Assignment 2

Pattern Recognition & Machine Learning

CS

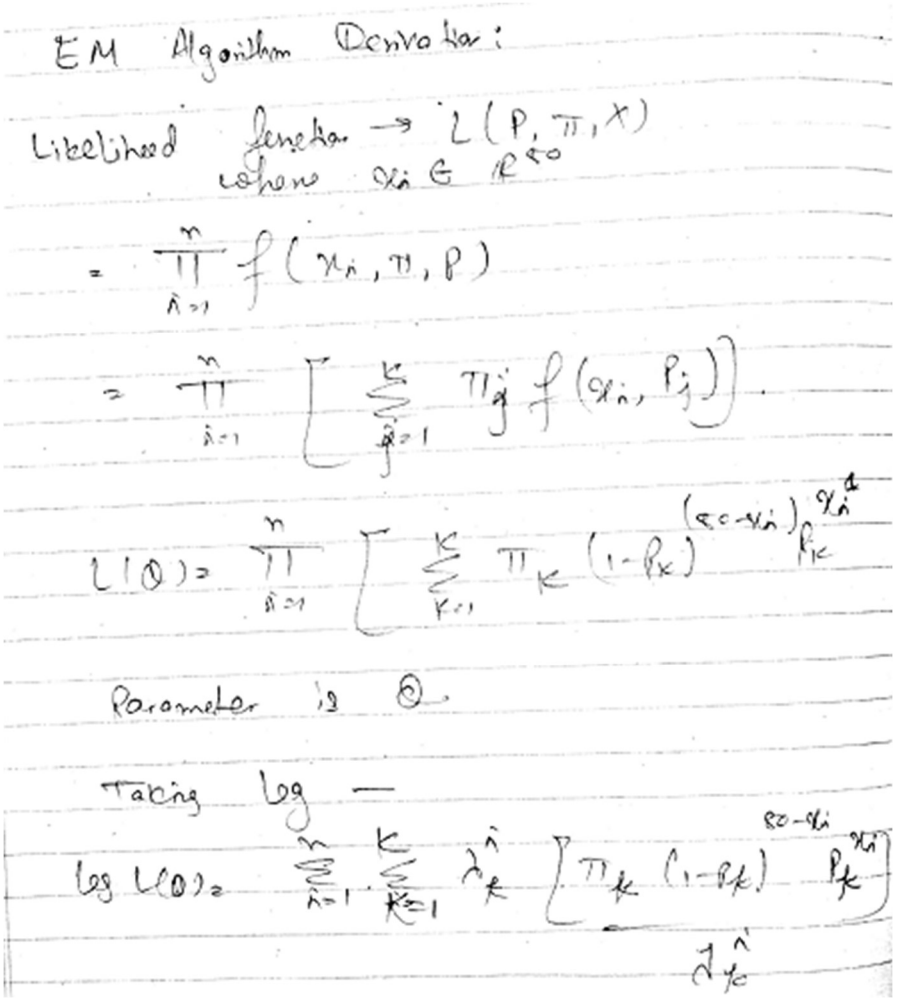
Sheikh Faiz Meraj

CS22M081

(1) You are given a data-set with 400 data points in {0, 1} 50 generated from a mixture of some distribution in the file A2Q1.csv. (Hint: Each datapoint is a flattened version of a {0, 1} 10×5 matrix.)

(i) Determine which probabilisitic mixture could have generated this data (It is not a Gaussian mixture). Derive the EM algorithm for your choice of mixture and show your calculations. Write a piece of code to implement the algorithm you derived by setting the number of mixtures K = 4. Plot the log-likelihood (averaged over 100 random initializations) as a function of iterations.

EM derivation is as follows:



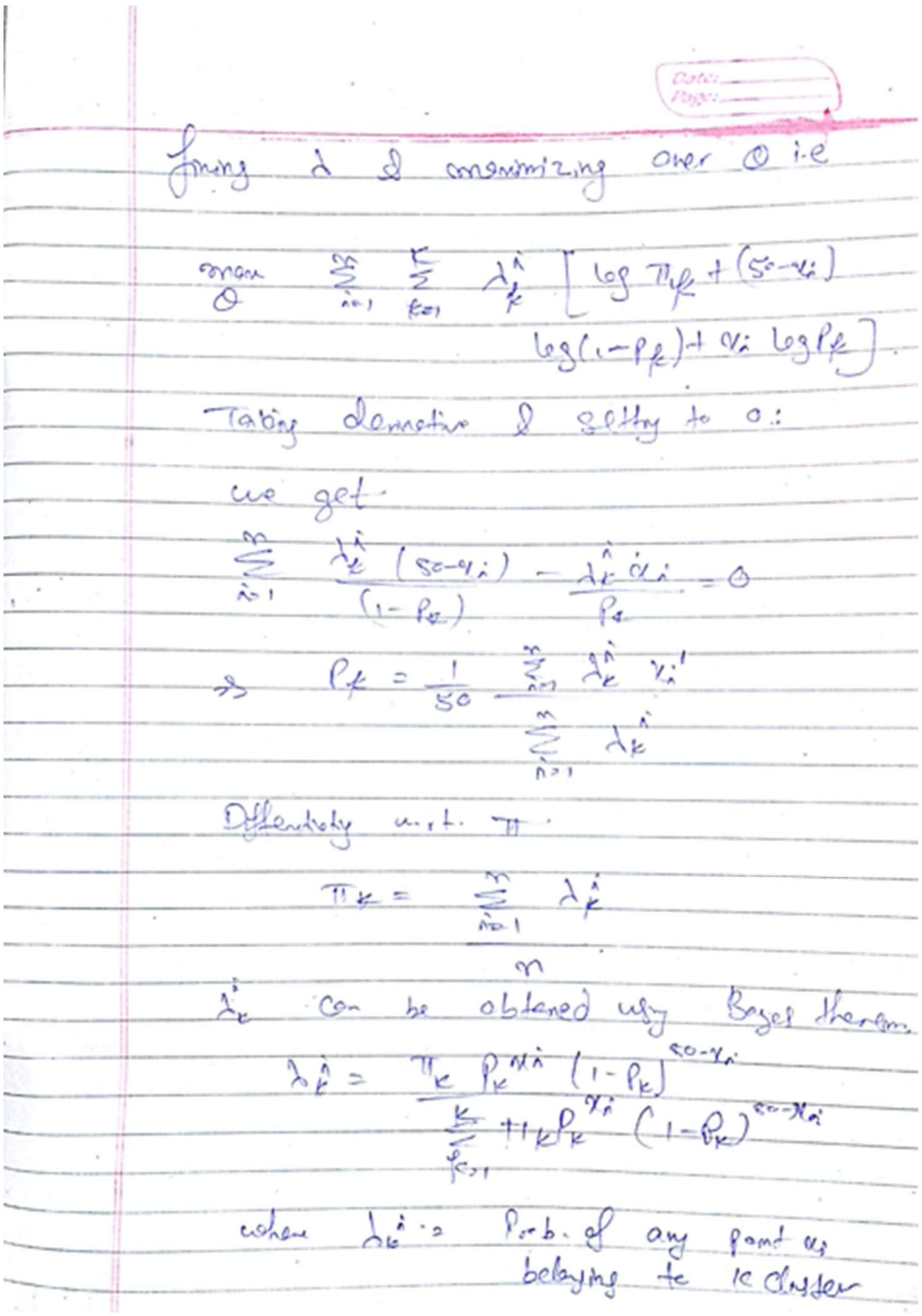
Parameter is θ.

Taking log —

$$\log L(\theta)_2 \sum_{i=1}^{n} \sum_{k=1}^{K} \hat{\lambda}_k \left[\frac{\pi_k (1-p_k)^{80-x_i} p_k^{x_i}}{\hat{\lambda}_k} \right]$$

from Jensen's inEquality:

$$\sum_{i=1}^{n} \sum_{k=1}^{K} \hat{\lambda}_k \log \left[\frac{\pi_k (1-p_k)^{80-x_i} p_k^{x_i}}{\hat{\lambda}_k} \right]$$



Fixing λ & maximizing over θ i.e

$$\underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{n} \sum_{k=1}^{K} \lambda_k^i \left[\log \pi_k + (50 - x_i) \log(1 - p_k) + x_i \log p_k \right]$$

Taking derivative & setting to 0:

we get

$$\sum_{i=1}^{n} \frac{\lambda_k^i (50 - x_i)}{(1 - p_k)} - \frac{\lambda_k^i x_i}{p_k} = 0$$

$$\Rightarrow p_k = \frac{1}{50} \frac{\sum_{i=1}^{n} \lambda_k^i x_i'}{\sum_{i=1}^{n} \lambda_k^i}$$

Differentiating w.r.t. π

$$\pi_k = \frac{\sum_{i=1}^{n} \lambda_k^i}{n}$$

λ_k^i can be obtained using Bayes theorem

$$\lambda_k^i = \frac{\pi_k p_k^{x_i} (1 - p_k)^{50 - x_i}}{\sum_{k=1}^{K} \pi_k p_k^{x_i} (1 - p_k)^{50 - x_i}}$$

where λ_k^i = Prob. of any point x_i belonging to k^{th} cluster

Code snippet for the EM algorithm is as follows:

```python
def InitcomputeP(mu):
  P = mu.sum(axis = 1)
  P = np.multiply(1 / 50 , P)
  print(P.shape)
  return P
```

```python
def InitcomputePi(Z):
  Pi = []
  c1 = c2 = c3 = c4 = 0
  for i in Z:
    if i == 1:
      c1 = c1 + 1
    elif i == 2:
      c2 = c2 + 1
    elif i == 3:
      c3 = c3 + 1
    elif i == 4:
      c4 = c4 + 1
  p1 = c1 / 400
  p2 = c2 / 400
  p3 = c3 / 400
  p4 = c4 / 400
  Pi.append(p1)
  Pi.append(p2)
  Pi.append(p3)
  Pi.append(p4)
  return Pi
```

```
[ ]  1 P = InitcomputeP(mu)
     2 print(P)
     3 Pi = InitcomputePi(Z)
     4 print(Pi)
```

```
(4,)
[0.28930693 0.3625     0.35175573 0.372     ]
[0.2525, 0.08, 0.655, 0.0125]
```

```
[ ]  1 def computexi():
     2   no_ones = []
     3   for i in range(400):
     4     one = np.sum(df.iloc[i,0:])
     5     no_ones.append(one)
     6   print(no_ones)
     7   return no_ones
```

```
[ ]  1 no_ones = computexi()
```

```
 1 import numpy as np
 2 def computeLambda(Pi, P, no_ones):
 3   lammbda = np.zeros((400 ,4))
 4   for i in range(400):
 5     denomin = 0
 6     for j in range(4):
 7       denomin = denomin + Pi[j] * P[j]**no_ones[i] * (1-P[j])**(50 - no_ones[i])
 8     for k in range(4):
 9       numret = Pi[j] * np.power(P[j],no_ones[i]) * np.power((1-P[j]), (50 - no_ones[i]))
10       lamd = numret / denomin
11       lammbda[i][k] = lamd
12   print(lammbda)
13   return lammbda
14
15 lammbda = computeLambda(Pi, P, no_ones)
```

```
[ ]
def computeP(lammbda , no_ones):
  P = np.zeros(4)
  numer = 0
  denom = np.sum(lammbda, axis = 0)
  print(denom)
  for i in range(4):
    for j in range(400):
      numer = numer + lammbda[j][i] * (no_ones[j]/50)
      # print(numer)
    numer = numer / denom[i]
    P[i] = numer
  return P


P = computeP(lammbda, no_ones)
print('P' , P)

```

```
[ ]
def computeP(lammbda , no_ones):
  P = np.zeros(4)
  numer = 0
  denom = np.sum(lammbda, axis = 0)
  print(denom)
  for i in range(4):
    for j in range(400):
      numer = numer + lammbda[j][i] * (no_ones[j]/50)
      # print(numer)
    numer = numer / denom[i]
    P[i] = numer
  return P


P = computeP(lammbda, no_ones)
print('P' , P)

```

iii. Run the K-means algorithm with K = 4 on the same data. Plot the objective of K − means as a function of iterations.

Following is the code snippet for the k-means algorithm for the given dataset.

```
[ ]  1 #Initialization of the K means
     2 def Initialization(df,Z, k):
     3   import random
     4   centroid = []
     5   for i in range(k):
     6     r = random.randint(0,400)
     7     Z[r] = i+1
     8     centroid.append(df.iloc[r])
     9   return Z , centroid
```

```
[ ]  1 # Computing the Euclidean distcance for the error.
     2 def EuclideanDist (A , B):
     3   return np.sqrt(np.sum((A - B)**2))
```

```
     1 # Computing the error of the assignment step.
     2 def ComputeError(Z , centroid, k):
     3   sum = 0
     4   for i in range(len(df)):
     5     for j in range(0,k):
     6       if Z[i] == j+1:
     7         sum = sum + EuclideanDist(df.iloc[i], centroid[j])
     8   return sum
```

```
[ ]
# Computing the mean of each of the clusters.
def CalculateMean(centroid ,Z ,k):
    centroid = np.zeros((k+1,50))
    Times = np.zeros(k+1)
    for i in range(len(df)):
      for j in range(k+1):
        if Z[i] == (j+1):
          centroid[j] = np.add(centroid[j], df.iloc[i])
          Times[j] = Times[j] + 1
    for j in range(k):
      centroid[j] = np.multiply(centroid[j] , 1 / Times[j])
    return centroid

```

```
[ ]
# Computing the reassignment of each of the points for the clusters.
def Reassignment(centroid , Z , k):
  isReassign = False
  for i in range(len(df)):
    if Z[i] == 0:
      dist = 10000
    else:
      dist = EuclideanDist(centroid[int(Z[i]) - 1], df.iloc[i])
    for j in range(k):
      ED  = EuclideanDist(centroid[j] , df.iloc[i])
      if dist > ED:
        isReassign = True
        dist = ED
        Z[i] = j+1
  return Z, isReassign

```

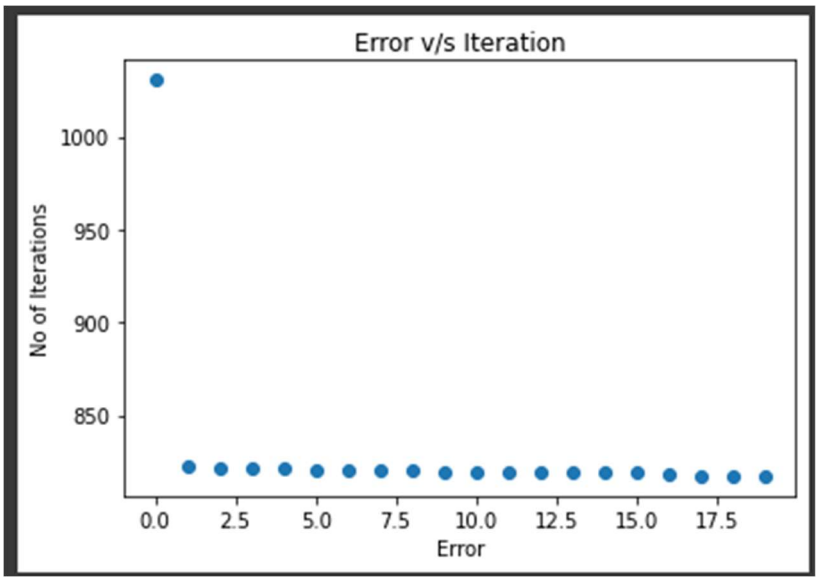
```python
# Plotting the graph of Error v/s No of iterations
def PlotError(ErrorSum):
  import matplotlib.pyplot as plt
  Iter = []
  for i in range(len(ErrorSum)):
    Iter.append(i)
  plt.scatter(Iter,ErrorSum)
  plt.title('Error v/s Iteration')
  plt.xlabel('Error')
  plt.ylabel('No of Iterations')
  plt.show()
```

```python
# Running th K Means algorithm for K = 4
k = 4
Z = np.zeros(400)
ErrorSum = []
Z, centroid = Initialization(df,Z , k)
centroid  = np.array(centroid)
Z, isReassign = Reassignment(centroid, Z, k)
ErrorSum.append(ComputeError(Z, centroid , k))

while isReassign:
  centroid = CalculateMean(centroid, Z, k)
  Z, isReassign = Reassignment(centroid, Z, k)
  ErrorSum.append(ComputeError(Z, centroid , k))
centroid  = np.array(centroid)

```

The error plot v/s the number of iterations after running the k means comes out to be as follows:



And the Error values comes out to be as:

```
[1030.529696607429,  822.3213665089061,  821.5034651299643,
821.0839635549501,  820.9569314319076,  820.708525425563,
820.342762948325,  820.0051716011207,  819.8457937253746,
819.6902233246408,  819.5071436513126,  819.3122145114503,
819.2265290532854,  819.1154423278472,  819.0532195432148,
818.8814370395679,  818.5388284731514,  817.6130983204259,
817.3496250537295, 817.1938824468676]
```

(2) You are given a data-set in the file A2Q2Data train.csv with 10000 points in (R 100 , R) (Each row corresponds to a datapoint where the first 100 components are features and the last component is the associated y value).

i. Obtain the least squares solution wML to the regression problem using the ana lytical solution.

Following is the snippet of the code for computing the closed form of wML:

```
[ ]  1 def w_Maximum_Likelihood(x, x_transpose, y):
     2   xxt = np.matmul( x_transpose ,x)
     3   xxt_inv = np.linalg.inv(xxt)
     4   xy = np.matmul(x_transpose, y)
     5   w_ml = np.matmul(xxt_inv, xy)
     6   print(w_ml)
     7   return w_ml
```

The formula for w_{ML} is as follows:

$W_{ML} = (X^T X)^{-1} X^T Y$

Here the shape of the w_{ML} is (100,1) which is same as the number of features.

ii. Code the gradient descent algorithm with suitable step size to solve the least squares algorithms and plot $||w^t - w^{ML}||2$ as a function of t. What do you observe?

Following are the code snippet of the code for the Gradient Descent Algorithm.

Here As per observation, I took the alpha as

alpha = 0.001

No of iterations = 600

and the updation for the w is as per the following formulae:

$w = w - X^T X$

```
[8]  1 def cost(x, w , n_datapoints):
     2   cost = 0
     3   xw = np.matmul(x, w)
     4   for i in range(n_datapoints):
     5     cost += (1/ n_datapoints) * (xw[i][0] - y[i][0])**2
     6   # print(cost)
     7   return cost
```

```
     1 def gd(x , iter, w, n_features,n_datapoints, alpha):
     2   cost_arr = []
     3   for i in range(iter):
     4     # print(x.shape , w.shape)
     5     xw = np.matmul(x,w)
     6     x_transpose = np.transpose(x)
     7     xw_y = np.subtract(xw, y)
     8     xxw = np.matmul(x_transpose, xw_y)
     9     dw = np.multiply(2/n_datapoints , xxw)
    10     w = np.subtract(w , np.multiply(alpha,dw))
    11     error = cost(x , w ,n_datapoints)
    12     cost_arr.append(error)
    13   return (cost_arr,w)
```

```
[10]  1 def plot_error(cost_arr , iter):
      2   import matplotlib.pyplot as plt
      3   epochs = range(iter)
      4   plt.xlabel('No of Iterations')
      5   plt.ylabel('Error Cost')
      6   plt.title('Error v/s No of Iterations')
      7   plt.plot(epochs , cost_arr)
```

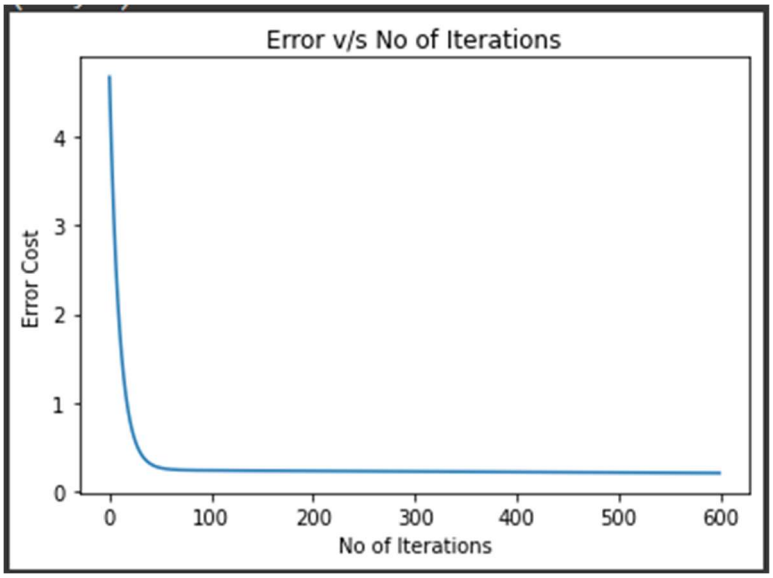
```
[11]  1 # w = np.random.rand(n_features)
      2 w = np.zeros(n_features)
      3 w = np.expand_dims(w, axis=1)
      4 print(w.shape)
      5 alpha = 0.001
      6 iter = 600
      7 cost_arr , w = gd(x,iter,w ,n_features, n_datapoints, alpha)
      8 plot_error(cost_arr, iter)
```

Here if we plot the error cost with respect to the number of iterations then the errors decreases exponentially in the starting as the number of iteration increases.

Uptil number of iteration = 50, the error drops from more than 4 to near to 0.

And after that the error decreases quite slowly as the number of iterations increases.

The plot for the Error Cost v/s Number of Iterations is plotted as follows:



iii. Code the stochastic gradient descent algorithm using batch size of 100 and plot $|w^t - w_{ML}|^2$ as a function of t. What are your observations?

The following are the code snippet for the stochastic gradient algorithm that I wrote :

```
1
2 def sgd(data,x, y , iter, w, n_features,n_datapoints, alpha , batchsize):
3   w_sum = np.zeros(n_features)
4   w = np.expand_dims(w, axis=1)
5   no_rounds = int(n_datapoints / batchsize)
6   Total_cost  = []
7   datat = np.transpose(x)
8   w_ml = w_Maximum_Likelihood(x, datat, y)
9   for i in range(iter):
10    data_x , data_y = generateData(data, batchsize, n_datapoints)
11    data_xt = np.transpose(data_x)
12    w_wml = np.subtract(w , w_ml)
13    err = np.linalg.norm(w_wml)
14    print(i , err)
15    xx_t = np.matmul(data_xt, data_x)
16    xx_tw = np.matmul(xx_t, w)
17    xy = np.matmul(data_xt, data_y)
18    dw = 2 * (np.subtract(xx_tw , xy))
19    w = np.subtract(w , np.multiply(alpha, dw))
20    Total_cost.append(err)
21    w_sum = np.add(w_sum , w)
22  print(w)
23  return w_sum , Total_cost
```

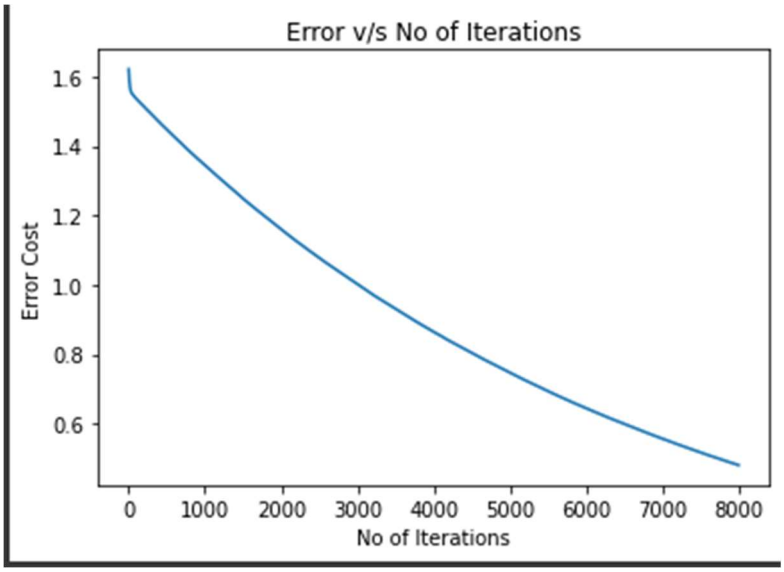
Here, we are taking the batch size of 100 for the datapoints.

Stochastic gradient descent computes the w with less number of computations without going over to the complete dataset.

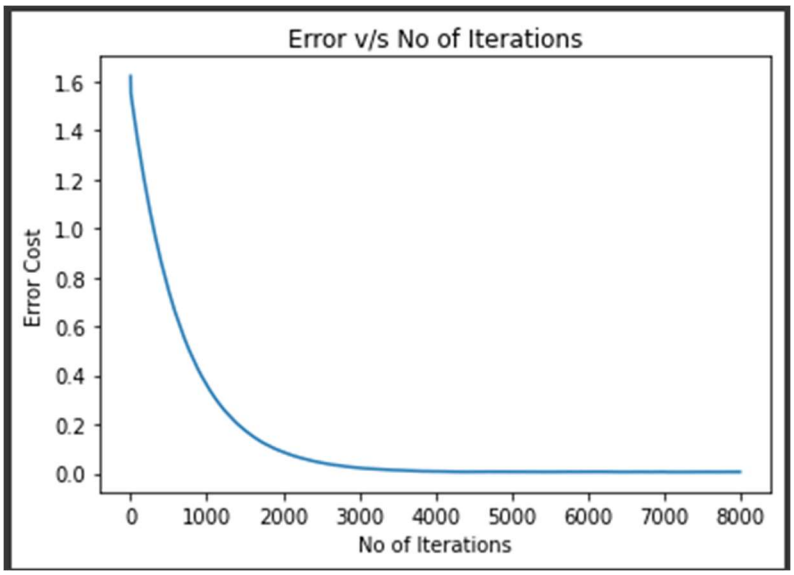
Since it doesn't go over the complete dataset and it considers only a certain batchsize(100 in our case) of dataset , so the error some time increases which calculating w for the number of iterations as the batch may not be the complete representation of the whole dataset.

But as we run it for the number of iterations, so the overall error decreases and we get the correct w.

For the batchsize of 100 , we get the following error vs Number of iteration graph:



If we increase the batchsize to 1000, the plot is as follows:



iv. Code the gradient descent algorithm for ridge regression. Cross-validate for var ious choices of λ and plot the error in the validation set as a function of λ. For the best λ chosen, obtain wR. Compare the test error (for the test data in the file A2Q2Data test.csv) of wR with wML. Which is better and why?

Following is the code snippet for the Ridge Regression:

```
# Function to run the Ridge Linear Regression Algorithm
def RidgeGD(data,x,y, n_datapoints, n_features , alpha, iter):
  w_list = []
  error_list = []
  minError = 10000000000
  minLambda = -1
  from sklearn.model_selection import train_test_split
  xtrain,xtest,ytrain,ytest = train_test_split(x,y,test_size=0.2,random_state=4)
  # print(xtrain, xtest, ytrain, ytest)
  xtrain = np.array(xtrain)
  ytest = np.array(ytest)
   (variable) ytest: ndarray
  ytest = np.array(ytest)

  # xtrain, ytrain, xtest, ytest = TestTrainSplit(df, x, y)
  xtrain_t = np.transpose(xtrain)
  #computing the x^Tx separetely.
  xtrain_txtrain = np.matmul(xtrain_t, xtrain)
  for lammbda in range(0,iter):
    w = np.ones(n_features)
    w = computeGradientDescent(5000, xtrain, ytrain,w, xtrain_txtrain, n_features, lammbda, alpha)
    error = errorCost(xtest, ytest, w , 2000)
    print(lammbda, error)
    error_list.append(error)
    if minError > error:
      minError = error
      minLambda = lammbda
    w_list.append(w)
  return error_list , minLambda
```

```
[6]
def computeGradient(x, y,xtrain_txtrain,w, lammbda):
  # compute dw
  x_t = np.transpose(x)
  dw = xtrain_txtrain
  dw = np.matmul(dw, w)
  x_ty = np.matmul(x_t, y)
  lambdaw = np.multiply(lammbda, w)
  dw = np.subtract(dw, x_ty)
  dw = np.add(dw, lambdaw)
  dw = np.multiply(2, dw)
  return dw

[7]
def cost(x,y, w , n_datapoints):
  cost = 0
  xw = np.matmul(x, w)
  for i in range(n_datapoints):
    cost += (1/ 2*n_datapoints) * (xw[i][0] - y[i][0])**2
  return cost

[8]
def computeGradientDescent(n_iteration , x, y ,w, xtrain_txtrain,n_features, lammbda, alpha):
  for i in range(n_iteration):
    dw = computeGradient(x, y,xtrain_txtrain, w , lammbda)
    alpha_dw = np.multiply(alpha, dw)
    w = np.subtract(w, alpha_dw)
  return w
```

```
def errorCost(x, y , w , n_datapoints):
  cost = 0
  xw = np.matmul(x, w)
  error = np.linalg.norm(y - xw)
  error = error ** 2
  return error
```

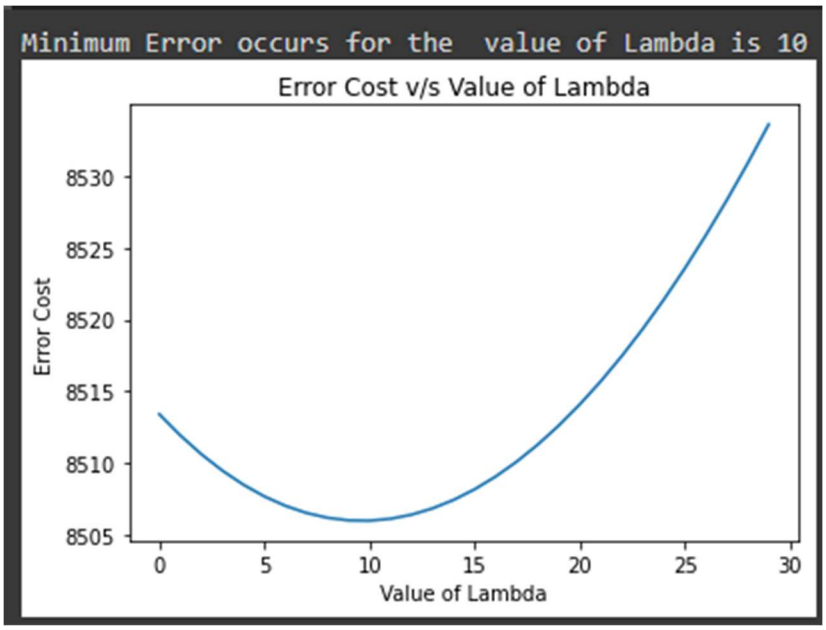
In the ridge regression, I split the complete data set in 80:20 ratio , and train for training data set to get the w.

After getting w and appropriate Lambda, I computed the error vs the value of Lambda graph and found out that the minimum error occurs for the value of lambda = 10.

After that I computed the w^R for the testing dataset.

Comparing the W_{ML} and W_R , I found out that the error in ridge comes out to be quite large i.e. `8505.970671433826` (after optimizing a lot) as compared $w_{ML.}$ which is `184.4323876725`.

After plotting the plot of value of Lambda v/s the value of error, it comes out as:



In the graph as we increase the value of lambda, the error decreases first as it computes the appropriate w. Firstly the w is the overfitted value for the training set, It reaches a minimum value and then again rises. Because as the value of lambda increases, value of error increases on training set but decreases on the testing set. But if we increase the lambda too high then the w began to come down towards x- axis and hence the error rises.