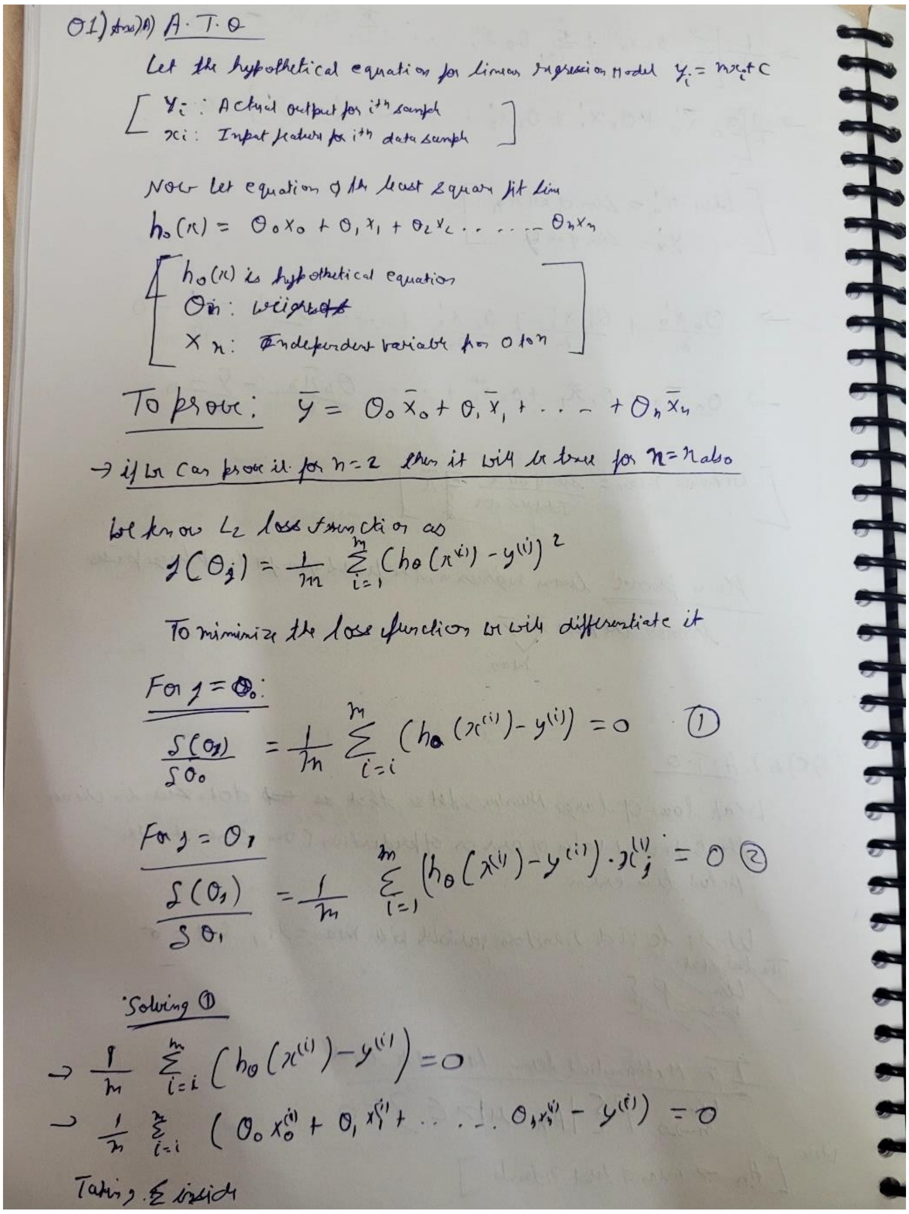
Section- A



Q1) Ans)A) A·T·Q

Let the hypothetical equation for linear regression Model $y_i = mx_i + C$

$\left[\begin{array}{l} y_i : \text{Actual output for } i^{th} \text{ sample} \\ x_i : \text{Input feature for } i^{th} \text{ data sample} \end{array} \right]$

Now let equation of the least square fit line

$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 \ldots \ldots \theta_n x_n$

$\left[\begin{array}{l} h_\theta(x) \text{ is hypothetical equation} \\ \theta_n : \text{weights} \\ x_n : \text{Independent variable for 0 to n} \end{array} \right]$

To prove: $\bar{y} = \theta_0 \bar{x}_0 + \theta_1 \bar{x}_1 + \ldots + \theta_n \bar{x}_n$

→ if we can prove it for n=2 then it will be true for n=n also

We know L_2 loss function as

$$J(\theta_j) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

To minimize the lose function we will differentiate it

For j = θ_0:

$$\frac{\delta(\theta_j)}{\delta\theta_0} = \frac{1}{m} \sum_{i=i}^{m} (h_\theta(x^{(i)}) - y^{(i)}) = 0 \quad ①$$

For j = θ_1

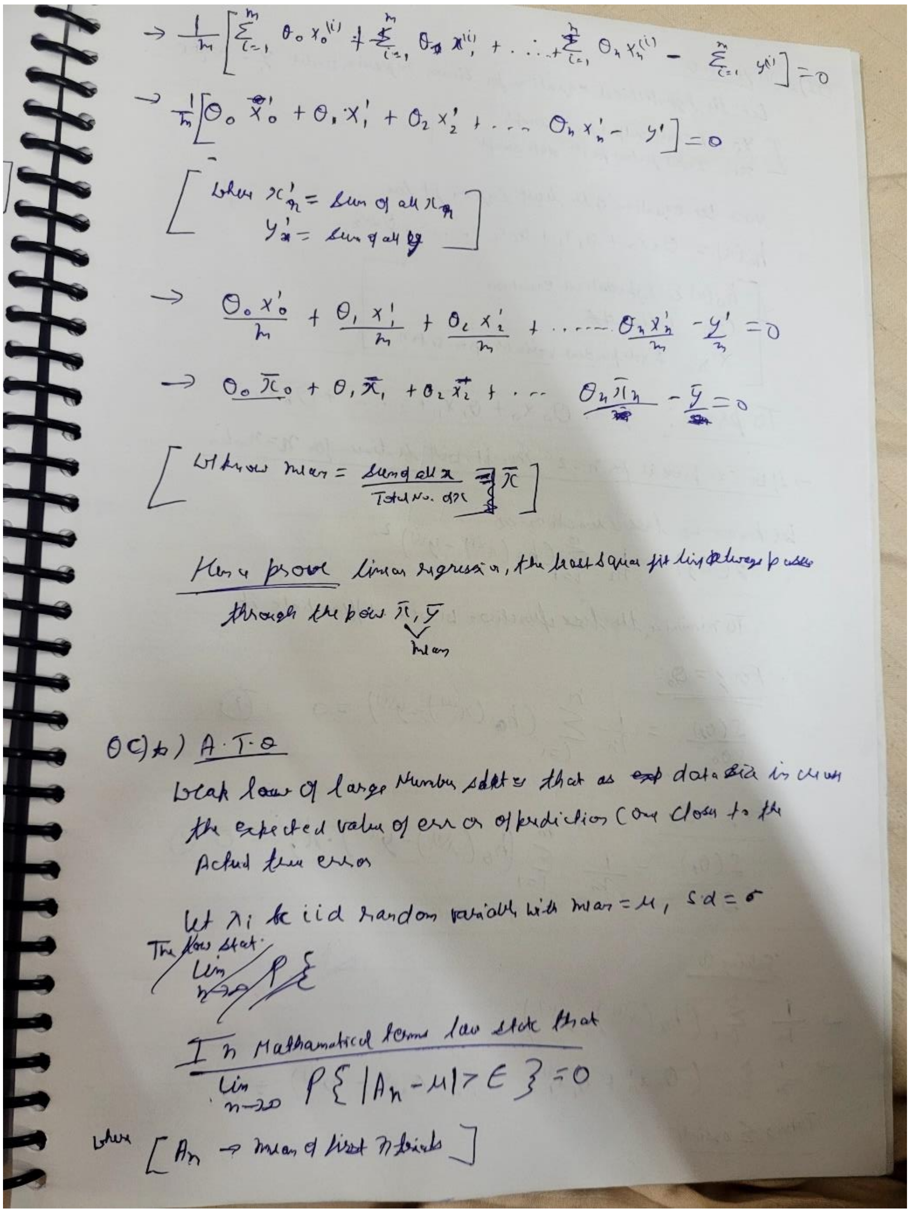
$$\frac{\delta(\theta_1)}{\delta\theta_1} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} = 0 \quad ②$$

Solving ①

$$\rightarrow \frac{1}{m} \sum_{i=i}^{m} (h_\theta(x^{(i)}) - y^{(i)}) = 0$$

$$\rightarrow \frac{1}{m} \sum_{i=i}^{m} (\theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} + \ldots \ldots \theta_j x_j^{(i)} - y^{(i)}) = 0$$

Taking Σ inside



$$\rightarrow \frac{1}{m}\left[\sum_{i=1}^{m}\theta_0 x_0^{(i)} + \sum_{i=1}^{m}\theta_1 x_1^{(i)} + \dots + \sum_{i=1}^{n}\theta_n x_n^{(i)} - \sum_{i=1}^{m} y^{(i)}\right] = 0$$

$$\rightarrow \frac{1}{m}\left[\theta_0 x'_0 + \theta_1 x'_1 + \theta_2 x'_2 + \dots \theta_n x'_n - y'\right] = 0$$

[where x'_n = sum of all x_n
y' = sum of all y]

$$\rightarrow \frac{\theta_0 x'_0}{m} + \frac{\theta_1 x'_1}{m} + \frac{\theta_2 x'_2}{m} + \dots \frac{\theta_n x'_n}{m} - \frac{y'}{m} = 0$$

$$\rightarrow \theta_0 \bar{x}_0 + \theta_1 \bar{x}_1 + \theta_2 \bar{x}_2 + \dots \theta_n \bar{x}_n - \bar{y} = 0$$

[We know mean $= \frac{\text{sum of all } x}{\text{Total No. of } x} = \bar{x}$]

Hence prove linear regression, the least square fit line always passes through the point $\underbrace{\bar{x}, \bar{y}}_{\text{mean}}$

Q C) b) A·T·Q

Weak law of large Number states that as ~~exp~~ data size is increases the expected value of error of prediction come closer to the Actual true error

Let x_i be iid random variable with mean $= \mu$, sd $= \sigma$

~~The law stat: $\lim_{n\to\infty} P\{$~~

In Mathematical terms law state that

$$\lim_{n\to\infty} P\{|A_n - \mu| > \epsilon\} = 0$$

where [A_n → mean of first n trials]

$$A_n = \frac{x_1 + x_2 \cdots\cdots x_n}{n}$$

$$E[A_n] = \frac{E[x_1 + x_2 \cdots\cdots + x_n]}{n}$$

$$E[A_n] = E\left[\frac{x_1 + x_2 \cdots\cdots x_n}{n}\right]$$

formula used :-

$$E[A_n] = \frac{n \times E[X_1]}{n} = \mu \quad ① \left[\begin{array}{l}\text{we know that ,}\\ E(x_1) = E(x_2) = \cdots E(x_n)\end{array}\right]$$

$$\text{Var}(A_n) = \text{Var}\left(\frac{x_1 + x_2 \cdots\cdots x_n}{n}\right) = \frac{1}{n^2}\left[\text{Var}(x_1) + \cdots \text{Var}(x_n)\right]$$

$$\cancel{= \frac{1}{n^2}[n}$$

$$= \frac{1}{n^2} \times n \times \sigma^2 = \frac{\sigma^2}{n}$$

~~By~~ from ~~Chaby~~ Chebyshev's ~~eq~~ Inequality

$$\rightarrow P\{|x-\mu| \geq \varepsilon\} \leq \frac{\sigma^2}{\varepsilon^2}$$

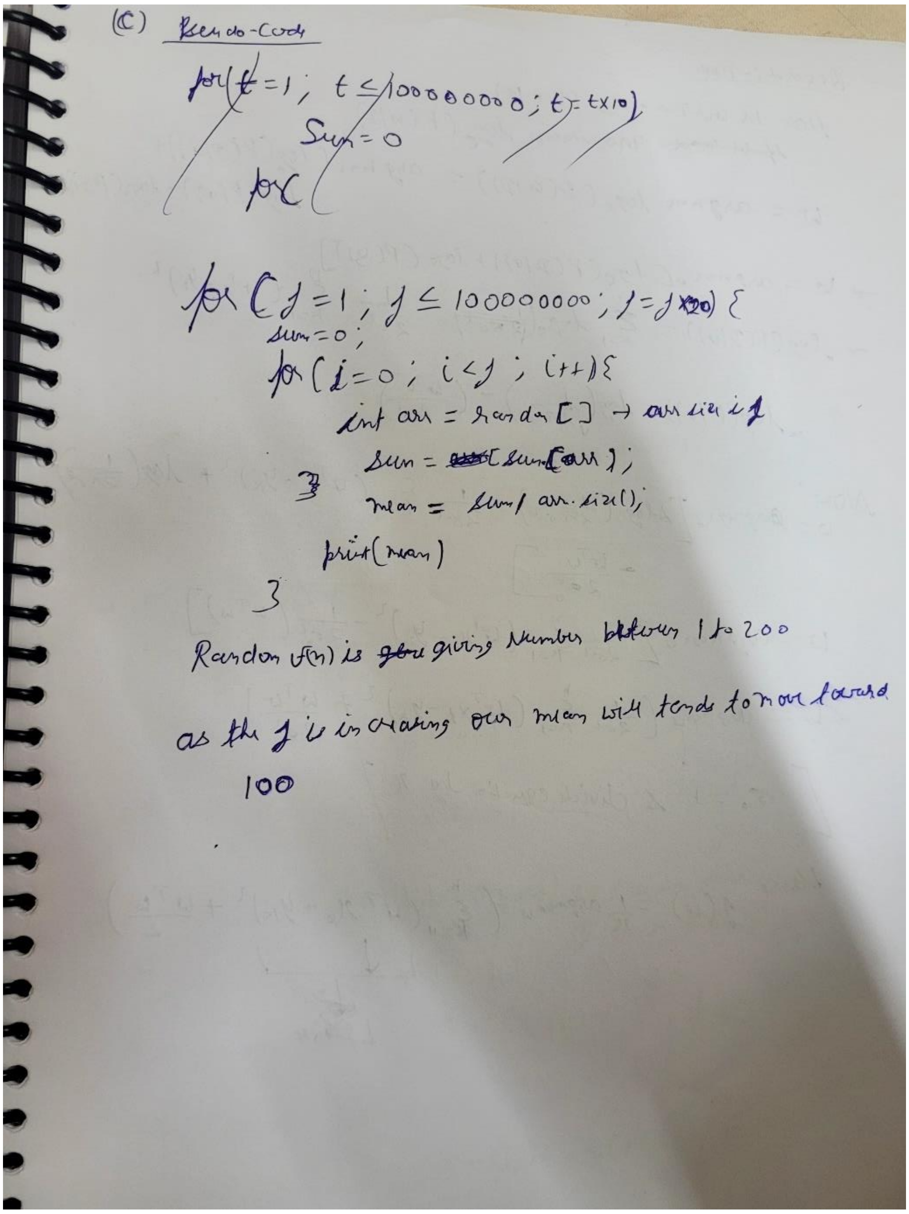
$$\rightarrow P\{|A_n-\mu| \geq \varepsilon\} \leq \frac{\sigma^2}{n\varepsilon^2}$$

$$\rightarrow \lim_{n\to\infty} P\{|A_n-\mu| \geq \varepsilon\} \leq \lim_{n\to\infty} \frac{\sigma^2}{n\varepsilon^2}$$

$\rightarrow n \longrightarrow \infty$ so value of $\frac{\sigma^2}{n\varepsilon^2}$ will tends to zero

$\rightarrow$ so we will get $\lim\limits_{n\to\infty} P\{|A_n-\mu| \geq \varepsilon\} = 0$

Hence proved



(C) Pseudo-Code

~~for(t=1; t≤100000000; t=t×10)~~
~~Sum=0~~
~~for(~~

```
for(j=1; j≤100000000; j=j×20){
  sum=0;
  for(i=0; i<j; i++){
    int arr = random[] → arr size is j
    sum = ~~[illegible]~~[sum[arr]];
  }
    mean = sum/ arr.size();
  print(mean)
}
```

Random f(n) is ~~[illegible]~~ giving Number between 1 to 200

as the j is increasing our mean will tends to move toward 100

(d) A-T-Q

The Map algorithm maximises the posterior probability

→ Bays law

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

→ Now for Map A = weights, B = data

→ $P(w|D) = \frac{1}{P(D)} \cdot P(D|w) \cdot P(w)$

→ [where $\frac{1}{P(D)}$ → Normalization, $P(D|w)$ → likelihood and $P(w)$ is prior knowledge]

→ we have a gaussian distribution of weights

$$P(w) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x-\mu}{2\sigma^2}}$$

Now $w \sim N(0, \sigma^2)$

→ $P(w) = \frac{1}{\sqrt{2\pi\sigma^2}} \times e^{-\left(\frac{w^T w}{2\sigma^2}\right)}$

→ Now ~~we can say $w \sim N(0, \sigma^2)$~~ we can say that $D|w \sim N(w^T x, \sigma^2)$

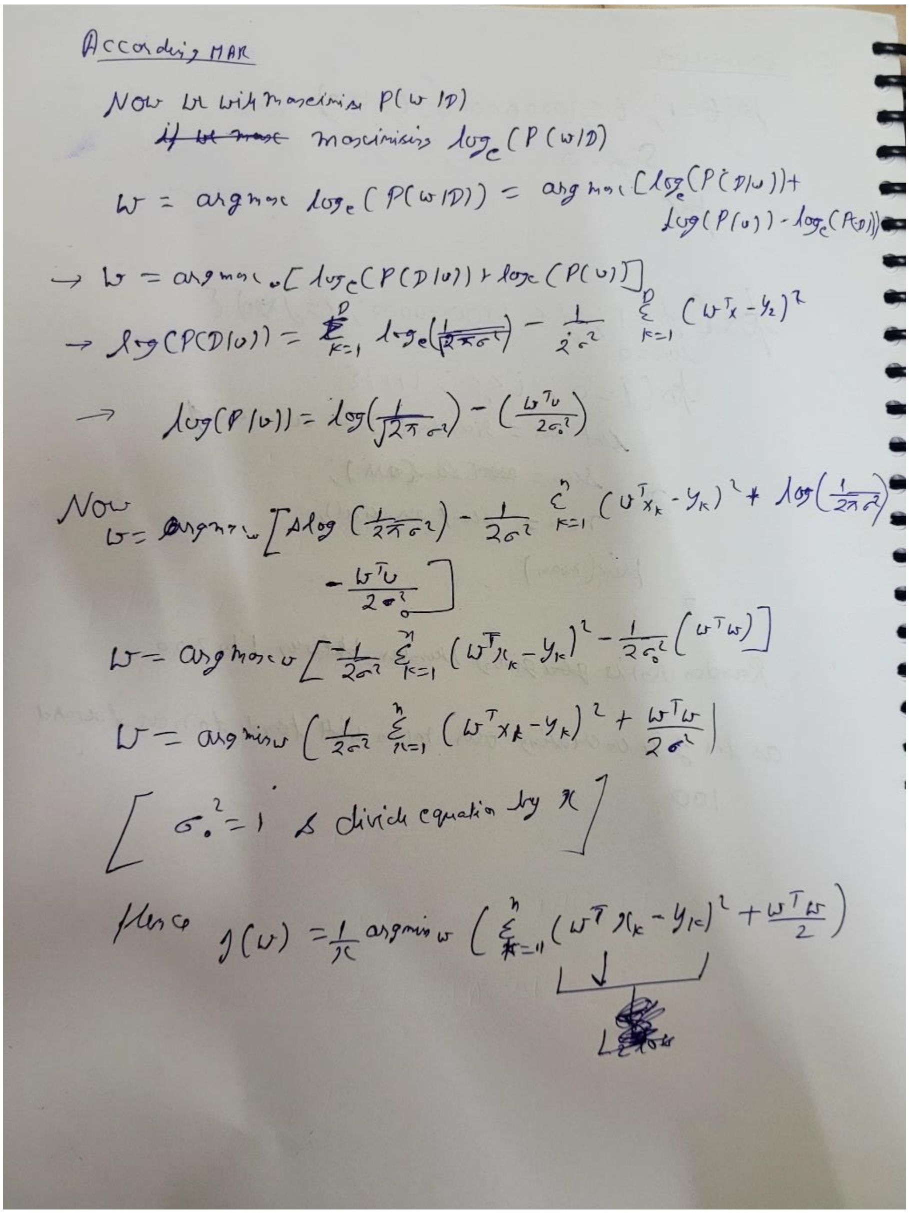
~~P(D~~

Now, $P(D|w) = \prod_{k=1}^{n} P(y_k|w) = \prod_{k=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\left(\frac{1}{2\sigma}(w^T x - y_k)^2\right)}$

Taking log ~~ton~~

$$\log_e(P(w|D)) = \log_e(P(D|w)) + \log_e(P(w)) - \log_e(P(D))$$

A~~ee~~



According MAP

Now we will maximise $P(w|D)$

~~if we must~~ maximising $\log_e(P(w|D))$

$$w = \arg\max \log_e(P(w|D)) = \arg\max[\log_e(P(D|w)) + \log(P(w)) - \log_e(P(D))]$$

$$\rightarrow w = \arg\max_w[\log_e(P(D|w)) + \log_e(P(w))]$$

$$\rightarrow \log(P(D|w)) = \sum_{k=1}^{D} \log_e\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) - \frac{1}{2\sigma^2}\sum_{k=1}^{D}(w^T x - y_k)^2$$

$$\rightarrow \log(P(w)) = \log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) - \left(\frac{w^T w}{2\sigma_0^2}\right)$$

Now

$$w = \arg\max_w\left[A\log\left(\frac{1}{2\pi\sigma^2}\right) - \frac{1}{2\sigma^2}\sum_{k=1}^{n}(w^T x_k - y_k)^2 + \log\left(\frac{1}{2\pi\sigma^2}\right) - \frac{w^T w}{2\sigma_0^2}\right]$$

$$w = \arg\max_w\left[\frac{-1}{2\sigma^2}\sum_{k=1}^{n}(w^T x_k - y_k)^2 - \frac{1}{2\sigma_0^2}(w^T w)\right]$$

$$w = \arg\min_w\left(\frac{1}{2\sigma^2}\sum_{k=1}^{n}(w^T x_k - y_k)^2 + \frac{w^T w}{2\sigma^2}\right)$$

$[\sigma_0^2 = 1$ & dividing equation by $n]$

Hence $$J(w) = \frac{1}{n}\arg\min_w\left(\underbrace{\sum_{k=1}^{n}(w^T x_k - y_k)^2}_{} + \frac{w^T w}{2}\right)$$

Answer B)

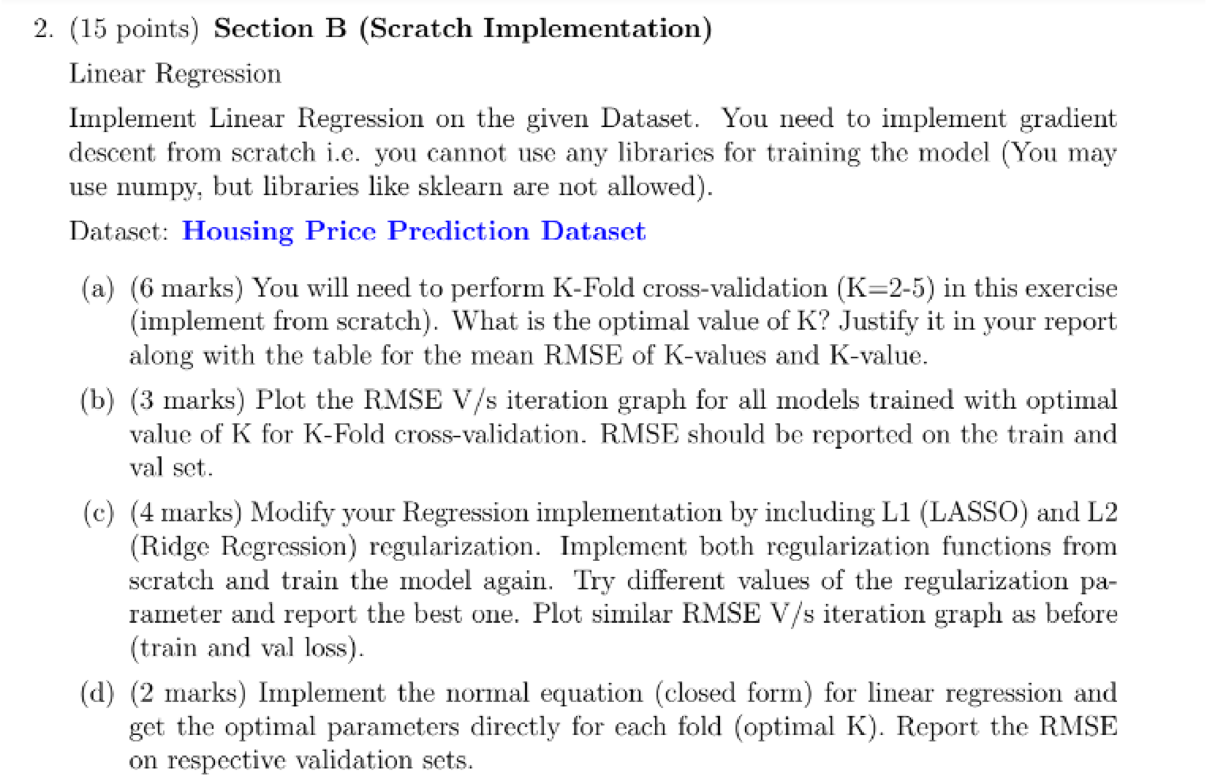
Lets take A = x , B = Y where x and y have range of real numbers

Z = A*B here we can see that Z have a strong correlation with A and B, as any one of them changes value of Z will change.

If A changes value of Z will change and similarly if B changes value of Z will change

But change in value of A doesn't change value of B so A and B have no correlation with each other same.

Section- B



2. (15 points) **Section B (Scratch Implementation)**

Linear Regression

Implement Linear Regression on the given Dataset. You need to implement gradient descent from scratch i.e. you cannot use any libraries for training the model (You may use numpy, but libraries like sklearn are not allowed).

Dataset: **Housing Price Prediction Dataset**

(a) (6 marks) You will need to perform K-Fold cross-validation (K=2-5) in this exercise (implement from scratch). What is the optimal value of K? Justify it in your report along with the table for the mean RMSE of K-values and K-value.

(b) (3 marks) Plot the RMSE V/s iteration graph for all models trained with optimal value of K for K-Fold cross-validation. RMSE should be reported on the train and val set.

(c) (4 marks) Modify your Regression implementation by including L1 (LASSO) and L2 (Ridge Regression) regularization. Implement both regularization functions from scratch and train the model again. Try different values of the regularization parameter and report the best one. Plot similar RMSE V/s iteration graph as before (train and val loss).

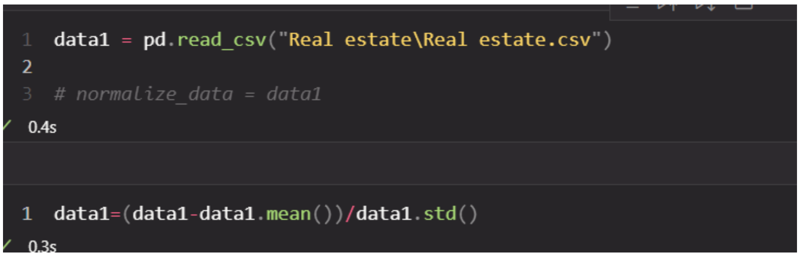
(d) (2 marks) Implement the normal equation (closed form) for linear regression and get the optimal parameters directly for each fold (optimal K). Report the RMSE on respective validation sets.

A)

DATA pre-processing:

In the given dataset we check for if there is any null value and then we scaled the entire data.

Scaling formula used here : $z = \frac{x_i - \mu}{\sigma}$



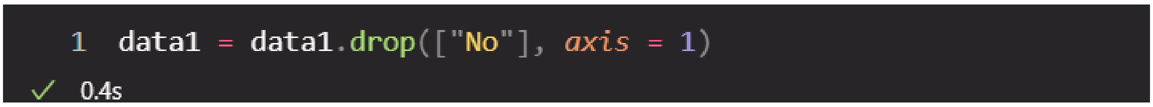
```
data1 = pd.read_csv("Real estate\Real estate.csv")

# normalize_data = data1
```

✓ 0.4s

```
data1=(data1-data1.mean())/data1.std()
```

✓ 0.3s

We have also dropped the column ‘No’ from the data.



```
data1 = data1.drop(["No"], axis = 1)
```

✓ 0.4s

Implement Gradient Descent from scratch:

All formulas:

Cost function: $J(\Theta) = \frac{1}{2m}\sum_{1}^{m}(h(x^{(i)}) - y^{(i)})^2$

$\Theta_0, \Theta_1, \ldots\ldots\ldots, \Theta_n$ - weights

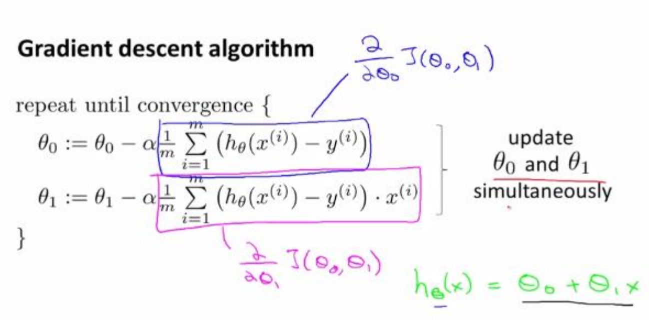
m : Total number of data samples

$x^{(i)}$: Input features for the ith data sample.

$h(x^{(i)})$: hypothesis

$y^{(i)}$: Actual output for ith data sample

Algorithm:



Gradient descent algorithm

repeat until convergence {

$$\theta_0 := \theta_0 - \alpha \boxed{\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)}$$

$$\theta_1 := \theta_1 - \alpha \boxed{\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right) \cdot x^{(i)}}$$

}

update θ_0 and θ_1 simultaneously

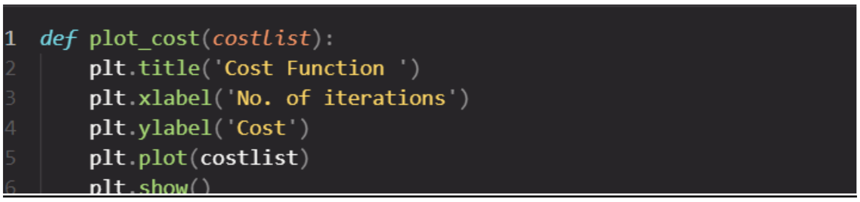
$\frac{\partial}{\partial\theta_0} J(\theta_0, \theta_1)$

$\frac{\partial}{\partial\theta_1} J(\theta_0, \theta_1)$

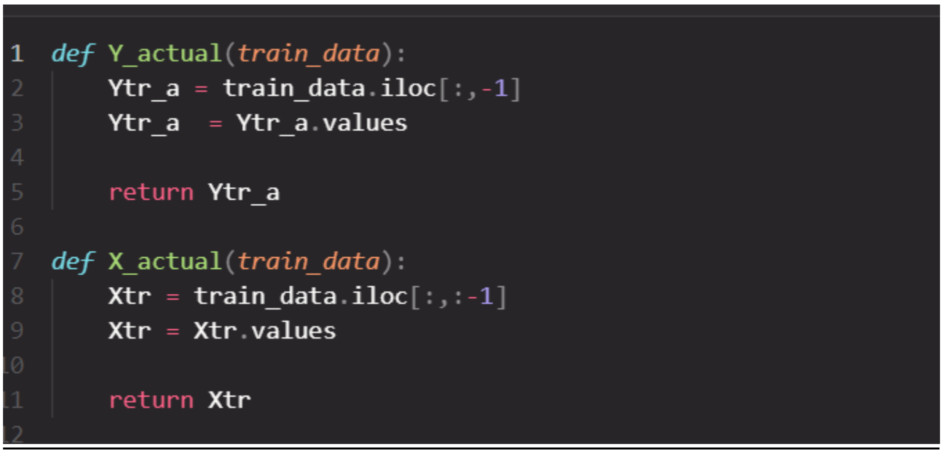
$h_\theta(x) = \theta_0 + \theta_1 x$

Code:

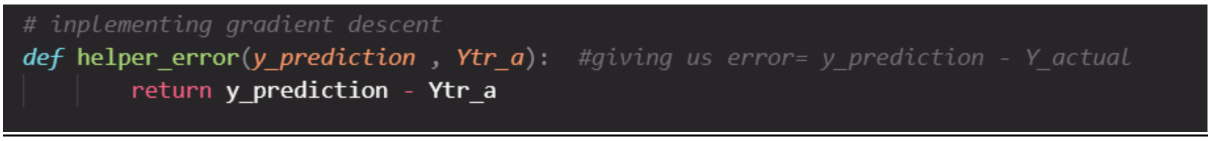
Side functions:



```python
def plot_cost(costlist):
    plt.title('Cost Function ')
    plt.xlabel('No. of iterations')
    plt.ylabel('Cost')
    plt.plot(costlist)
    plt.show()
```

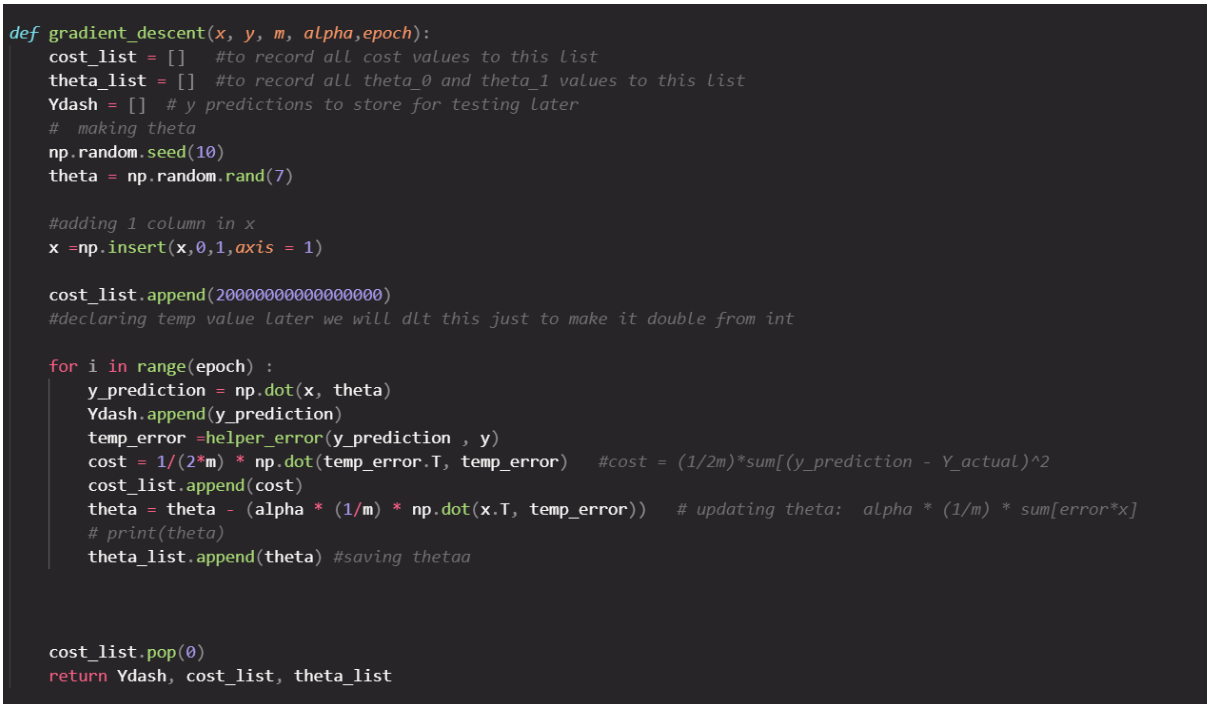


```python
def Y_actual(train_data):
    Ytr_a = train_data.iloc[:,-1]
    Ytr_a  = Ytr_a.values

    return Ytr_a

def X_actual(train_data):
    Xtr = train_data.iloc[:,:-1]
    Xtr = Xtr.values

    return Xtr
```



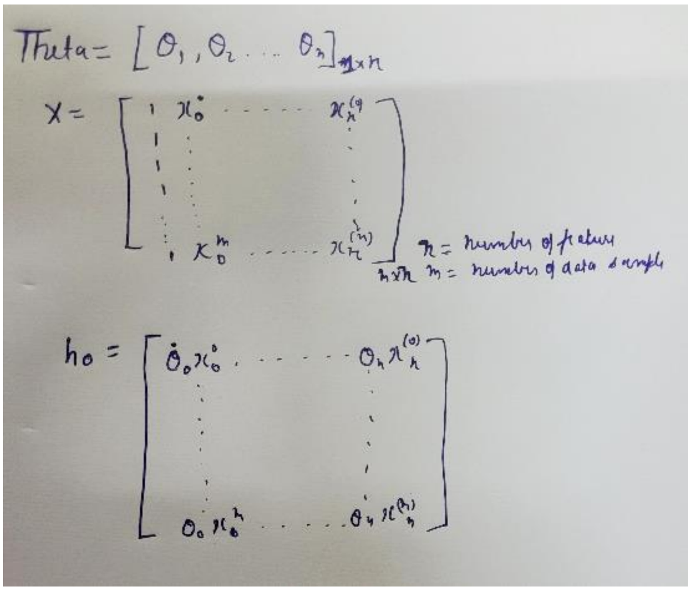
```python
# inplementing gradient descent
def helper_error(y_prediction , Ytr_a):  #giving us error= y_prediction - Y_actual
        return y_prediction - Ytr_a
```

Main function:



```python
def gradient_descent(x, y, m, alpha,epoch):
    cost_list = []   #to record all cost values to this list
    theta_list = []  #to record all theta_0 and theta_1 values to this list
    Ydash = []  # y predictions to store for testing later
    #  making theta
    np.random.seed(10)
    theta = np.random.rand(7)

    #adding 1 column in x
    x =np.insert(x,0,1,axis = 1)

    cost_list.append(20000000000000000)
    #declaring temp value later we will dlt this just to make it double from int

    for i in range(epoch) :
        y_prediction = np.dot(x, theta)
        Ydash.append(y_prediction)
        temp_error =helper_error(y_prediction , y)
        cost = 1/(2*m) * np.dot(temp_error.T, temp_error)    #cost = (1/2m)*sum[(y_prediction - Y_actual)^2
        cost_list.append(cost)
        theta = theta - (alpha * (1/m) * np.dot(x.T, temp_error))   # updating theta:  alpha * (1/m) * sum[error*x]
        # print(theta)
        theta_list.append(theta) #saving thetaa



    cost_list.pop(0)
    return Ydash, cost_list, theta_list
```

Theta : weight of model



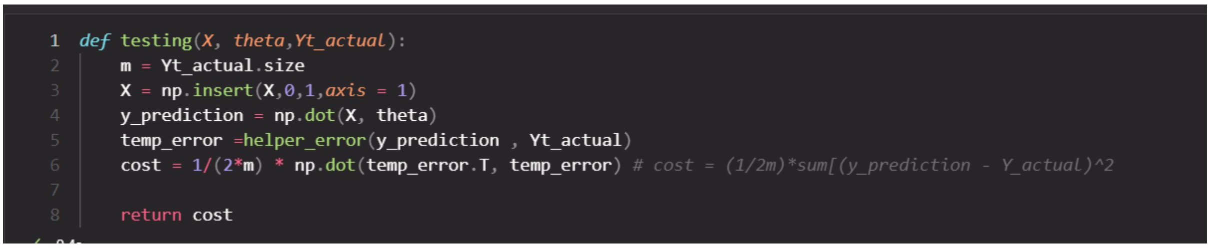
Theta= $[\theta_1, \theta_2 \ldots \theta_n]_{1 \times n}$

$$X = \begin{bmatrix} 1 & x_0^0 & \cdots & x_n^{(0)} \\ \vdots & \vdots & & \vdots \\ 1 & x_0^m & \cdots & x_n^{(m)} \end{bmatrix}_{m \times n}$$

n = number of feature

m = number of data sample

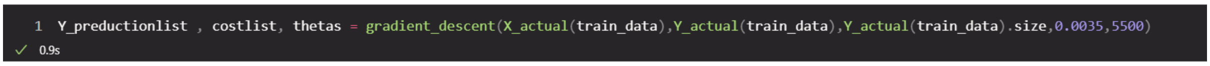
$$h_\theta = \begin{bmatrix} \theta_0 x_0^0 & \cdots & \theta_n x_n^{(0)} \\ \vdots & & \vdots \\ \theta_0 x_0^m & \cdots & \theta_n x_n^{(m)} \end{bmatrix}$$



```
def testing(X, theta,Yt_actual):
    m = Yt_actual.size
    X = np.insert(X,0,1,axis = 1)
    y_prediction = np.dot(X, theta)
    temp_error =helper_error(y_prediction , Yt_actual)
    cost = 1/(2*m) * np.dot(temp_error.T, temp_error) # cost = (1/2m)*sum[(y_prediction - Y_actual)^2

    return cost
```

We have initialized all the theta for the time randomly, then updating theta in every epoch

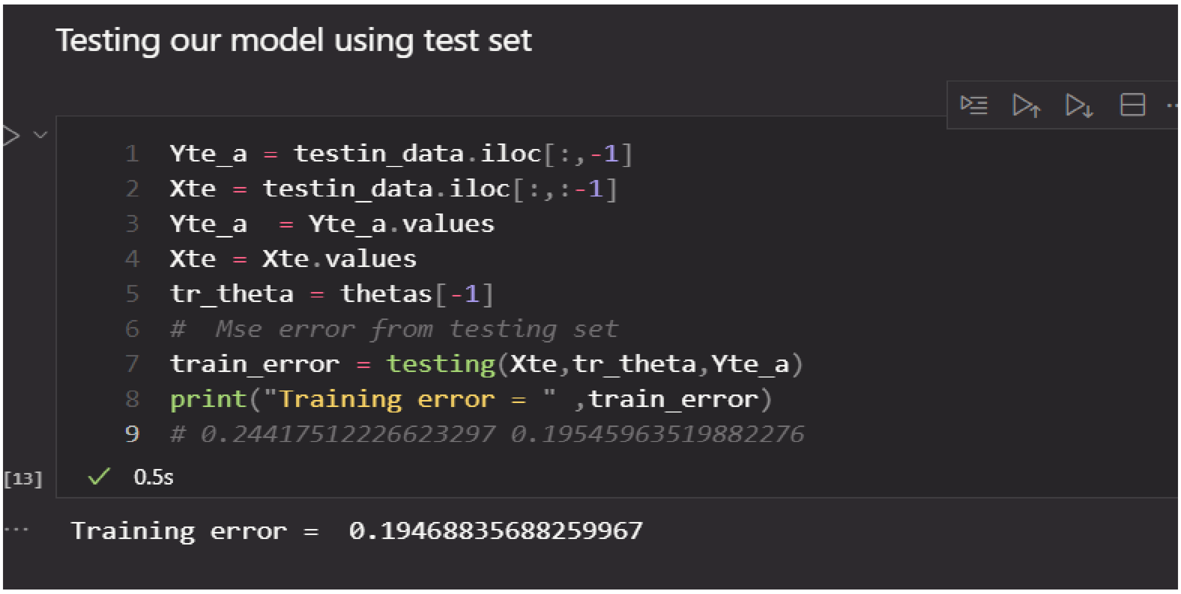
- ***We have also Calculated the testing and training error for our model***



```
Y_preductionlist , costlist, thetas = gradient_descent(X_actual(train_data),Y_actual(train_data),Y_actual(train_data).size,0.0035,5500)
```

✓ 0.9s

Alpha = 0.0035

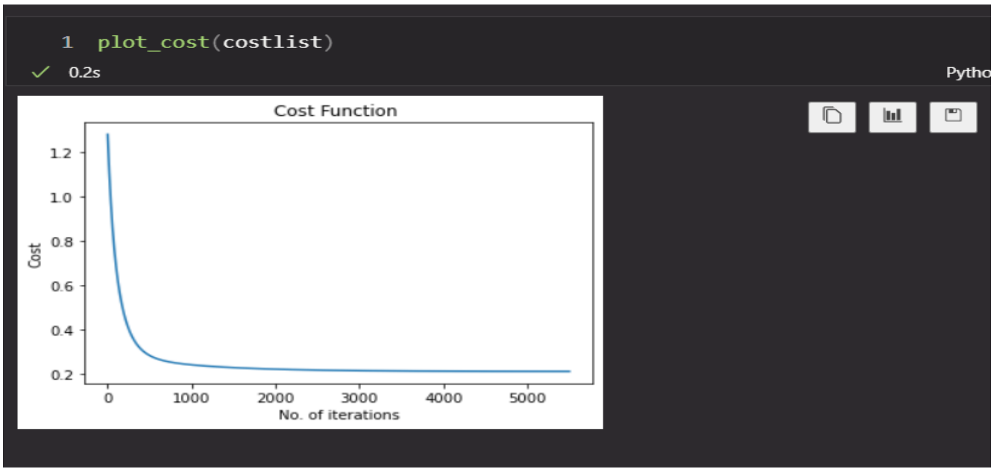


Testing our model using test set

```
Yte_a = testin_data.iloc[:,-1]
Xte = testin_data.iloc[:,:-1]
Yte_a  = Yte_a.values
Xte = Xte.values
tr_theta = thetas[-1]
#  Mse error from testing set
train_error = testing(Xte,tr_theta,Yte_a)
print("Training error = " ,train_error)
# 0.24417512226623297 0.19545963519882276
```

[13] ✓ 0.5s

```
Training error =  0.19468835688259967
```

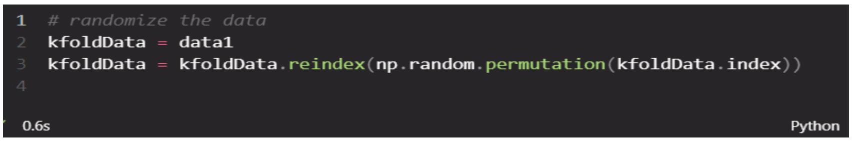


```
plot_cost(costlist)
```

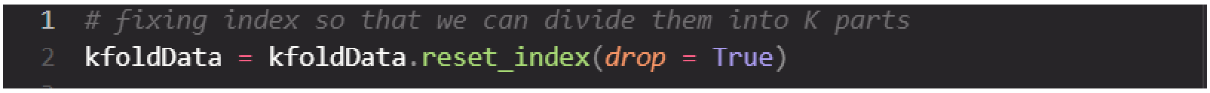
✓ 0.2s

Part A)

First we have randomised the data sample



```
# randomize the data
kfoldData = data1
kfoldData = kfoldData.reindex(np.random.permutation(kfoldData.index))

```



```
# fixing index so that we can divide them into K parts
kfoldData = kfoldData.reset_index(drop = True)
```

Then we have done K-fold cross validation according to given question:

First we divided the data into K parts (K =5,4,3,2) and for every K we have made Train($_{n}$) set and Test($_{m}$) set {where n = k-1 and m =K-n}

Then we have Trained our model with the help of gradient descent on the Train set for every n and tested the model on Test set. We have then calculated the Root mean square error(rmse) with help of train model weights.

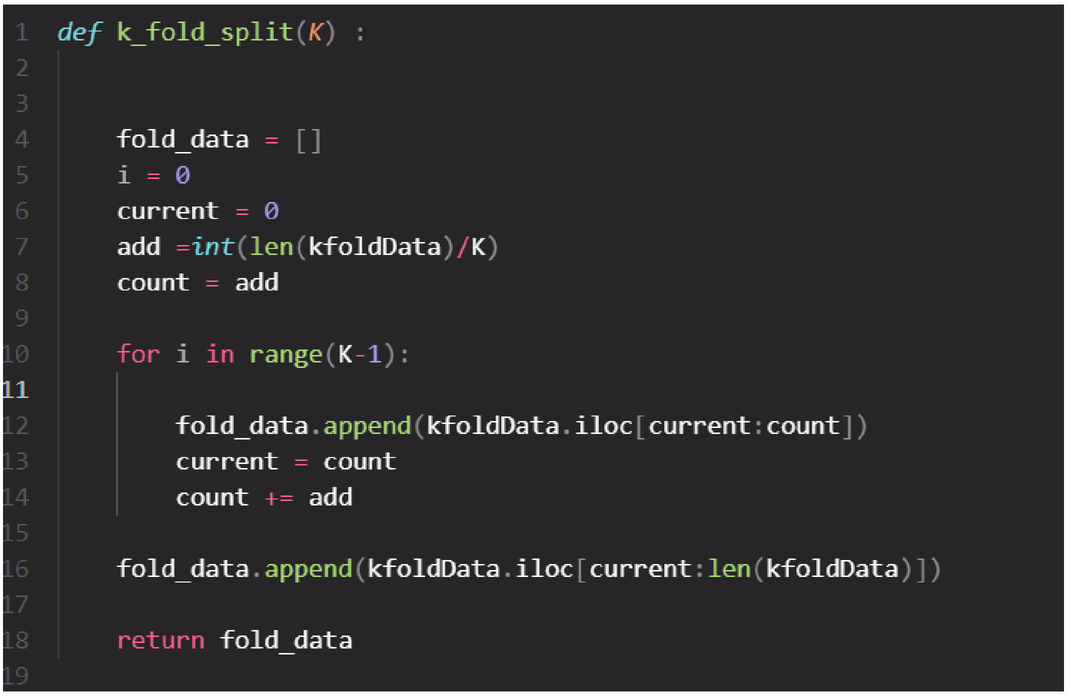
Rmse for all K-values

Table: K_value vs Mean Rmse

K-value	Mean Rmse	
5	Fold 1	0.598533
	Fold 2	0.543886
	Fold 3	0.567571
	Fold 4	0.535383
	Fold 5	0.937109
	Mean:0.636797	
4	Fold 1	0.628975
	Fold 2	0.511595
	Fold 3	0.535202
	Fold 4	0.870449
	Mean:0.636555	
3	Fold 1	0.597461
	Fold 2	0.549705
	Fold 3	0.788908
	Mean:0.6453584	
2	Fold 1	0.5915280
	Fold 2	0.7233616
	0.651257	

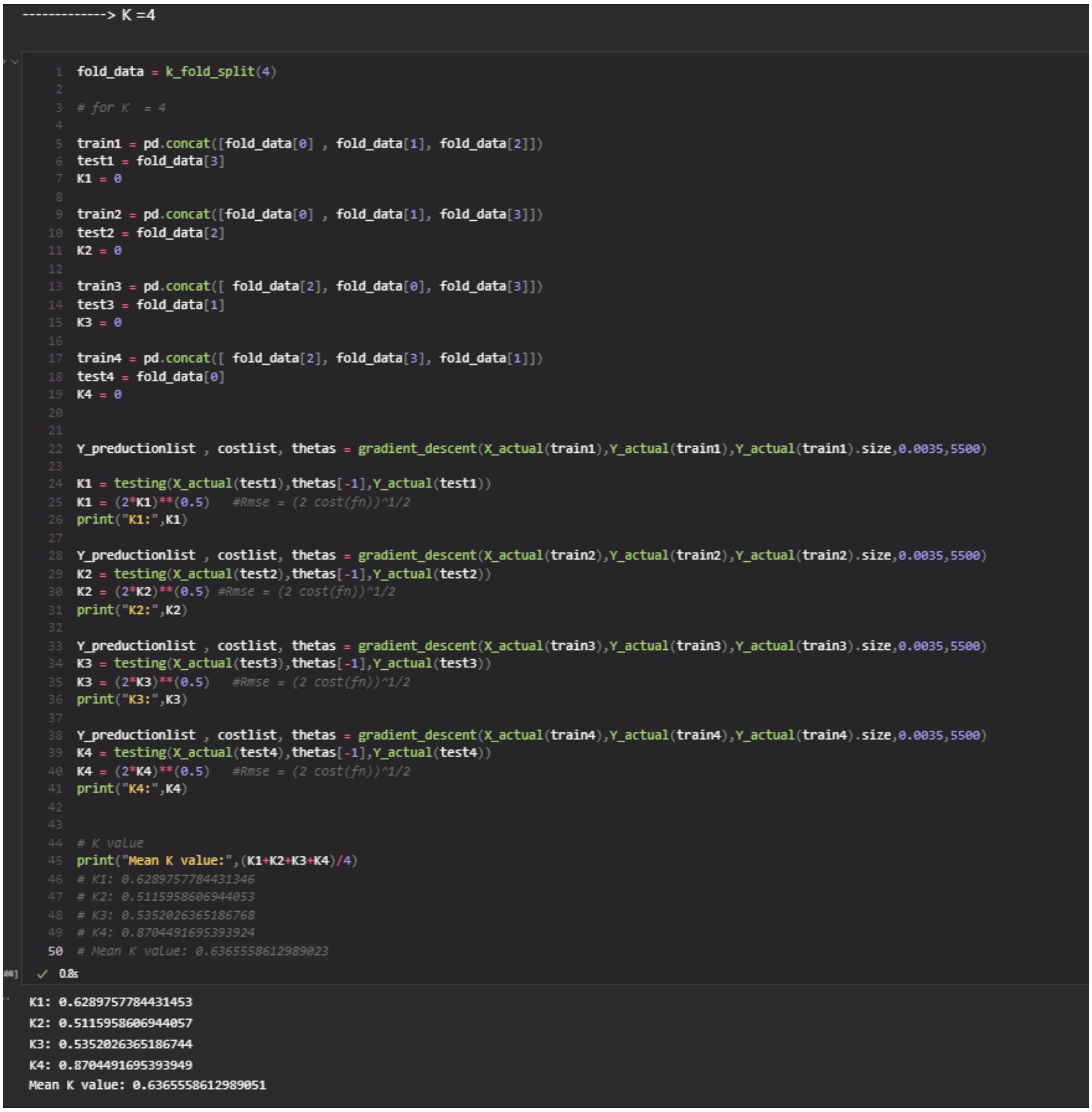
The optimal K-value is **K =4** that is **0.63655**

Code:



```python
def k_fold_split(K) :


    fold_data = []
    i = 0
    current = 0
    add =int(len(kfoldData)/K)
    count = add

    for i in range(K-1):

        fold_data.append(kfoldData.iloc[current:count])
        current = count
        count += add

    fold_data.append(kfoldData.iloc[current:len(kfoldData)])

    return fold_data

```

Showing output code for Best Kmean:



-------------> K =4

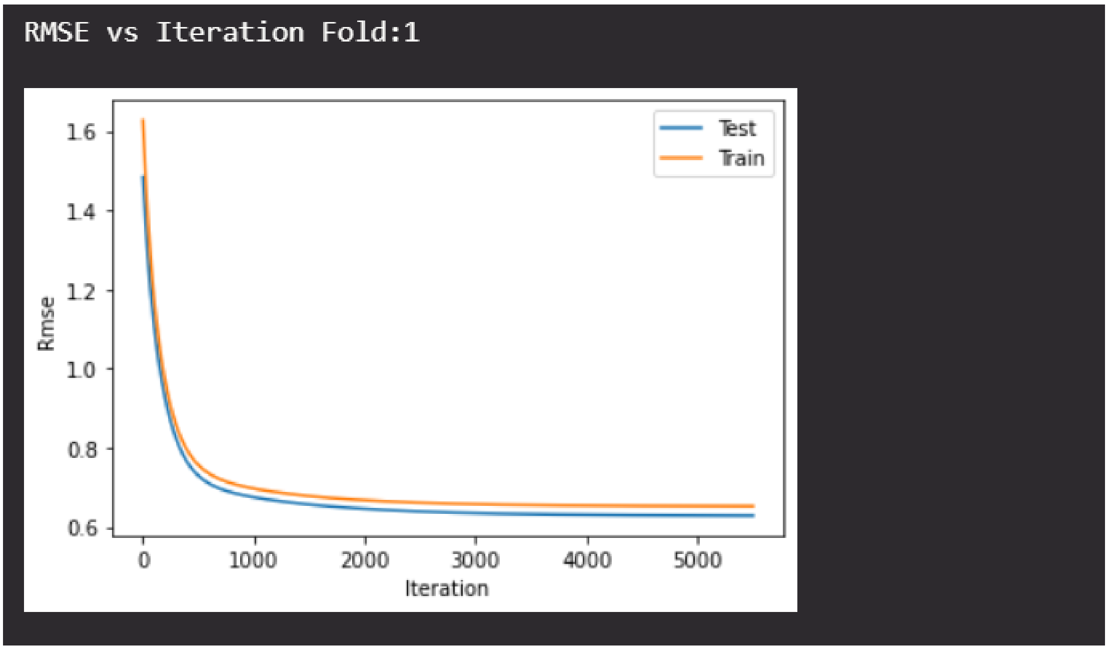
```
fold_data = k_fold_split(4)

# for K  = 4

train1 = pd.concat([fold_data[0] , fold_data[1], fold_data[2]])
test1 = fold_data[3]
K1 = 0

train2 = pd.concat([fold_data[0] , fold_data[1], fold_data[3]])
test2 = fold_data[2]
K2 = 0

train3 = pd.concat([ fold_data[2], fold_data[0], fold_data[3]])
test3 = fold_data[1]
K3 = 0

train4 = pd.concat([ fold_data[2], fold_data[3], fold_data[1]])
test4 = fold_data[0]
K4 = 0


Y_preductionlist , costlist, thetas = gradient_descent(X_actual(train1),Y_actual(train1),Y_actual(train1).size,0.0035,5500)

K1 = testing(X_actual(test1),thetas[-1],Y_actual(test1))
K1 = (2*K1)**(0.5)   #Rmse = (2 cost(fn))^1/2
print("K1:",K1)

Y_preductionlist , costlist, thetas = gradient_descent(X_actual(train2),Y_actual(train2),Y_actual(train2).size,0.0035,5500)
K2 = testing(X_actual(test2),thetas[-1],Y_actual(test2))
K2 = (2*K2)**(0.5) #Rmse = (2 cost(fn))^1/2
print("K2:",K2)

Y_preductionlist , costlist, thetas = gradient_descent(X_actual(train3),Y_actual(train3),Y_actual(train3).size,0.0035,5500)
K3 = testing(X_actual(test3),thetas[-1],Y_actual(test3))
K3 = (2*K3)**(0.5)   #Rmse = (2 cost(fn))^1/2
print("K3:",K3)

Y_preductionlist , costlist, thetas = gradient_descent(X_actual(train4),Y_actual(train4),Y_actual(train4).size,0.0035,5500)
K4 = testing(X_actual(test4),thetas[-1],Y_actual(test4))
K4 = (2*K4)**(0.5)   #Rmse = (2 cost(fn))^1/2
print("K4:",K4)


# K value
print("Mean K value:",(K1+K2+K3+K4)/4)
# K1: 0.6289757784431346
# K2: 0.5115958606944053
# K3: 0.5352026365186768
# K4: 0.8704491695393924
# Mean K value: 0.6365558612989023
```

✓ 0.8s

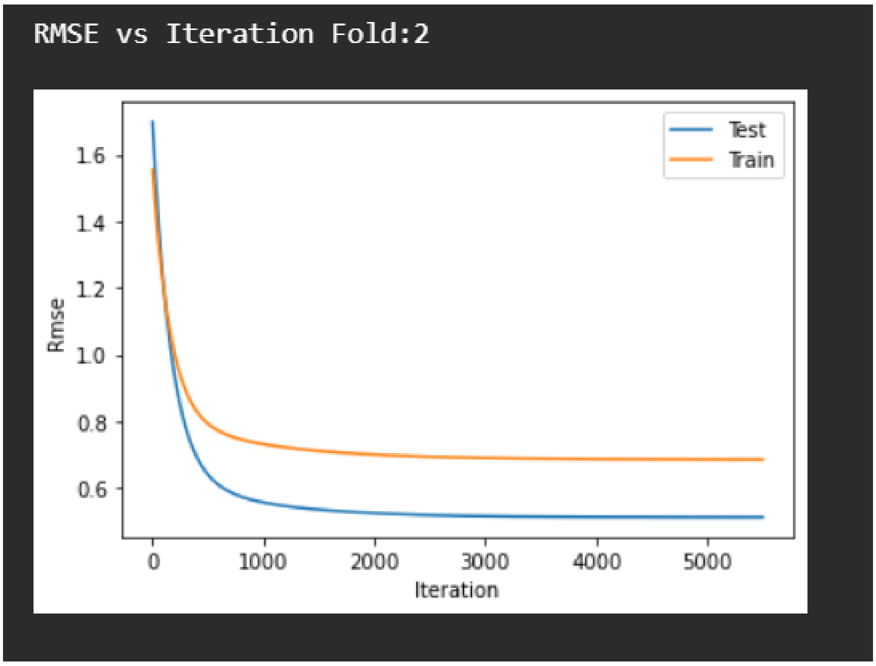
```
K1: 0.6289757784431453
K2: 0.5115958606944057
K3: 0.5352026365186744
K4: 0.8704491695393949
Mean K value: 0.6365558612989051
```

PART B

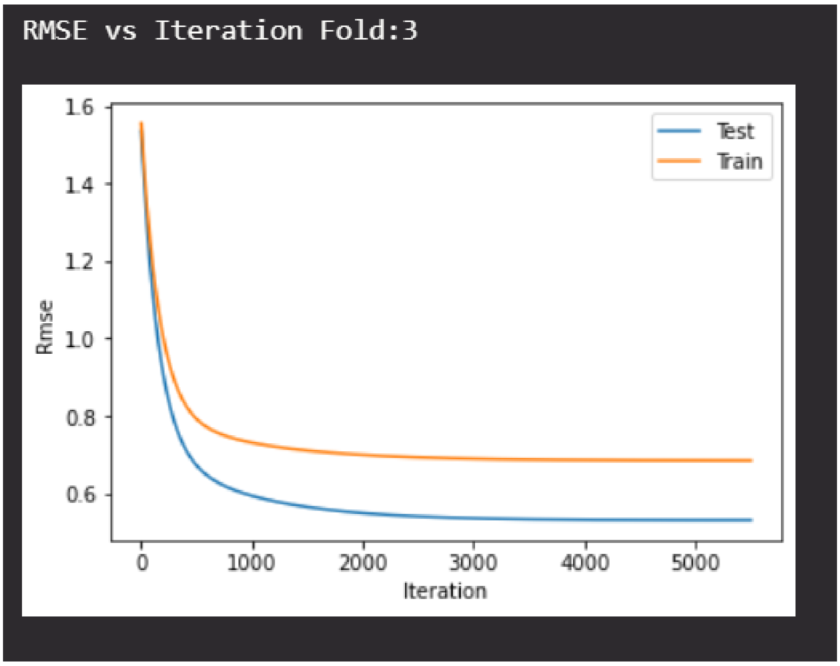
RMSE vs Iteration graph for all models trained with optimal value of K=4 for K-fold



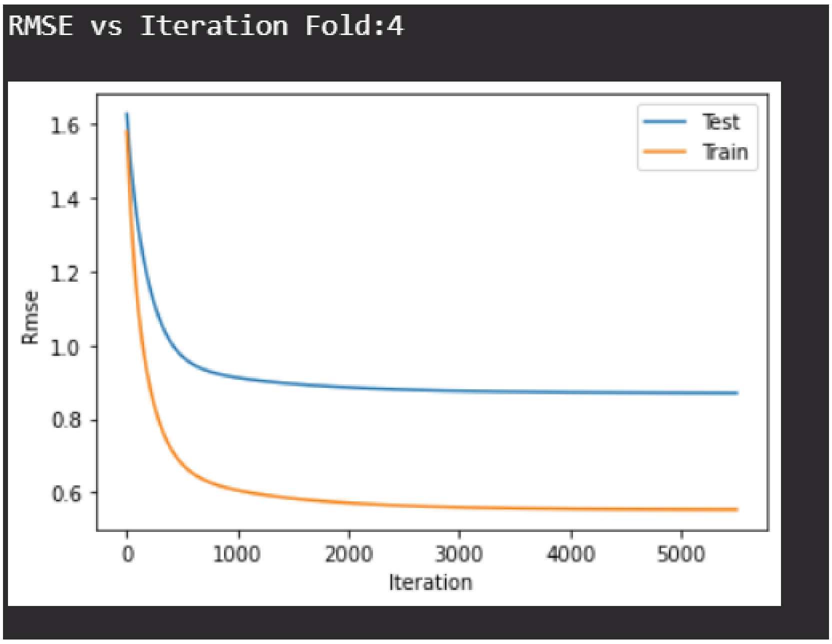
RMSE vs Iteration Fold:1



RMSE vs Iteration Fold:2



RMSE vs Iteration Fold:3

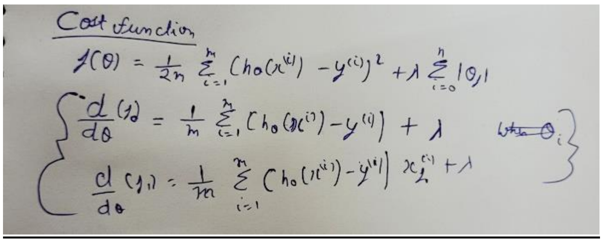


RMSE vs Iteration Fold:4

PART :C

L1 regularization:

Formula used:

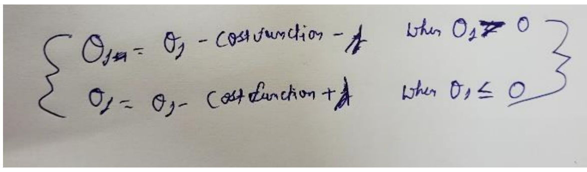


Cost function

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{i=0}^{n}|\theta_j|$$

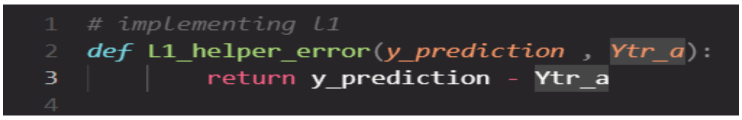
$$\left\{\begin{array}{ll} \frac{d}{d\theta}(J_0) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) + \lambda & \text{~~when~~ } \theta_i \\ \frac{d}{d\theta}(J_1) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})\, x_1^{(i)} + \lambda & \end{array}\right\}$$

Updating Θj (way to update the theta inside code for L1)

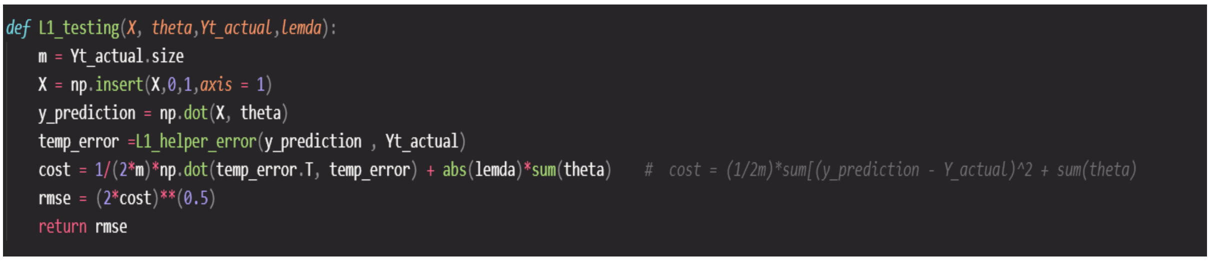


$$\left\{\begin{array}{ll} \theta_j = \theta_j - \text{costfunction} - \lambda & \text{when } \theta_j \geq 0 \\ \theta_j = \theta_j - \text{costfunction} + \lambda & \text{when } \theta_j \leq 0 \end{array}\right\}$$

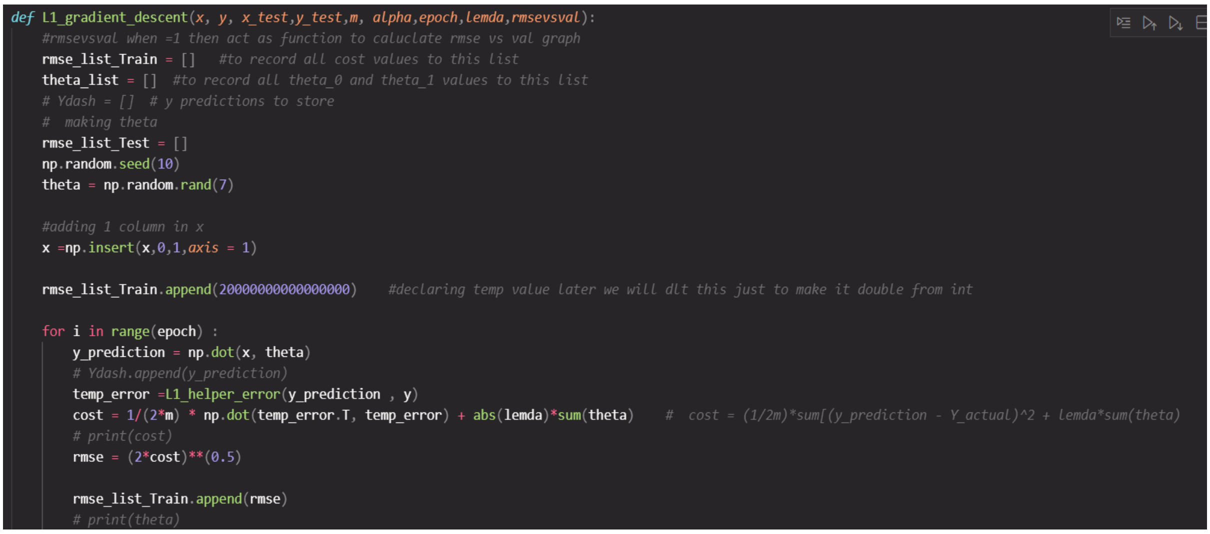
Code



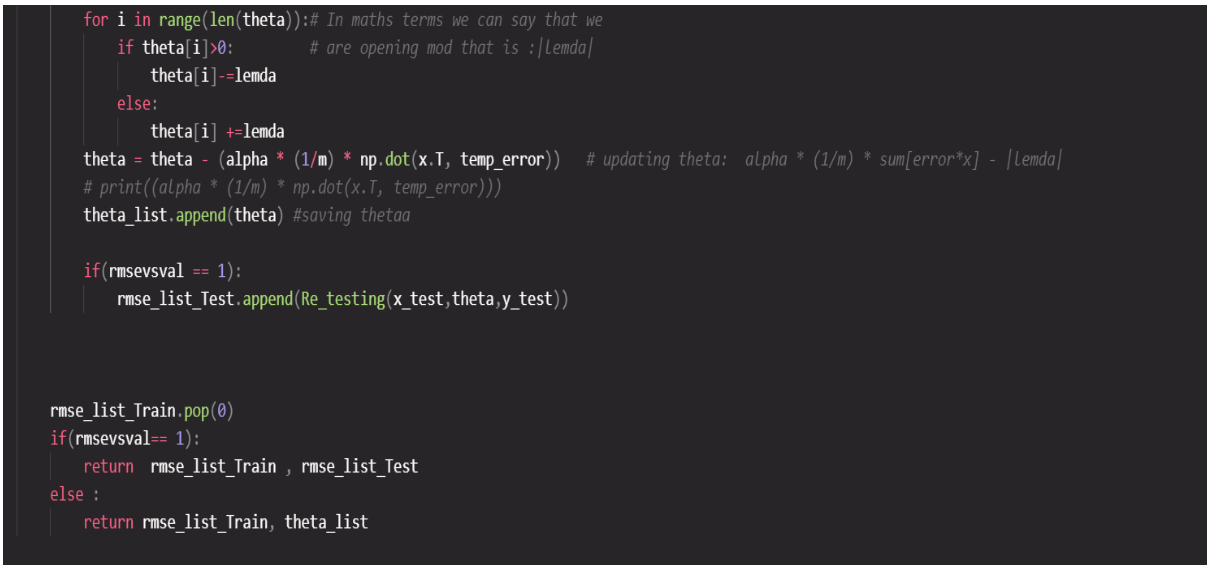
```
# implementing L1
def L1_helper_error(y_prediction , Ytr_a):
        return y_prediction - Ytr_a

```



```
def L1_testing(X, theta,Yt_actual,lemda):
    m = Yt_actual.size
    X = np.insert(X,0,1,axis = 1)
    y_prediction = np.dot(X, theta)
    temp_error =L1_helper_error(y_prediction , Yt_actual)
    cost = 1/(2*m)*np.dot(temp_error.T, temp_error) + abs(lemda)*sum(theta)    #  cost = (1/2m)*sum[(y_prediction - Y_actual)^2 + sum(theta)
    rmse = (2*cost)**(0.5)
    return rmse
```



```
def L1_gradient_descent(x, y, x_test,y_test,m, alpha,epoch,lemda,rmsevsval):
    #rmsevsval when =1 then act as function to caluclate rmse vs val graph
    rmse_list_Train = []   #to record all cost values to this list
    theta_list = []  #to record all theta_0 and theta_1 values to this list
    # Ydash = []  # y predictions to store
    #  making theta
    rmse_list_Test = []
    np.random.seed(10)
    theta = np.random.rand(7)

    #adding 1 column in x
    x =np.insert(x,0,1,axis = 1)

    rmse_list_Train.append(20000000000000000)    #declaring temp value later we will dlt this just to make it double from int

    for i in range(epoch) :
        y_prediction = np.dot(x, theta)
        # Ydash.append(y_prediction)
        temp_error =L1_helper_error(y_prediction , y)
        cost = 1/(2*m) * np.dot(temp_error.T, temp_error) + abs(lemda)*sum(theta)    #  cost = (1/2m)*sum[(y_prediction - Y_actual)^2 + lemda*sum(theta)
        # print(cost)
        rmse = (2*cost)**(0.5)

        rmse_list_Train.append(rmse)
        # print(theta)
```



```
        for i in range(len(theta)):# In maths terms we can say that we
            if theta[i]>0:         # are opening mod that is :|lemda|
                theta[i]-=lemda
            else:
                theta[i] +=lemda
        theta = theta - (alpha * (1/m) * np.dot(x.T, temp_error))   # updating theta:  alpha * (1/m) * sum[error*x] - |lemda|
        # print((alpha * (1/m) * np.dot(x.T, temp_error)))
        theta_list.append(theta) #saving thetaa

        if(rmsevsval == 1):
            rmse_list_Test.append(Re_testing(x_test,theta,y_test))



    rmse_list_Train.pop(0)
    if(rmsevsval== 1):
        return  rmse_list_Train , rmse_list_Test
    else :
        return rmse_list_Train, theta_list
```

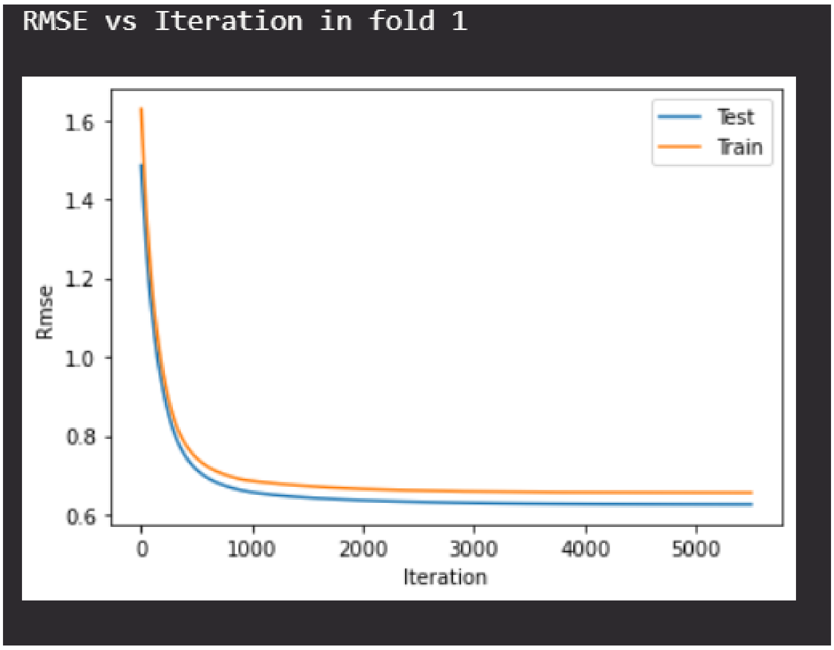
Different values of lambda vs Mean RMSE:

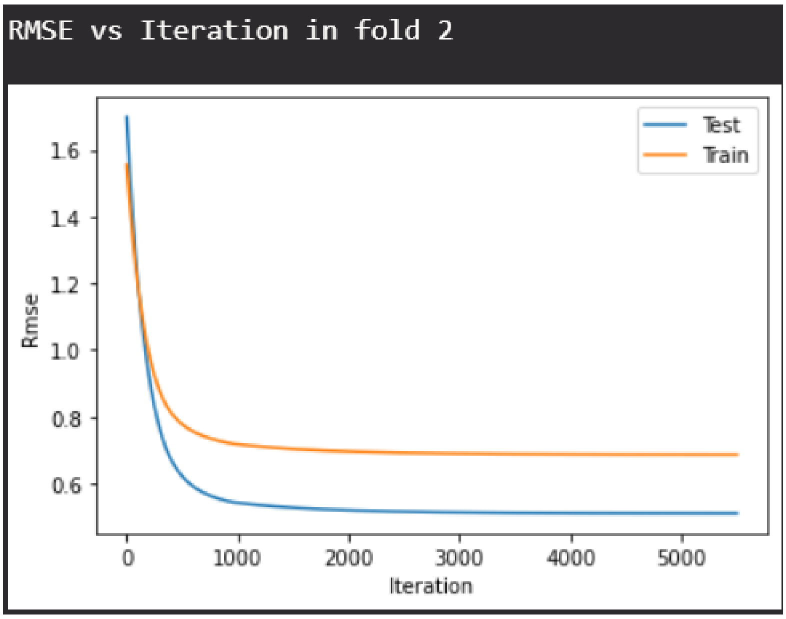
Table:

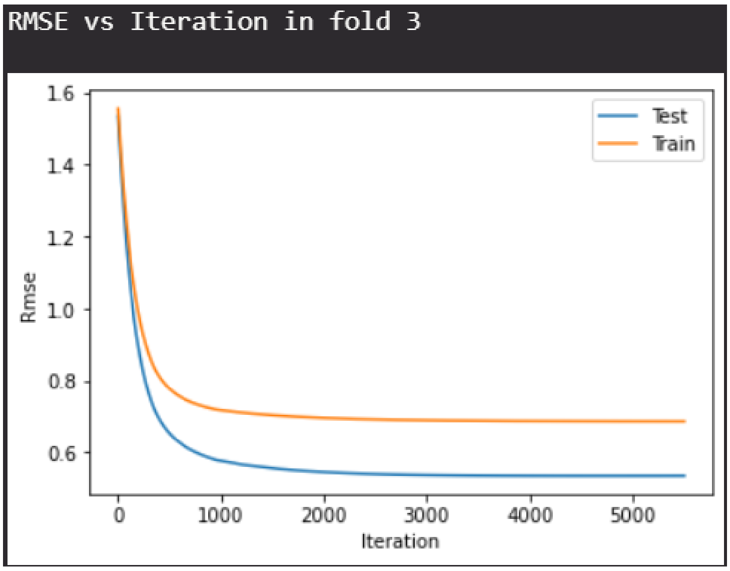
Lambda	Mean Rmse
0.1	0.9516836
0.001	0.7142055
0.0001	0.625595
0.000051	0.6261506
0.0000059	0.6285677
0.00000069	0.6289269

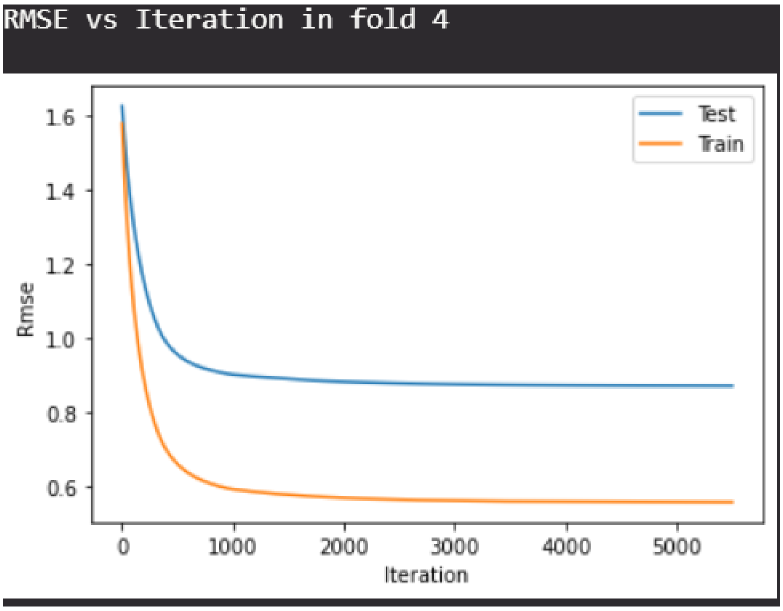
K-Fold Cross Validation Method with K = 4 and Lambda = 0.0001

Graph :

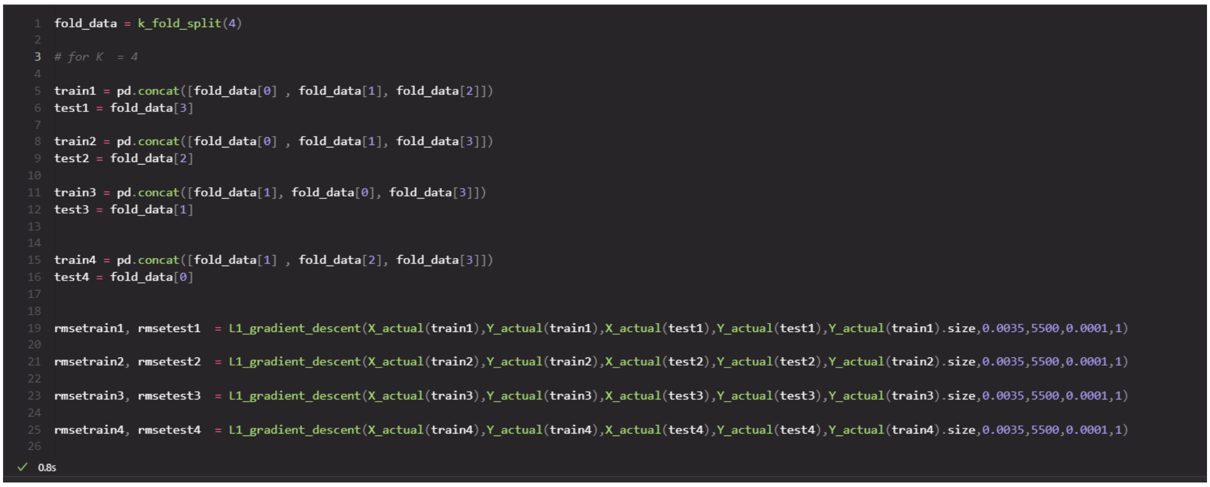



RMSE vs Iteration in fold 2

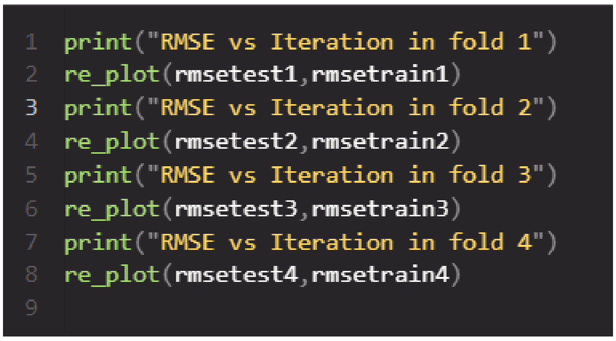

RMSE vs Iteration in fold 3


RMSE vs Iteration in fold 4

Code for K = 4 and lambda = 0.0001



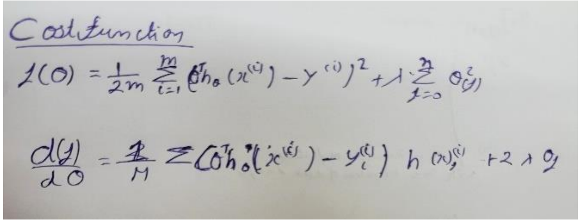
```
fold_data = k_fold_split(4)

# for K  = 4

train1 = pd.concat([fold_data[0] , fold_data[1], fold_data[2]])
test1 = fold_data[3]

train2 = pd.concat([fold_data[0] , fold_data[1], fold_data[3]])
test2 = fold_data[2]

train3 = pd.concat([fold_data[1], fold_data[0], fold_data[3]])
test3 = fold_data[1]


train4 = pd.concat([fold_data[1] , fold_data[2], fold_data[3]])
test4 = fold_data[0]


rmsetrain1, rmsetest1  = L1_gradient_descent(X_actual(train1),Y_actual(train1),X_actual(test1),Y_actual(test1),Y_actual(train1).size,0.0035,5500,0.0001,1)

rmsetrain2, rmsetest2  = L1_gradient_descent(X_actual(train2),Y_actual(train2),X_actual(test2),Y_actual(test2),Y_actual(train2).size,0.0035,5500,0.0001,1)

rmsetrain3, rmsetest3  = L1_gradient_descent(X_actual(train3),Y_actual(train3),X_actual(test3),Y_actual(test3),Y_actual(train3).size,0.0035,5500,0.0001,1)

rmsetrain4, rmsetest4  = L1_gradient_descent(X_actual(train4),Y_actual(train4),X_actual(test4),Y_actual(test4),Y_actual(train4).size,0.0035,5500,0.0001,1)

✓ 0.8s
```



```
print("RMSE vs Iteration in fold 1")
re_plot(rmsetest1,rmsetrain1)
print("RMSE vs Iteration in fold 2")
re_plot(rmsetest2,rmsetrain2)
print("RMSE vs Iteration in fold 3")
re_plot(rmsetest3,rmsetrain3)
print("RMSE vs Iteration in fold 4")
re_plot(rmsetest4,rmsetrain4)
```

L2 regulaeization(Ridge regression)

Formulas used:

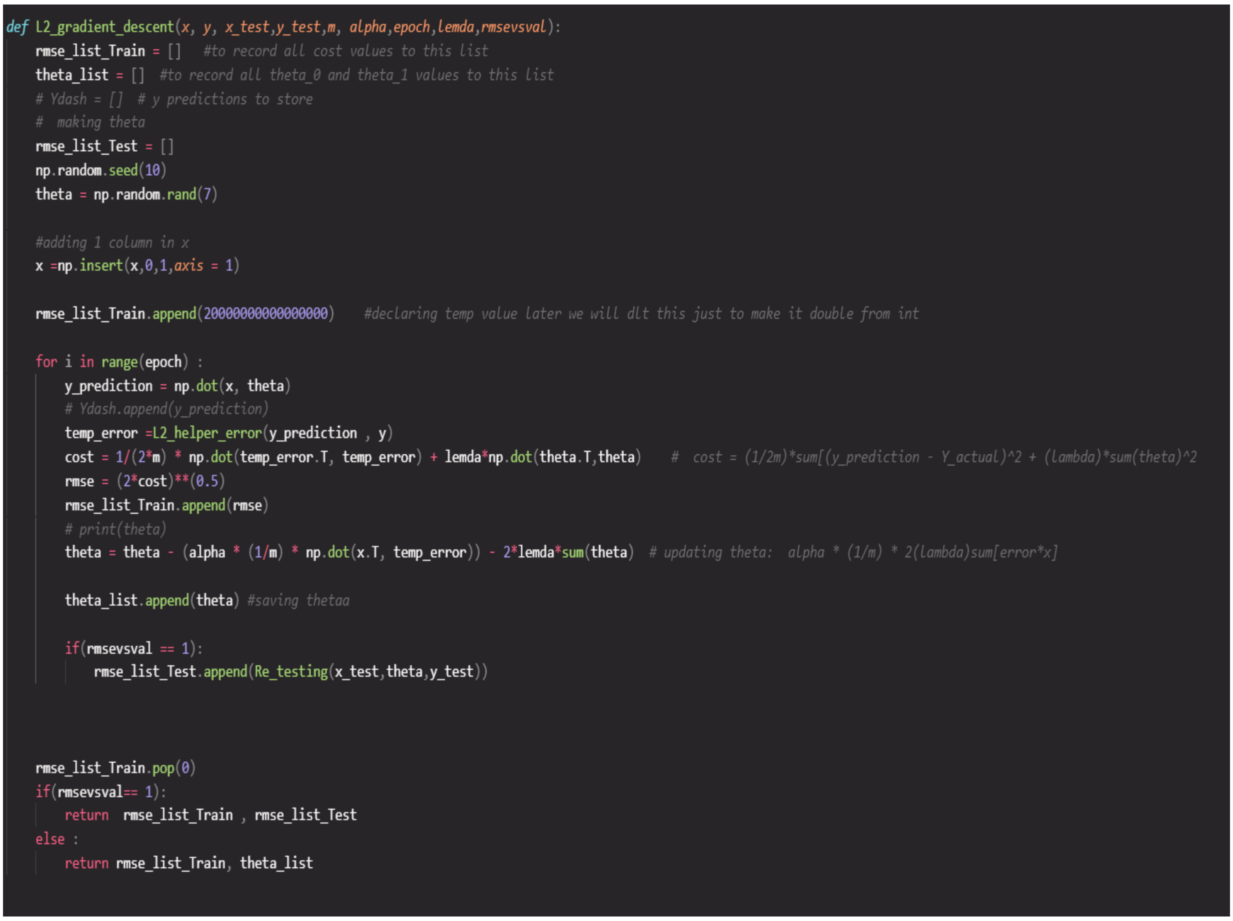


Cost function

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (\theta^T h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=0}^{n} \theta_{(j)}^2$$

$$\frac{d(J)}{d\theta} = \frac{1}{M} \sum (\theta^T h_\theta(x^{(i)}) - y_i^{(i)}) \, h(x)_j^{(i)} + 2\lambda\theta_j$$

Code:



```
def L2_gradient_descent(x, y, x_test,y_test,m, alpha,epoch,lemda,rmsevsval):
    rmse_list_Train = []   #to record all cost values to this list
    theta_list = []  #to record all theta_0 and theta_1 values to this list
    # Ydash = []  # y predictions to store
    #  making theta
    rmse_list_Test = []
    np.random.seed(10)
    theta = np.random.rand(7)

    #adding 1 column in x
    x =np.insert(x,0,1,axis = 1)

    rmse_list_Train.append(20000000000000000)    #declaring temp value later we will dlt this just to make it double from int

    for i in range(epoch) :
        y_prediction = np.dot(x, theta)
        # Ydash.append(y_prediction)
        temp_error =L2_helper_error(y_prediction , y)
        cost = 1/(2*m) * np.dot(temp_error.T, temp_error) + lemda*np.dot(theta.T,theta)    #  cost = (1/2m)*sum[(y_prediction - Y_actual)^2 + (lambda)*sum(theta)^2
        rmse = (2*cost)**(0.5)
        rmse_list_Train.append(rmse)
        # print(theta)
        theta = theta - (alpha * (1/m) * np.dot(x.T, temp_error)) - 2*lemda*sum(theta)  # updating theta:  alpha * (1/m) * 2(lambda)sum[error*x]

        theta_list.append(theta) #saving thetaa

        if(rmsevsval == 1):
            rmse_list_Test.append(Re_testing(x_test,theta,y_test))



    rmse_list_Train.pop(0)
    if(rmsevsval== 1):
        return  rmse_list_Train , rmse_list_Test
    else :
        return rmse_list_Train, theta_list
```

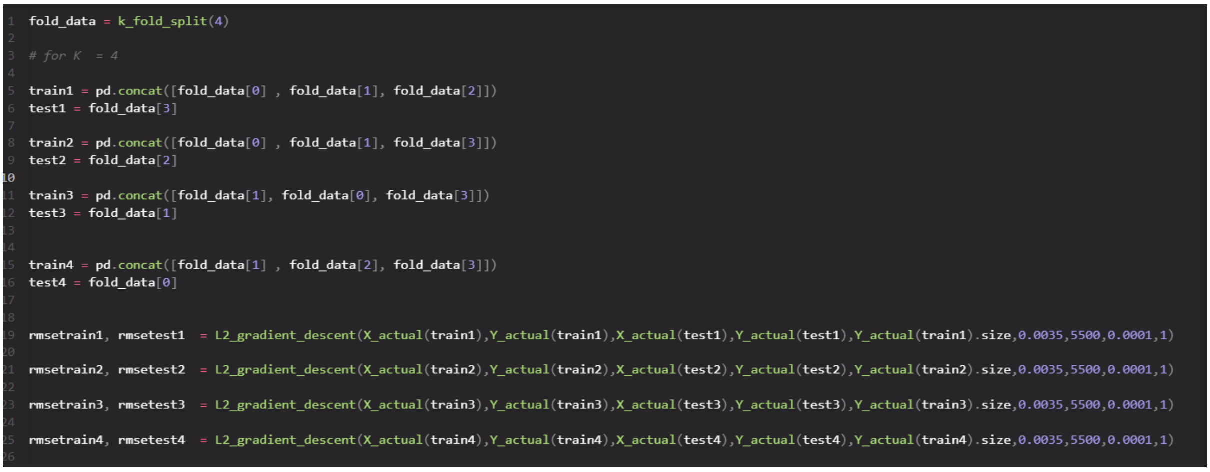
Different values of lambda vs Mean RMSE:

Lambda	**Mean Rmse**
0.1	0.8888019
0.001	0.7704632
0.0001	0.6275338
0.000051	0.62808050
0.0000059	0.62885283
0.00000069	0.62885283

Best value of lamda for L2(**Ridge regression**) is 0.0001

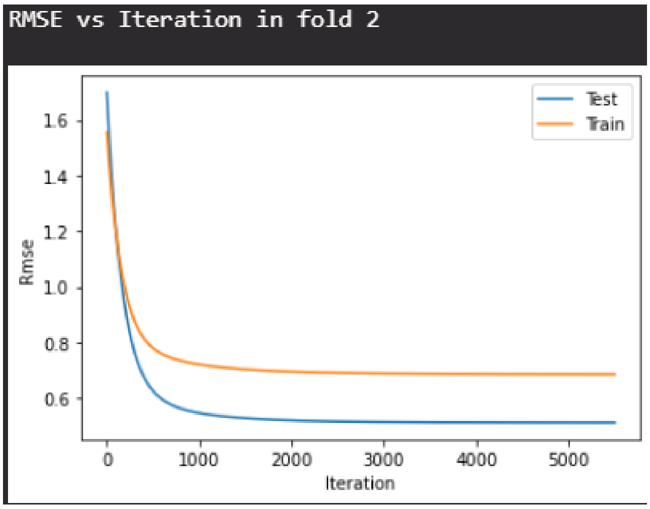
L2 plot Rmse V/s iteration for K =4

Code:

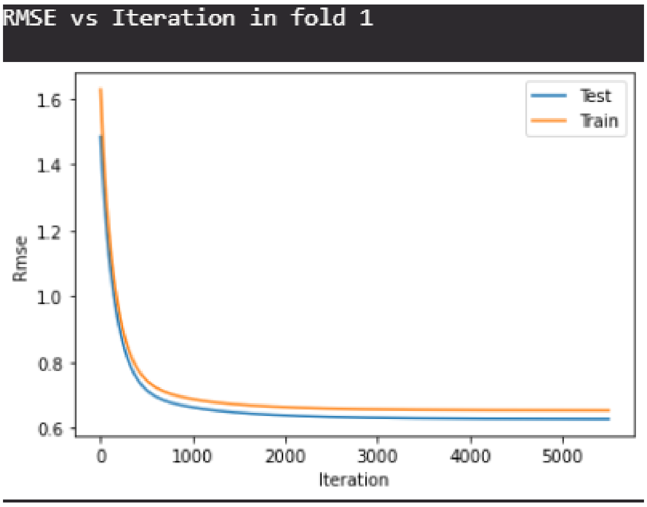


```
fold_data = k_fold_split(4)

# for K  = 4

train1 = pd.concat([fold_data[0] , fold_data[1], fold_data[2]])
test1 = fold_data[3]

train2 = pd.concat([fold_data[0] , fold_data[1], fold_data[3]])
test2 = fold_data[2]

train3 = pd.concat([fold_data[1], fold_data[0], fold_data[3]])
test3 = fold_data[1]


train4 = pd.concat([fold_data[1] , fold_data[2], fold_data[3]])
test4 = fold_data[0]


rmsetrain1, rmsetest1  = L2_gradient_descent(X_actual(train1),Y_actual(train1),X_actual(test1),Y_actual(test1),Y_actual(train1).size,0.0035,5500,0.0001,1)

rmsetrain2, rmsetest2  = L2_gradient_descent(X_actual(train2),Y_actual(train2),X_actual(test2),Y_actual(test2),Y_actual(train2).size,0.0035,5500,0.0001,1)

rmsetrain3, rmsetest3  = L2_gradient_descent(X_actual(train3),Y_actual(train3),X_actual(test3),Y_actual(test3),Y_actual(train3).size,0.0035,5500,0.0001,1)

rmsetrain4, rmsetest4  = L2_gradient_descent(X_actual(train4),Y_actual(train4),X_actual(test4),Y_actual(test4),Y_actual(train4).size,0.0035,5500,0.0001,1)

```

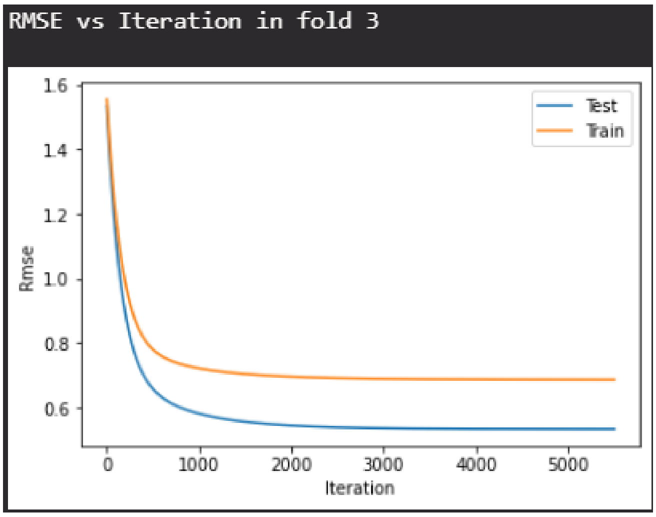
Graph:



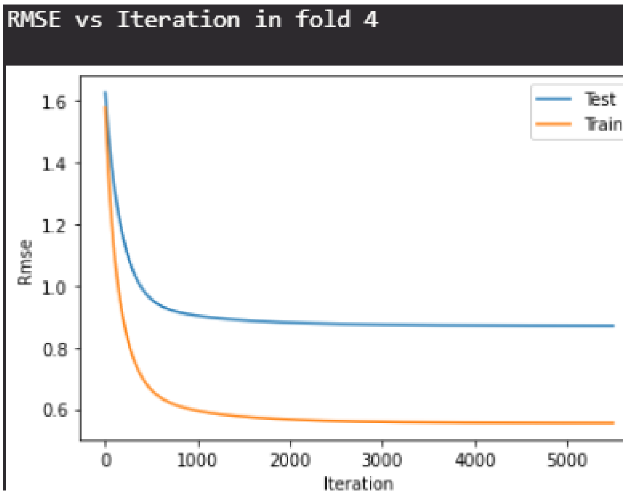
RMSE vs Iteration in fold 2



RMSE vs Iteration in fold 1



RMSE vs Iteration in fold 3



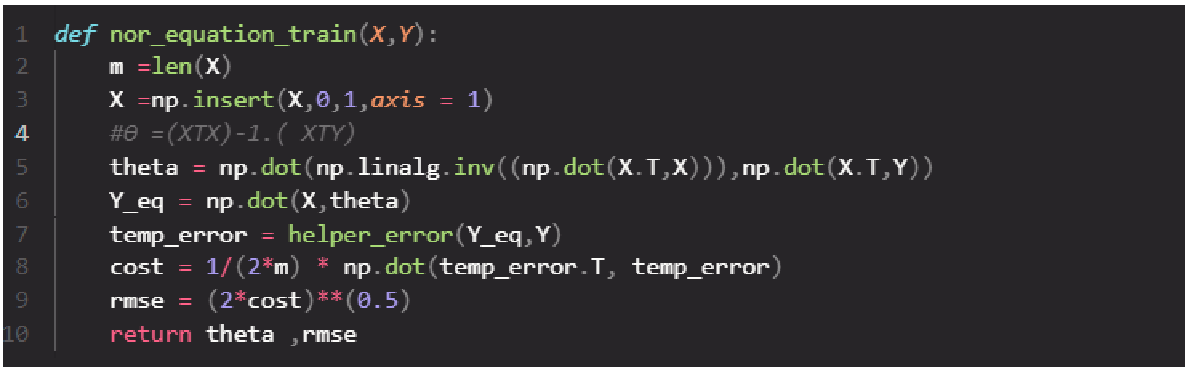
RMSE vs Iteration in fold 4

PART :**D**

Normal equation :

$$\underline{\theta =(X^TX)^{-1}.(X^TY)}$$

Code:



```
def nor_equation_train(X,Y):
    m =len(X)
    X =np.insert(X,0,1,axis = 1)
    #θ =(XTX)-1.( XTY)
    theta = np.dot(np.linalg.inv((np.dot(X.T,X))),np.dot(X.T,Y))
    Y_eq = np.dot(X,theta)
    temp_error = helper_error(Y_eq,Y)
    cost = 1/(2*m) * np.dot(temp_error.T, temp_error)
    rmse = (2*cost)**(0.5)
    return theta ,rmse
```

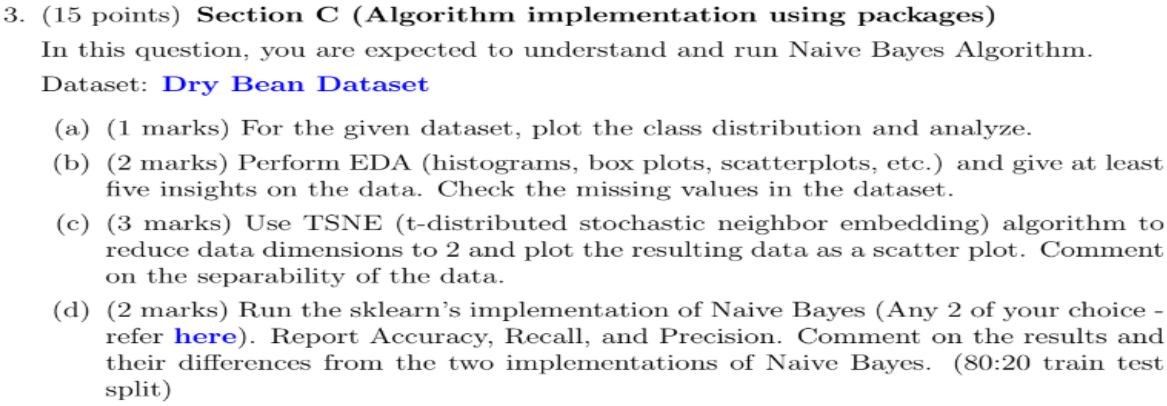
As we know optimal value of K=4

Training our model for K=4

Table for K-value vs RMSE

Fold no.	**Optimal parameters**		RMSE
1	*Parameter*	*Value*	0.627966
	θ_0	0.0207766	
	θ_1	0.0877178	
	θ_2	-0.2477262	
	θ_3	-.04113045	
	θ_4	0.23591649	
	θ_5	0.22846858	
	θ_6	-0.0130103	
2	*Parameter*	*Value*	0.511711
	θ_0	$-2.8157*e^{-4}$	
	θ_1	0.011904	
	θ_2	-0.22159	
	θ_3	-0.42743	
	θ_4	0.22803	
	θ_5	0.20452	
	θ_6	0.002188	
3	*Parameter*	*Value*	0.5320325
	θ_0	$-2.8157e^{-4}$	
	θ_1	0.19404	
	θ_2	0.22159	
	θ_3	-0.42743	
	θ_4	0.22802	
	θ_5	0.20452	
	θ_6	$2.18820e^{-3}$	
4	*Parameter*	*Value*	0.8691125
	θ_0	-0.01102	
	θ_1	0.114846	
	θ_2	-0.234727	
	θ_3	0.3940832	
	θ_4	0.2766535	
	θ_5	0.1821304	
	θ_6	-0.019418	

SECTION – C



3. (15 points) **Section C (Algorithm implementation using packages)**

In this question, you are expected to understand and run Naive Bayes Algorithm.

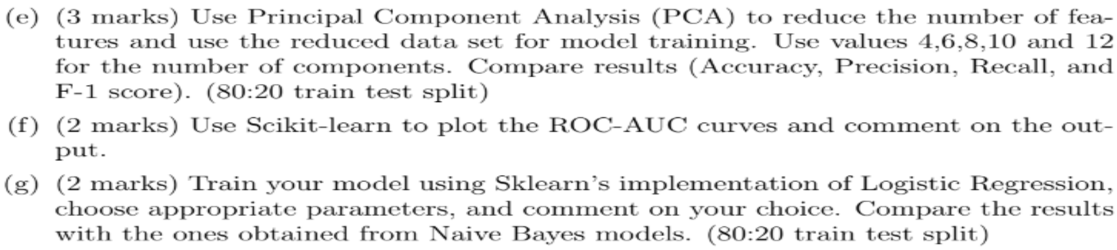
Dataset: **Dry Bean Dataset**

(a) (1 marks) For the given dataset, plot the class distribution and analyze.

(b) (2 marks) Perform EDA (histograms, box plots, scatterplots, etc.) and give at least five insights on the data. Check the missing values in the dataset.

(c) (3 marks) Use TSNE (t-distributed stochastic neighbor embedding) algorithm to reduce data dimensions to 2 and plot the resulting data as a scatter plot. Comment on the separability of the data.

(d) (2 marks) Run the sklearn's implementation of Naive Bayes (Any 2 of your choice - refer **here**). Report Accuracy, Recall, and Precision. Comment on the results and their differences from the two implementations of Naive Bayes. (80:20 train test split)

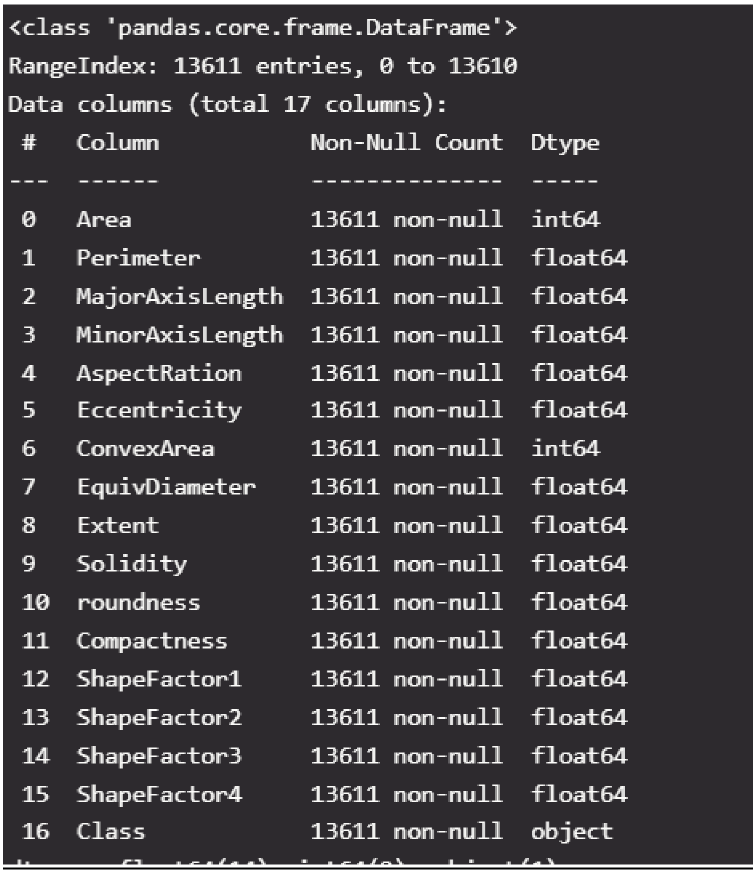


(e) (3 marks) Use Principal Component Analysis (PCA) to reduce the number of features and use the reduced data set for model training. Use values 4,6,8,10 and 12 for the number of components. Compare results (Accuracy, Precision, Recall, and F-1 score). (80:20 train test split)

(f) (2 marks) Use Scikit-learn to plot the ROC-AUC curves and comment on the output.

(g) (2 marks) Train your model using Sklearn's implementation of Logistic Regression, choose appropriate parameters, and comment on your choice. Compare the results with the ones obtained from Naive Bayes models. (80:20 train test split)

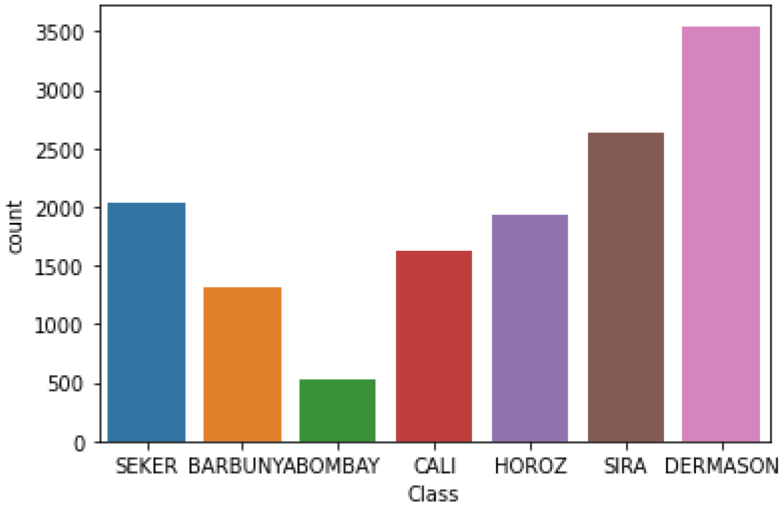
Checking for any missing value In data or NA values



```
<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)
```

PART :**A**

Class distribution:



Analysis

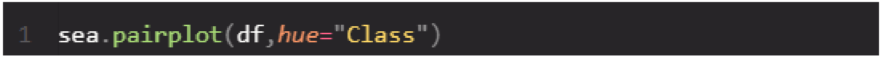
From the above plot, we can say that :

1) Our data set classified dry beans into seven parts based on different features. Therefore it is multi-class classification
2) The Dermason dry bean has the highest count, meaning it has more sample data in the entire data set.
3) The Bombay dry bean has the lowest count, meaning it has least sample data in the entire data set

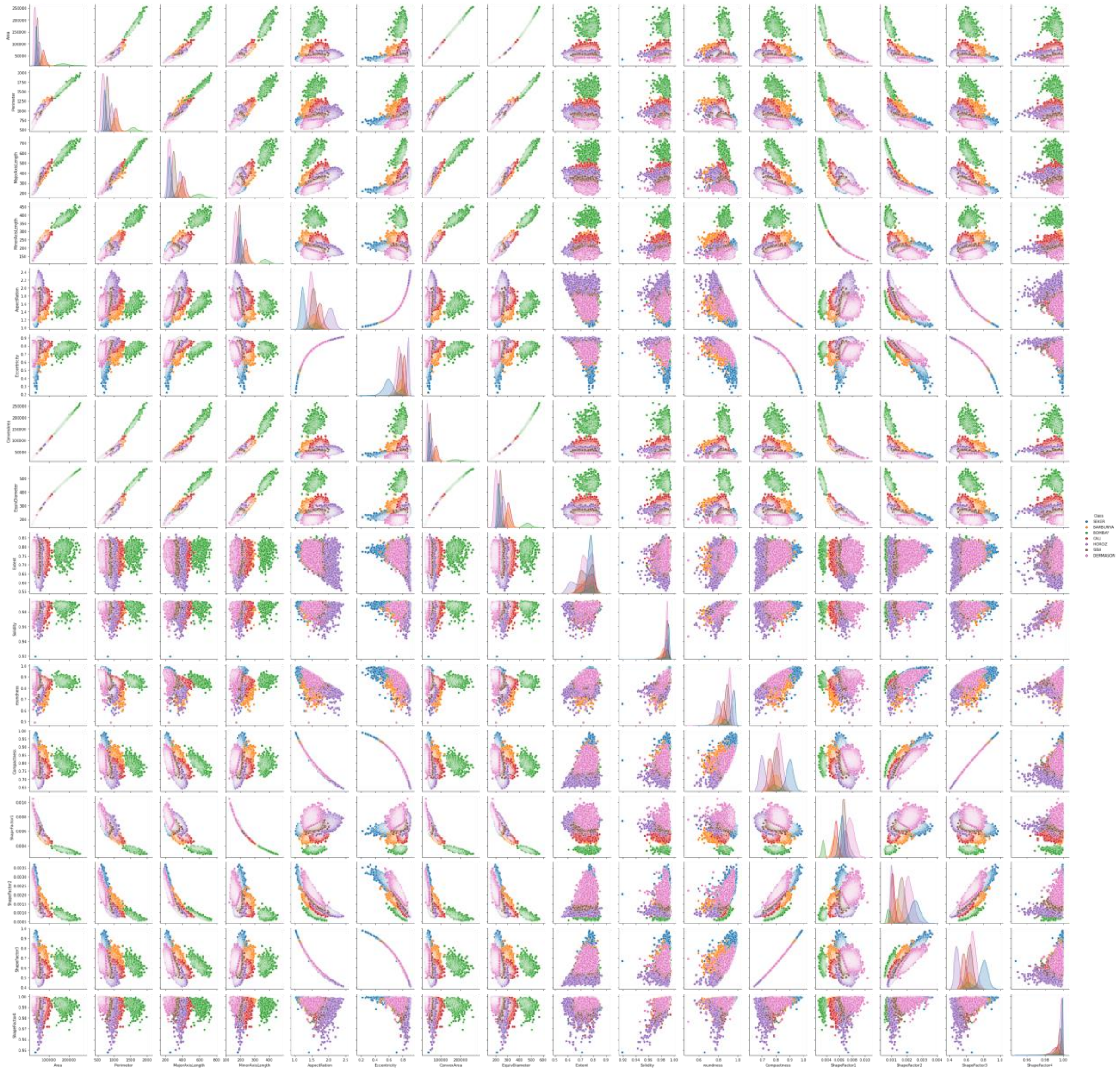
<u>PART :**B**</u>

1) **Scatter plot:**

   Code:

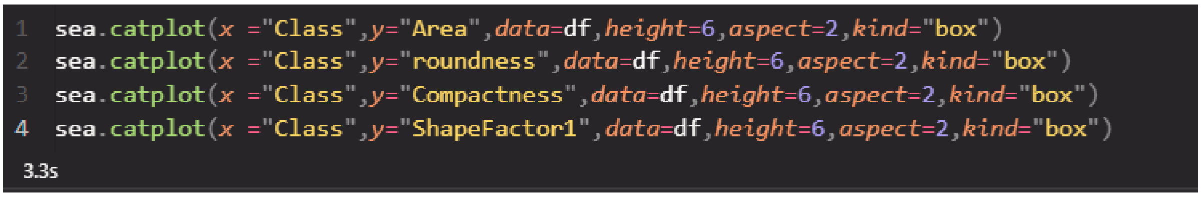


```
sea.pairplot(df,hue="Class")
```



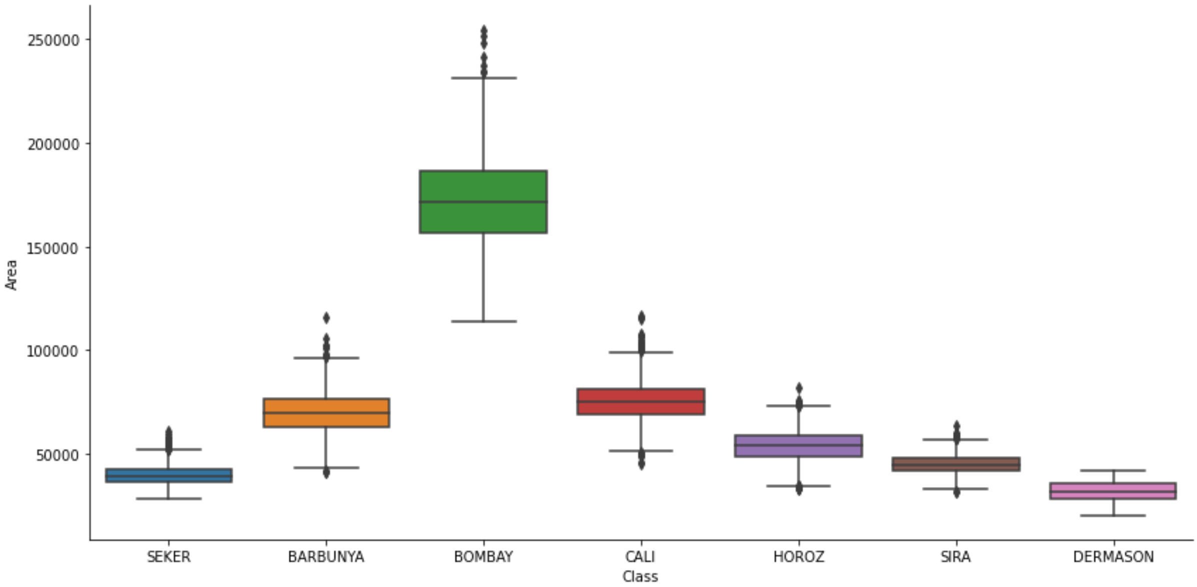
2) Box plots:

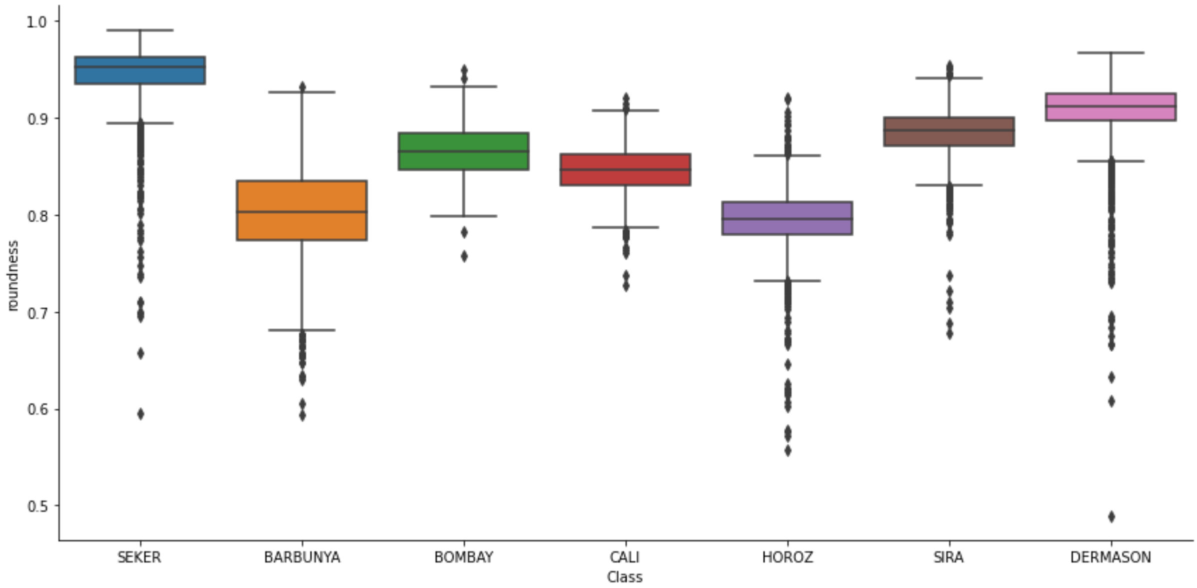
Code:

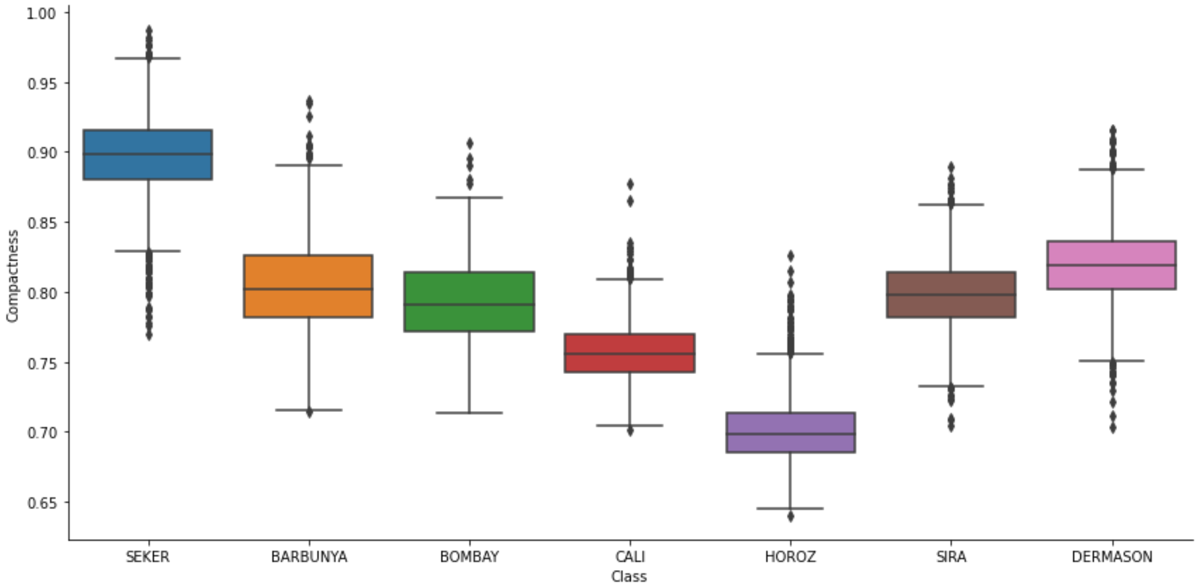


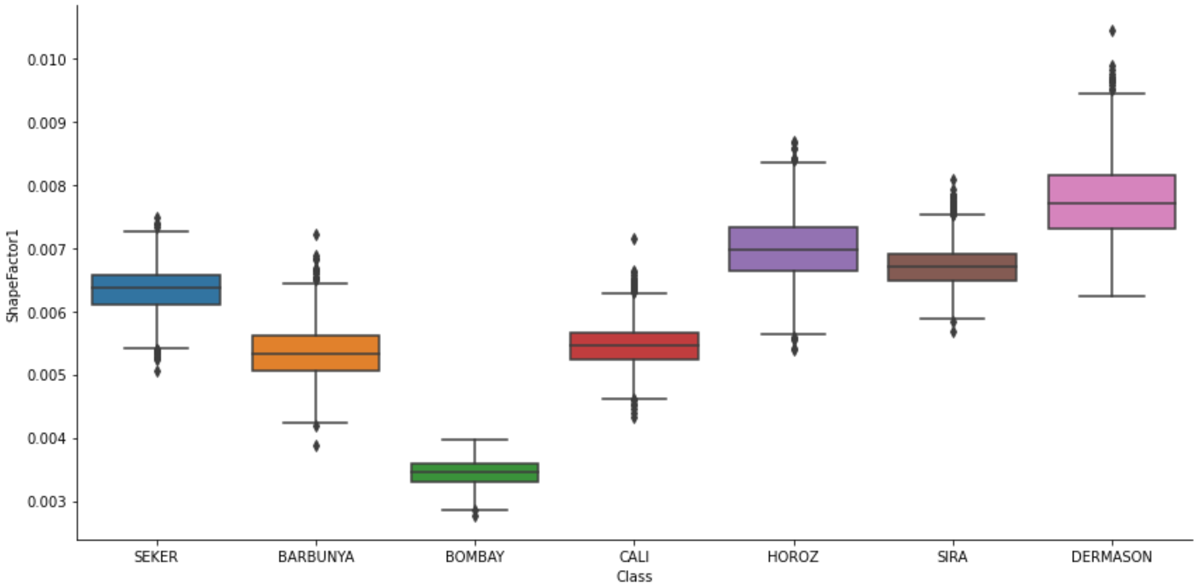
```
sea.catplot(x ="Class",y="Area",data=df,height=6,aspect=2,kind="box")
sea.catplot(x ="Class",y="roundness",data=df,height=6,aspect=2,kind="box")
sea.catplot(x ="Class",y="Compactness",data=df,height=6,aspect=2,kind="box")
sea.catplot(x ="Class",y="ShapeFactor1",data=df,height=6,aspect=2,kind="box")
```

3.3s


250000
200000
150000
100000
50000
Area
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON
Class

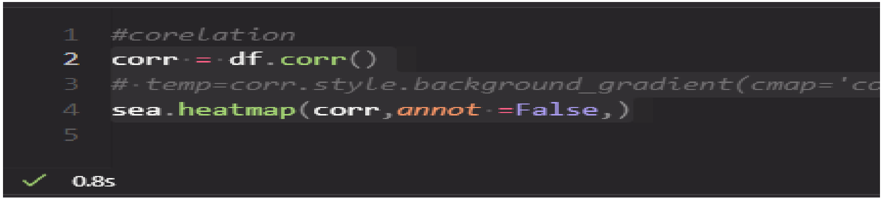

1.0
0.9
0.8
0.7
0.6
0.5
roundness
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON
Class


1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
Compactness
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON
Class

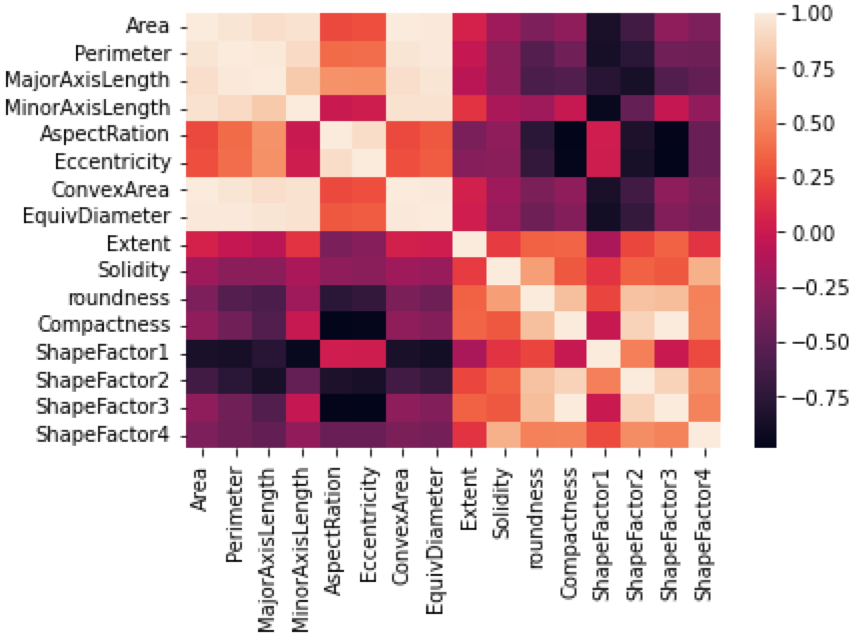


3) Corelation map

Code:



```
#corelation
corr = df.corr()
# temp=corr.style.background_gradient(cmap='co
sea.heatmap(corr,annot =False,)

✓ 0.8s
```



<u>Insights from the data:</u>

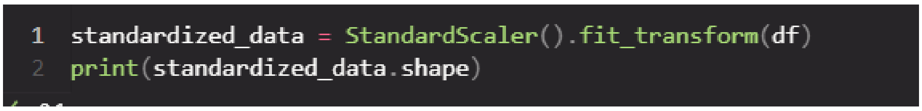
- Using Boxplot, we tell about the distribution of data set, variance, and outliers of features in our data.
  1) Area , perimeter, MajorAxislength, MinorAxisLength has identical distribution
  2) Eccentricity,Solidity,Roundness have higher number of outliers

- Using scatter plot:
  1) Shape factor 2 and major axis length are hyperbolic functions in nature only in positive x and y axis.
  2) Perimeter and major axis length have linear relationship
  3) We also know the density distribution.

- Area, convex have very high positive correlation with each other.
- Minor axis length and shape factor1 have very high negative correlation

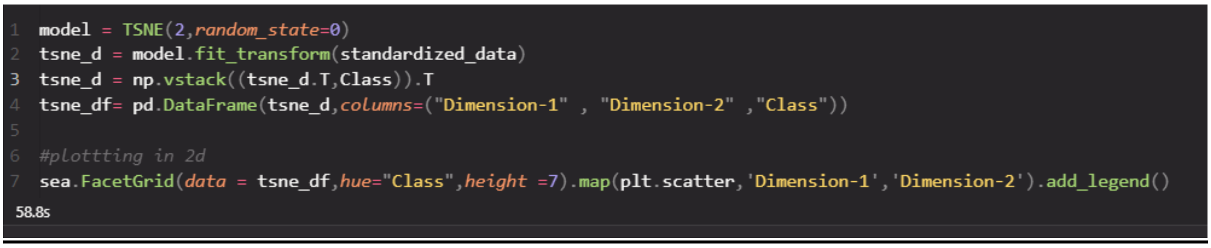
PART :**C**

First we have done data processing

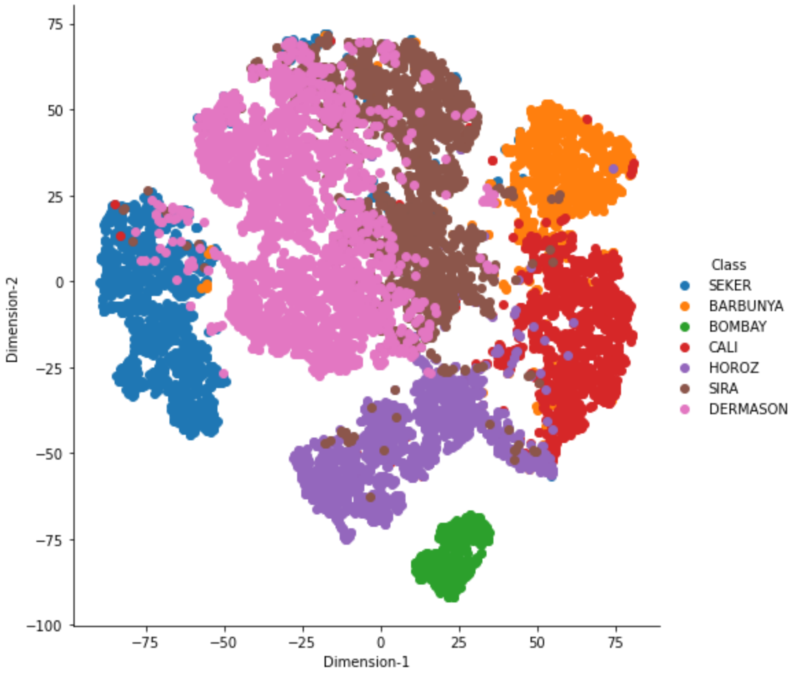


```
standardized_data = StandardScaler().fit_transform(df)
print(standardized_data.shape)
```

Code:



```
model = TSNE(2,random_state=0)
tsne_d = model.fit_transform(standardized_data)
tsne_d = np.vstack((tsne_d.T,Class)).T
tsne_df= pd.DataFrame(tsne_d,columns=("Dimension-1" , "Dimension-2" ,"Class"))

#plottting in 2d
sea.FacetGrid(data = tsne_df,hue="Class",height =7).map(plt.scatter,'Dimension-1','Dimension-2').add_legend()
58.8s
```



Comment on the separability of the data.

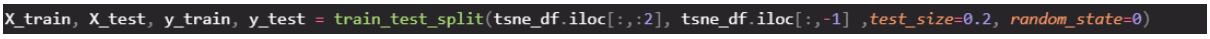
From the above plot, we can say:

- BOMBAY class have low correlation and no data is overlapping with other classes
- DERMANSON and Sira class have high correlation
- Cali class have many outliers as we can see it has many scatter dots far away from it

PART :D

According to given question, I have implemented Bernoulli Naïve Bayes and Gaussian Naïve bayes

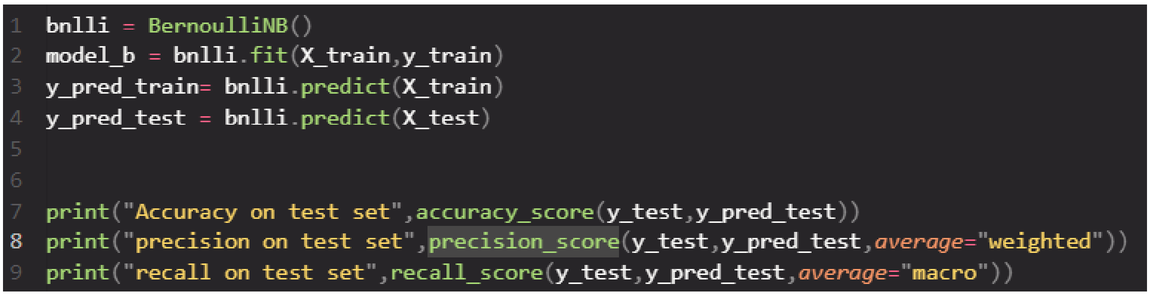
- Split data into test and train set



```
X_train, X_test, y_train, y_test = train_test_split(tsne_df.iloc[:,:2], tsne_df.iloc[:,-1] ,test_size=0.2, random_state=0)
```

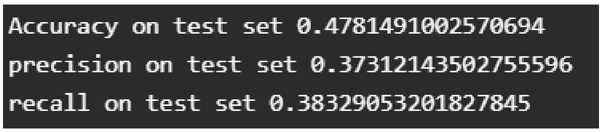
Bernoulli Naïve Bayes

Code:



```
bnlli = BernoulliNB()
model_b = bnlli.fit(X_train,y_train)
y_pred_train= bnlli.predict(X_train)
y_pred_test = bnlli.predict(X_test)


print("Accuracy on test set",accuracy_score(y_test,y_pred_test))
print("precision on test set",precision_score(y_test,y_pred_test,average="weighted"))
print("recall on test set",recall_score(y_test,y_pred_test,average="macro"))
```

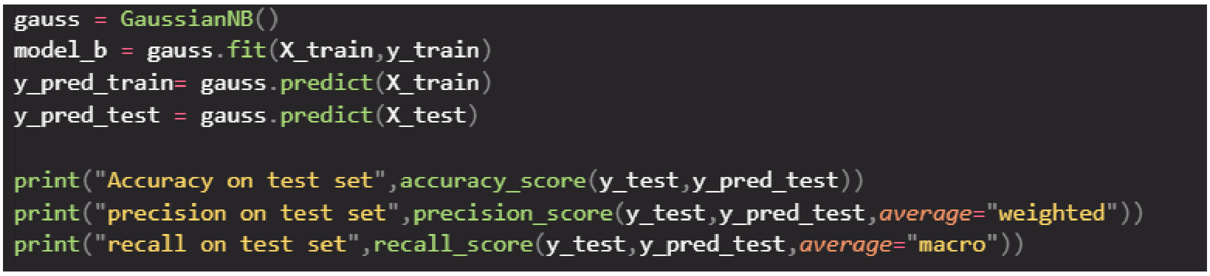
Accuracy, precision ,recall are mention below



```
Accuracy on test set 0.47814910025706694
precision on test set 0.37312143502755596
recall on test set 0.38329053201827845
```

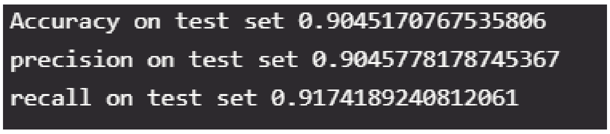
Gaussian Naïve Bayes

Code:



```
gauss = GaussianNB()
model_b = gauss.fit(X_train,y_train)
y_pred_train= gauss.predict(X_train)
y_pred_test = gauss.predict(X_test)

print("Accuracy on test set",accuracy_score(y_test,y_pred_test))
print("precision on test set",precision_score(y_test,y_pred_test,average="weighted"))
print("recall on test set",recall_score(y_test,y_pred_test,average="macro"))
```

Accuracy, precision ,recall are mention below



```
Accuracy on test set 0.9045170767535806
precision on test set 0.9045778178745367
recall on test set 0.9174189240812061
```

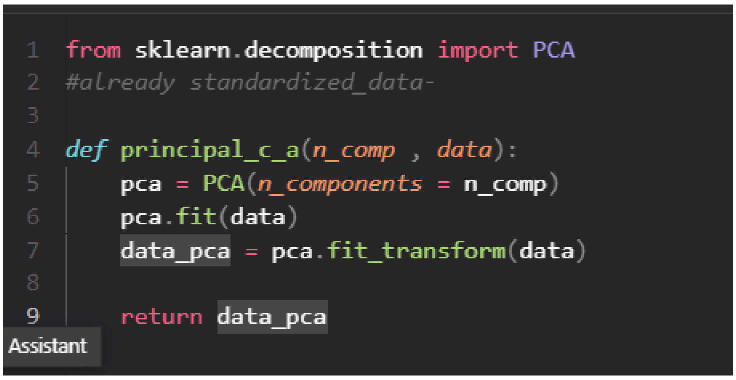
Result: The accuracy, precision and recall of *Gaussian Naïve Bayes*

Far more than *Bernoulli Naïve Bayes*

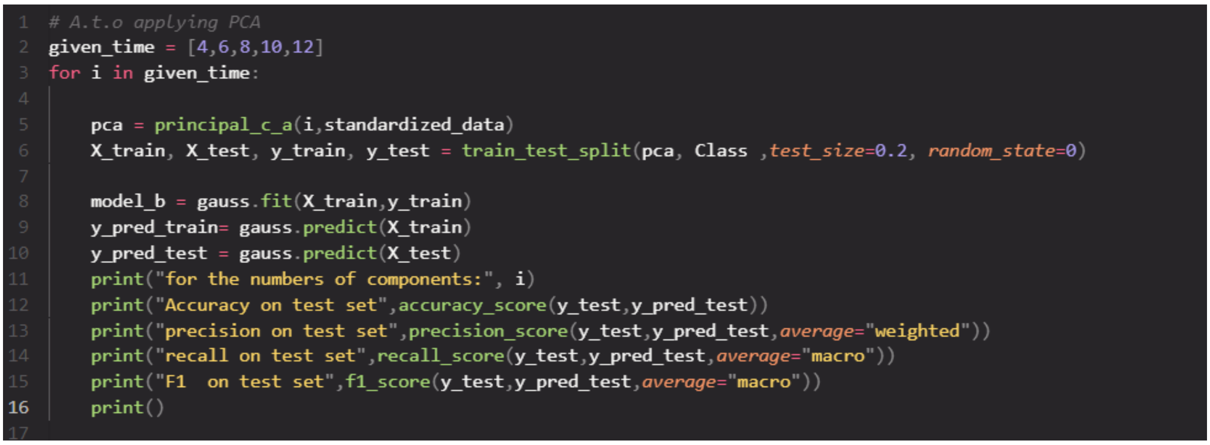
PART :E

Use Principal Component Analysis (PCA) to reduce the number of features and use the reduced data set for model training.

Code:

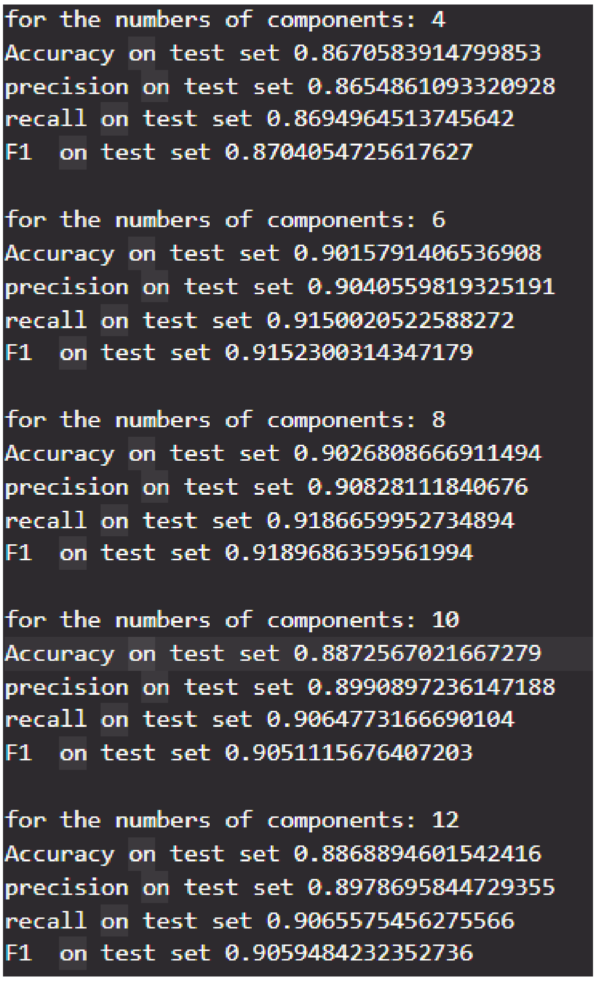


```
from sklearn.decomposition import PCA
#already standardized_data-

def principal_c_a(n_comp , data):
    pca = PCA(n_components = n_comp)
    pca.fit(data)
    data_pca = pca.fit_transform(data)

    return data_pca
```



```
# A.t.o applying PCA
given_time = [4,6,8,10,12]
for i in given_time:

    pca = principal_c_a(i,standardized_data)
    X_train, X_test, y_train, y_test = train_test_split(pca, Class ,test_size=0.2, random_state=0)

    model_b = gauss.fit(X_train,y_train)
    y_pred_train= gauss.predict(X_train)
    y_pred_test = gauss.predict(X_test)
    print("for the numbers of components:", i)
    print("Accuracy on test set",accuracy_score(y_test,y_pred_test))
    print("precision on test set",precision_score(y_test,y_pred_test,average="weighted"))
    print("recall on test set",recall_score(y_test,y_pred_test,average="macro"))
    print("F1  on test set",f1_score(y_test,y_pred_test,average="macro"))
    print()
```

for the numbers of components outputs:

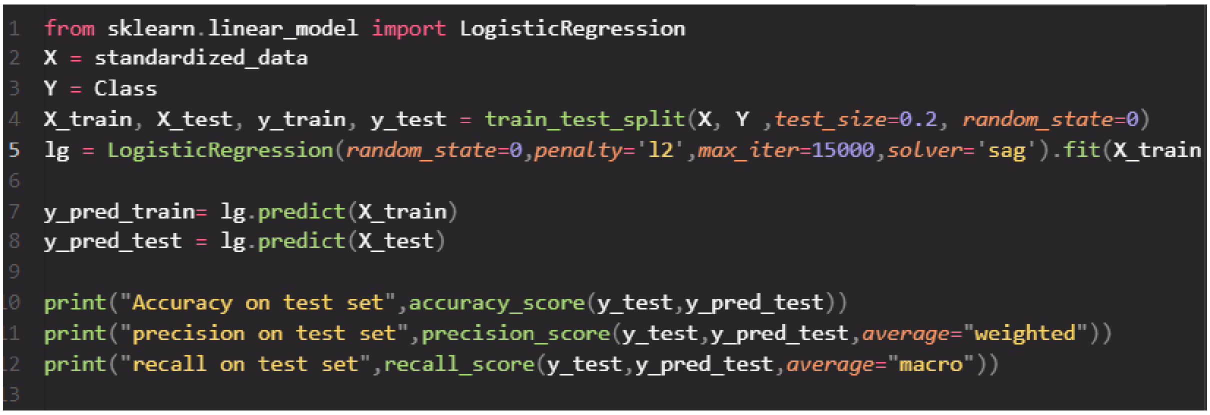


```
for the numbers of components: 4
Accuracy on test set 0.8670583914799853
precision on test set 0.8654861093320928
recall on test set 0.8694964513745642
F1  on test set 0.8704054725617627

for the numbers of components: 6
Accuracy on test set 0.9015791406536908
precision on test set 0.9040559819325191
recall on test set 0.9150020522588272
F1  on test set 0.9152300314347179

for the numbers of components: 8
Accuracy on test set 0.9026808666911494
precision on test set 0.90828111840676
recall on test set 0.9186659952734894
F1  on test set 0.9189686359561994

for the numbers of components: 10
Accuracy on test set 0.8872567021667279
precision on test set 0.8990897236147188
recall on test set 0.9064773166690104
F1  on test set 0.9051115676407203

for the numbers of components: 12
Accuracy on test set 0.8868894601542416
precision on test set 0.8978695844729355
recall on test set 0.9065575456275566
F1  on test set 0.9059484232352736
```

PART :G

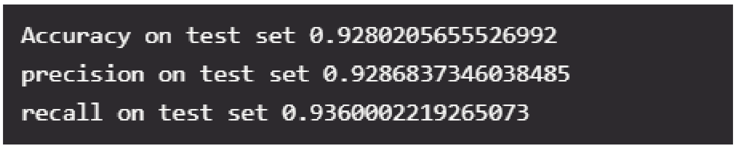
using Sklearn's implementation of Logistic Regression,choose appropriate parameters

Code:



```
from sklearn.linear_model import LogisticRegression
X = standardized_data
Y = Class
X_train, X_test, y_train, y_test = train_test_split(X, Y ,test_size=0.2, random_state=0)
lg = LogisticRegression(random_state=0,penalty='l2',max_iter=15000,solver='sag').fit(X_train

y_pred_train= lg.predict(X_train)
y_pred_test = lg.predict(X_test)

print("Accuracy on test set",accuracy_score(y_test,y_pred_test))
print("precision on test set",precision_score(y_test,y_pred_test,average="weighted"))
print("recall on test set",recall_score(y_test,y_pred_test,average="macro"))
```

Output:



```
Accuracy on test set 0.9280205655526992
precision on test set 0.9286837346038485
recall on test set 0.9360002219265073
```

1) I have taken L2 penality so that model don’t overfit
2) I have taken solver – “sag” because given data is multiclass classification. In multiclass classification only 4 solver available. Both “sag” and “saga” are best for normalized data and our data used here are normalized one only
3) Max_iter = 15000 because after numbers of trials we get best value for this

Result : The result of Logistic Regression far more better than Naive Bayes models.