Dimensions (area, perimeter, majoraxislength, minoraxislength, convexarea, equivdiameter), than every other Beans (of any other class).

```
Similarly, 'DERMASON' Class beans have smallest Size, Area & Dimensions, than every
other Beans (of any other class).
```

6) From the Histogram plot of feature "Solidity", we can say that Beans of every 'Class' have nearly ythe same solidity, and are highly solid.

7) All the features related to size, area & dimensions (area, perimeter, majoraxislength, minoraxislength, convexarea, equivdiameter) have large no. of outliers (that are high valued).

8) Since there are large no. of Beans with 'DERMASON' class. So, the trained model will show low bias & low variance on testing data, rich in Beans with 'DERMASON' class.

9) Since there are fewer Beans with 'BOMBAY' class. So, the trained model will show high bias on testing data, rich in Beans with 'BOMBAY' class.

Solution for Q3 (c)

Separating Input Dataset and Output labels/classes

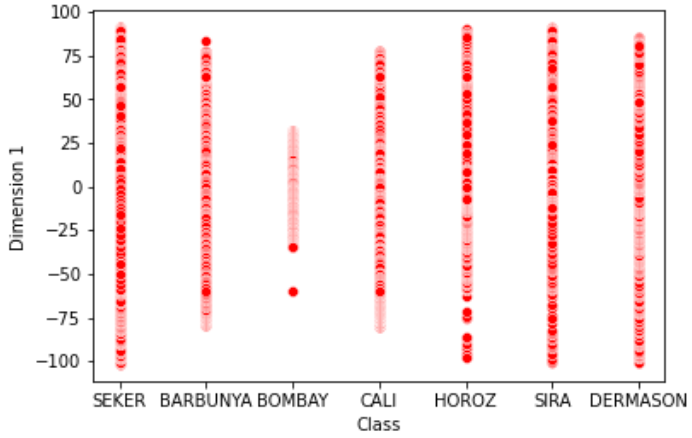
In [ ]:
```
x = df.drop('Class', axis=1)
y = df['Class']
```

In [ ]:
```
from sklearn.manifold import TSNE
```

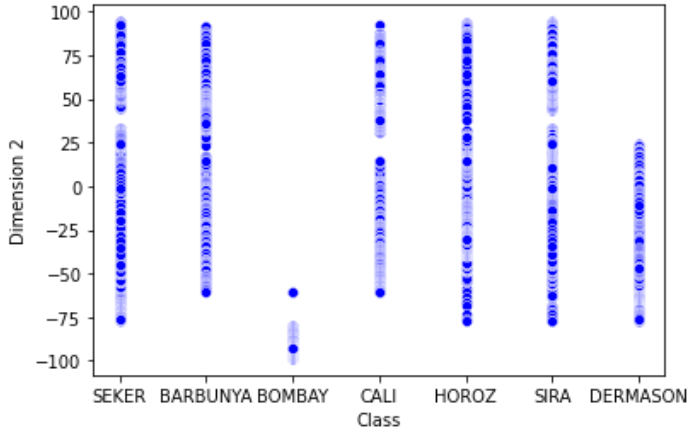
In [ ]:
```
model = TSNE(n_components=2, random_state=49)
TSNE_Dataset = model.fit_transform(x)
```

In [ ]:
```
tsne_data = pd.DataFrame(data=TSNE_Dataset, columns=['Dimension 1','Dimension 2'])
tsne_data['Class'] = df['Class']
```

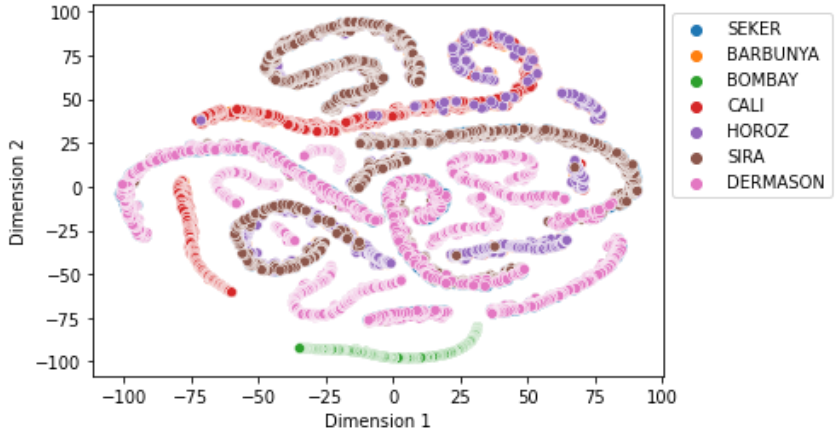
In [ ]:
```
sns.scatterplot(x='Class',y='Dimension 1',data=tsne_data,color='red')
plt.show()
```



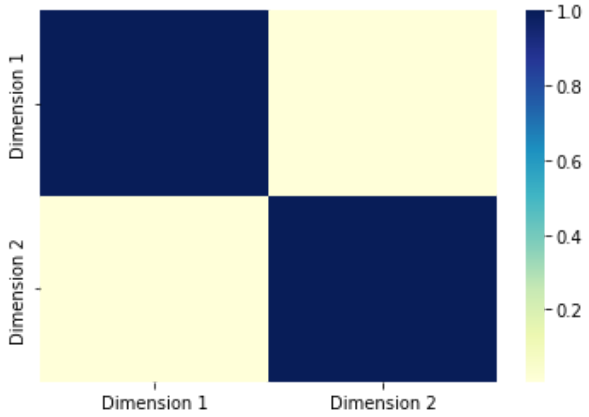
In [ ]:
```
sns.scatterplot(x='Class',y='Dimension 2',data=tsne_data,color='blue')
plt.show()
```



In [ ]:
```
sns.scatterplot(x='Dimension 1', y='Dimension 2', hue='Class', data=tsne_data)
plt.legend(bbox_to_anchor=(1, 1), loc=2)
plt.show()
```



In [ ]:
```
sns.heatmap(tsne_data.corr(), cmap='YlGnBu');
```



Comment on the separability of Data:

After reducing Data dimension to 2 features, using TSNE.

1) We can see From Scatter plots and Heatmap, both of these two (reduced) dimensions/features have zero correlation with each other.

2) So, we conclude TSNE reduces the dimensions of original dataset into new uncorrelated dimensions.

Solution for Q3 (d)

In [ ]:
```
from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB
from sklearn.model_selection import train_test_split
from sklearn import metrics
```

Running Naive Bayes on Actual Dataset

In [ ]:
```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
```

In [ ]:
```
gaussian_nb = GaussianNB()
gaussian_nb.fit(x_train, y_train)
y_prediction = gaussian_nb.predict(x_test)

# Accuracy of Model
accuracy = metrics.accuracy_score(y_test, y_prediction)

# Precision of Model
precision = metrics.precision_score(y_test, y_prediction,average='macro')

# Recall of model
recall = metrics.recall_score(y_test, y_prediction,average='macro')

print("Statistics for 'Gaussian Naive Bayes' algorithm:")
print(f"Accuracy in Prediction: {accuracy}")
print(f"Precision in Prediction: {precision}")
print(f"Recall in Prediction: {recall}")
```

```
Statistics for 'Gaussian Naive Bayes' algorithm:
Accuracy in Prediction: 0.7690047741461623
Precision in Prediction: 0.7709507753500044
Recall in Prediction: 0.7712278082780358
```

In [ ]:
```
multinomial_nb = MultinomialNB()
multinomial_nb.fit(x_train, y_train)
y_prediction = multinomial_nb.predict(x_test)

# Accuracy of Model
accuracy = metrics.accuracy_score(y_test, y_prediction)
```

```
# Precision of Model
precision = metrics.precision_score(y_test, y_prediction,average='macro')

# Recall of model
recall = metrics.recall_score(y_test, y_prediction,average='macro')

print("Statistics for 'Multinomial Naive Bayes' algorithm:")
print(f"Accuracy in Prediction: {accuracy}")
print(f"Precision in Prediction: {precision}")
print(f"Recall in Prediction: {recall}")
```

```
Statistics for 'Multinomial Naive Bayes' algorithm:
Accuracy in Prediction: 0.7903048108703635
Precision in Prediction: 0.796291029886272
Recall in Prediction: 0.7944391671238085
```

Results:

1) Multinomial Naive Bayes shows somewhat greater accuracy over the Gaussian implementation of Naive Bayes.

2) Multinomial implementation of Naive Bayes have roughly the same accuracy, precision and recall.

3) Also, Gaussian implementation of Naive Bayes have roughly the same accuracy, precision and recall.

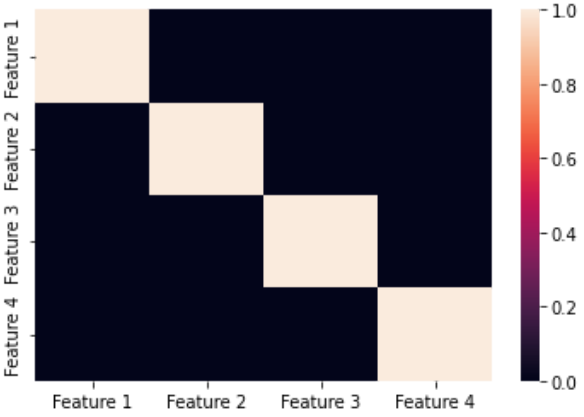
Solution for Q3 (e)

In [ ]:
```
from sklearn.decomposition import PCA
```

In [ ]:
```
for features in [4,6,8,10,12]:
    print(">>> ****************** PCA with no. of features reduced to",features, "****************** ")
    pca = PCA(n_components=features)
    pca.fit(x)
    pca_data = pca.transform(x)

    pca_data = pd.DataFrame(pca_data, columns=[f"Feature {i}" for i in range(1,features+1)])

    # Applying "Gaussian Naive Bayes" on output of PCA
    x_train, x_test, y_train, y_test = train_test_split(pca_data, y, test_size=0.2)

    gaussian_nb = GaussianNB()
    gaussian_nb.fit(x_train, y_train)
    y_prediction = gaussian_nb.predict(x_test)

    # Accuracy, Precision, Recall, and F1 score of Model
    accuracy = metrics.accuracy_score(y_test, y_prediction)
    precision = metrics.precision_score(y_test, y_prediction,average='macro')
    recall = metrics.recall_score(y_test, y_prediction,average='macro')
    f1_score = metrics.f1_score(y_test, y_prediction,average='macro')

    print("Statistics for 'Gaussian Naive Bayes' algorithm:")
    print(f"Accuracy: {accuracy}")
    print(f"Precision: {precision}")
    print(f"Recall: {recall}")
    print(f"F1-Score: {f1_score}")
    print("\nHeatmap to illustrate the correlation between various reduced columns:")
    sns.heatmap(data=pca_data.corr())
    plt.show()
    print()
```

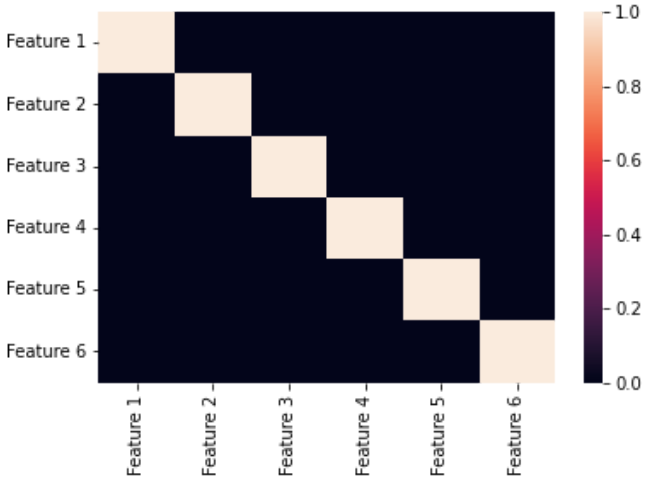
>>> ********** PCA with no. of features reduced to 4 **********
Statistics for 'Gaussian Naive Bayes' algorithm:
Accuracy: 0.9070877708409842
Precision: 0.9186925286811737
Recall: 0.9192183994393199
F1-Score: 0.9183124022939712

Heatmap to illustrate the correlation between various reduced columns:



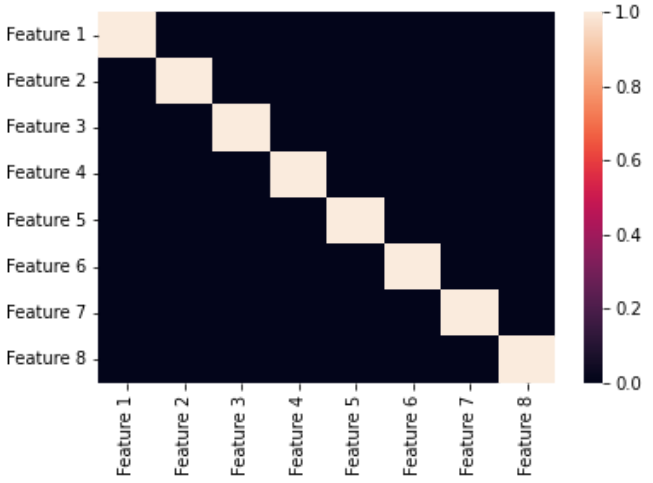
>>> ********** PCA with no. of features reduced to 6 **********
Statistics for 'Gaussian Naive Bayes' algorithm:
Accuracy: 0.8912963643040764
Precision: 0.905829703650867
Recall: 0.9097209713337772
F1-Score: 0.9066567504356888

Heatmap to illustrate the correlation between various reduced columns:



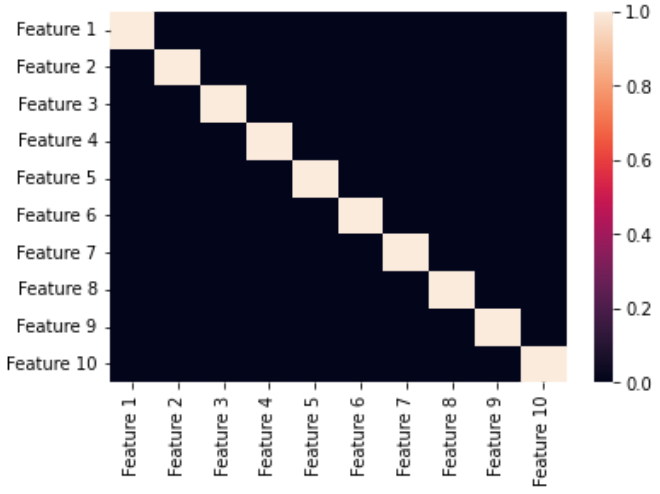
>>> ********** PCA with no. of features reduced to 8 **********
Statistics for 'Gaussian Naive Bayes' algorithm:
Accuracy: 0.8865222181417555
Precision: 0.9016329467506766
Recall: 0.9070617607824092
F1-Score: 0.9035940892538962

Heatmap to illustrate the correlation between various reduced columns:



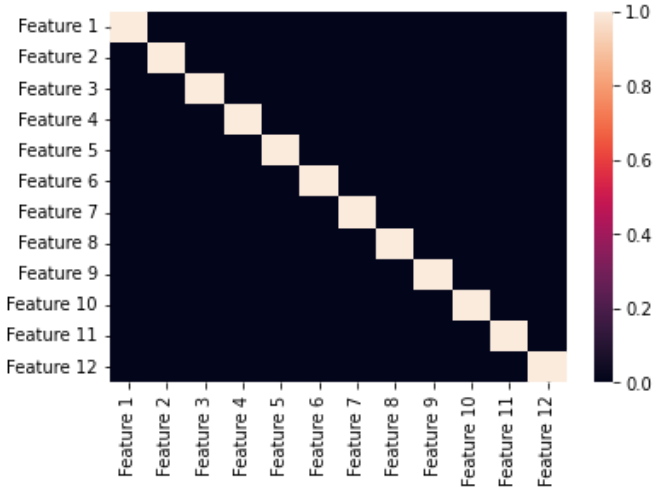
>>> ********** PCA with no. of features reduced to 10 **********
Statistics for 'Gaussian Naive Bayes' algorithm:
Accuracy: 0.8957032684539111
Precision: 0.907290510363039
Recall: 0.9114766716200879
F1-Score: 0.908190032255605

Heatmap to illustrate the correlation between various reduced columns:



>>> ********** PCA with no. of features reduced to 12 **********
Statistics for 'Gaussian Naive Bayes' algorithm:
Accuracy: 0.8982739625413148
Precision: 0.9100844093442447
Recall: 0.9131468292706685
F1-Score: 0.910889786904862

Heatmap to illustrate the correlation between various reduced columns:



Results:

1) After applying the PCA (to reduce no. of features) in the given Dataset, the Gaussian Naive Bayes models shows greater accuracy in the reduced Dataset, than the original dataset.

2) In PCA with different no. of reduced components (or featured), all the Gaussian Naive Bayes models shows nearly the same accuracy, precision, recall, and f1-score.

3) In PCA with different no. of reduced components, all the features/components have zero correlation with each other.

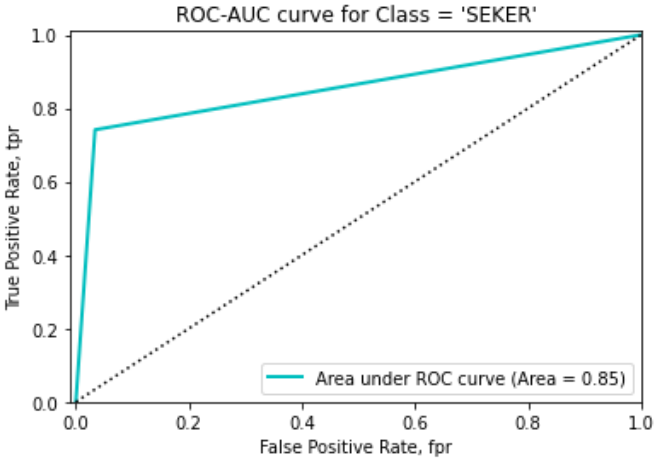
Solution for Q3 (f)

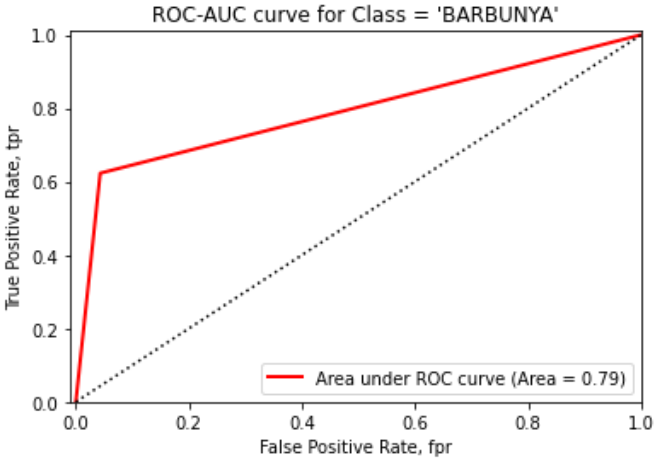
In [ ]:

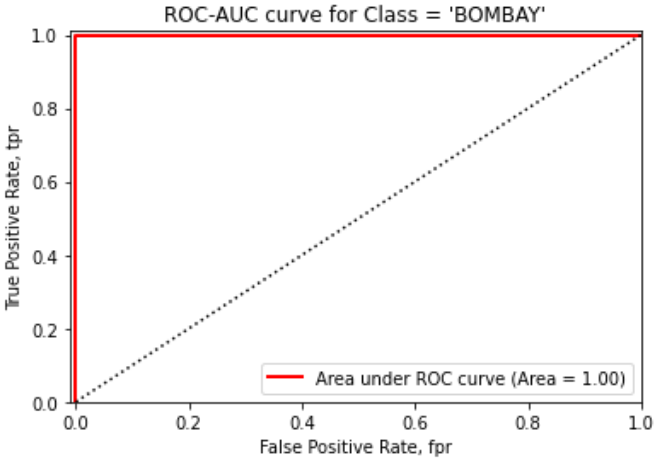
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import label_binarize
from sklearn import metrics
```

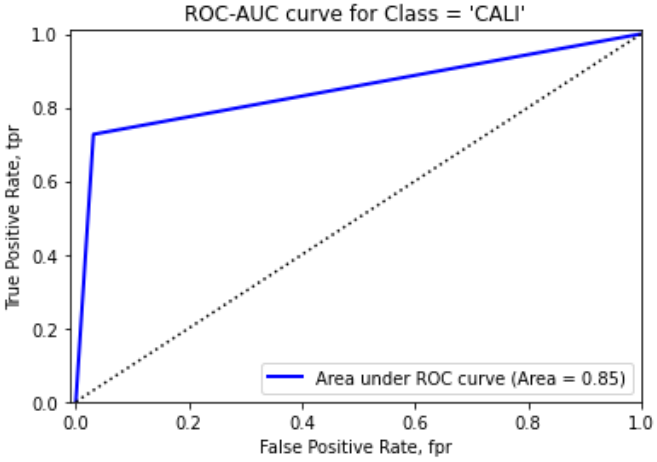
In [ ]:

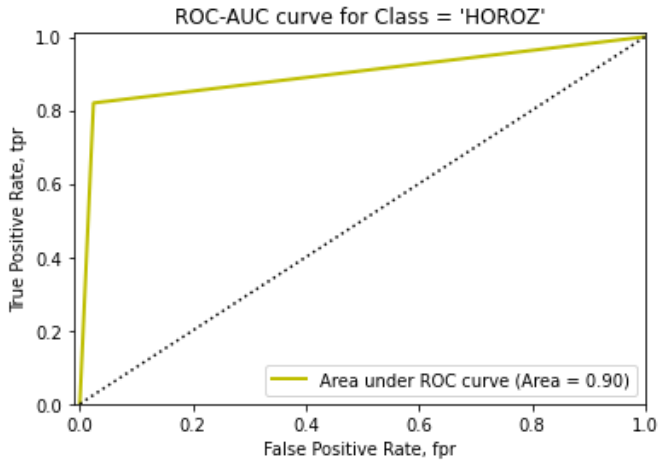
```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=5)

nb_model = MultinomialNB()
nb_model.fit(x_train, y_train)
y_pred = nb_model.predict(x_test)

num_class = y.nunique()
y_test = label_binarize(y_test, classes=y.unique())
y_pred = label_binarize(y_pred, classes=y.unique())


# Plot "ROC curve" and compute "Area under curve of ROC" for each class
color = ['r','b','g','k','m','y','c']
for j in range(num_class):
    fpr, tpr, _ = metrics.roc_curve(y_test[:, j], y_pred[:, j])
    roc_auc = metrics.auc(fpr, tpr)
    # roc_auc = metrics.roc_auc_score(y_test[:, j], y_pred[:, j])

    plt.plot(fpr, tpr, color=np.random.choice(color), lw=2, label=f'Area under ROC curve (Area = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], 'k:')
    plt.xlim([-0.01, 1])
    plt.ylim([0, 1.01])
    plt.xlabel('False Positive Rate, fpr')
    plt.ylabel('True Positive Rate, tpr')
    plt.title(f"ROC-AUC curve for Class = '{y.unique()[j]}'")
    plt.legend(loc="lower right")
    plt.show()
```

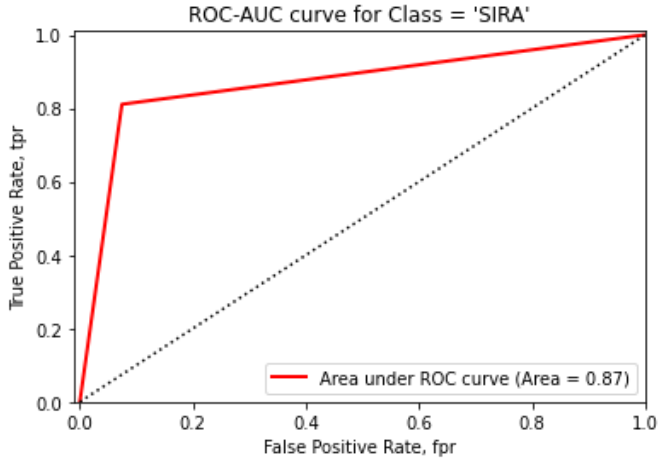

ROC-AUC curve for Class = 'SEKER'
True Positive Rate, tpr
False Positive Rate, fpr
Area under ROC curve (Area = 0.85)

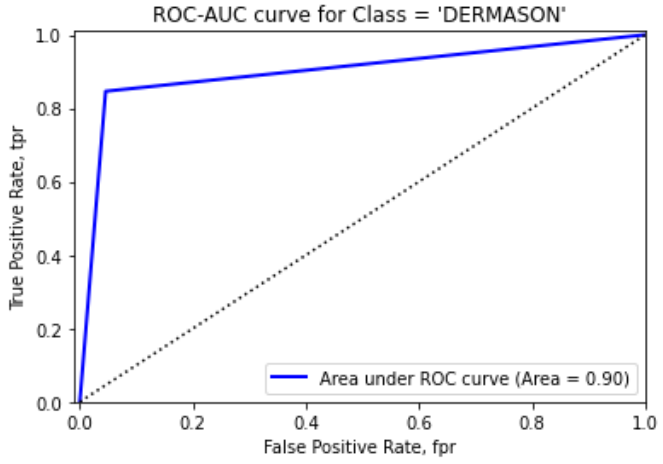

ROC-AUC curve for Class = 'BARBUNYA'
True Positive Rate, tpr
False Positive Rate, fpr
Area under ROC curve (Area = 0.79)


ROC-AUC curve for Class = 'BOMBAY'
True Positive Rate, tpr
False Positive Rate, fpr
Area under ROC curve (Area = 1.00)


ROC-AUC curve for Class = 'CALI'
True Positive Rate, tpr
False Positive Rate, fpr
Area under ROC curve (Area = 0.85)







Comment on ROC-AUC-curve Plot:

1) For a good model, the area under the curve of ROC should be close to 1. The area under ROC curve for various class distributions is observed close to 1 (0.85 +- 0.05).

2) For a model, more the ROC curve touches the Top-left corner of the plot, more better is the model in classfying between various classes/categories. We can see that the ROC curves are close to Top-left corner of the plot, hence the our gaussian naive bayes model is performing better.

Solution for Q3 (g)

In [ ]:

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn import metrics
```

Parameters chosen are:

1) multi_class="multinomial" for multiclass classification problem 2) penalty='l2' for Ridge Regression to avoid overfitting 3) max_iter=1000 for solver to converge (iterations in Gradient descent)

In [ ]:

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=5)

log_reg_model = LogisticRegression(multi_class="multinomial", penalty='l2', max_iter=2000)
log_reg_model.fit(x_train, y_train)
y_pred = log_reg_model.predict(x_test)

# Accuracy of Model
accuracy = metrics.accuracy_score(y_test, y_pred)

# Precision of Model
precision = metrics.precision_score(y_test, y_pred,average='macro')

# Recall of model
recall = metrics.recall_score(y_test, y_pred,average='macro')

print("Statistics for 'Logistic Regression' algorithm:")
print(f"Accuracy in Prediction: {accuracy}")
print(f"Precision in Prediction: {precision}")
print(f"Recall in Prediction: {recall}")
```

```
Statistics for 'Logistic Regression' algorithm:
Accuracy in Prediction: 0.9059860448035255
Precision in Prediction: 0.9176968779157052
Recall in Prediction: 0.9151517751738899
```

Results:

1) Logistic regression shows greater accuracy, precision and recall over both the Gaussian and Multinomial implementation of Naive Bayes

2) It seems Logistic regression model is a better fit on this dataset/problem, than Naive bayes model.