VISWESWAR SIRISH PARUPUDI

2020AAPS0330H

NNFL ASSIGNMENT 1

GOOGLE COLAB FILES LINKS

Q1.

https://colab.research.google.com/drive/1XEbNlOKeIStotKweoLVxN_oboveNCOIj?usp=sharing

Q2.

https://colab.research.google.com/drive/1TrGgslYJZzUvZWDvnIAm1dJA-8pKB8Iw?usp=sharing

Q3.

https://colab.research.google.com/drive/1BOApUXJlJL_XLvQl6ZdvDA9amDCy3lvt?usp=sharing

Q4. (ONE VS ALL)

https://colab.research.google.com/drive/1RgQgCO5j0yKp01Ed8g7PHdOrEw7ZVJ9Q?usp=sharing

Q4. (ONE VS ONE)

https://colab.research.google.com/drive/1IeLq9IEkyURCr6T8mN5ADlWIh0QFO3zr?usp=sharing

Q5.

https://colab.research.google.com/drive/1BYYin2mnLBApaERlhtNlKfOgWgY9nW5z?usp=sharing

Q1

MOUNTING THE DRIVE

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

importing libraries

```
import pandas as pd
import math
import numpy as np
import matplotlib.pyplot as plt
import sklearn
```

Defining Cost Functions

```
def cost_function_l2(X,y,w,lamb):

    hypothesis = np.dot(X,w.T)
    #J = (1/(2*len(y)))*np.sum((hypothesis-y)**2)
    J= (1/(2*len(y)))*np.sum((hypothesis-y)**2) + (lamb/2)*np.sum(w**2)
    return J


def cost_function_l1(X,y,w,lamb):
    w=w.ravel()
    hypothesis = np.dot(X,np.transpose(w))
    #J = (1/(2*len(y)))*np.sum((hypothesis-y)**2)
    J= (1/(2*len(y)))*np.sum((hypothesis-y)**2) + (lamb/2)*np.sum(abs(w))
    return J


def cost_function(X,y,w):

    hypothesis = np.dot(X,w.T)
    J = (1/(2*len(y)))*np.sum((hypothesis-y)**2)
#     J= (1/(2*len(y)))*np.sum((hypothesis-y)**2) + (lamb/2)*np.sum(w**2)
    return J
```

Defining gradient descent functions

```
def batch_gradient_descent(X,y,w,alpha,iters):
    cost_history = np.zeros(iters)
    for i in range(iters):
```

```
        hypothesis = np.dot(X,w.T)
        w = w - (alpha/len(y)) * np.dot(hypothesis - y, X)
        cost_history[i] = cost_function(X,y,w)
    return w, cost_history



def stochastic_gradient_descent_l2(X,y,w,alpha, iters,lamb):
    cost_history = np.zeros(iters)
    for i in range(iters):
        rand_index = np.random.randint(len(y)-1)
        ind_x = X[rand_index:rand_index+1]
        ind_y = y[rand_index:rand_index+1]
        w = w*(1-alpha*lamb) - alpha * (ind_x.T.dot(ind_x.dot(w) - ind_y))
        cost_history[i] = cost_function_l2(ind_x,ind_y,w,lamb)
    return w, cost_history


def MB_gradient_descent_l1(X,y,w,alpha, iters, batch_size,lamb):
    cost_history = np.zeros(iters)

    w= np.zeros(((X.shape[1]),1))

    for i in range(iters):

        rand_index = np.random.randint(len(y)-batch_size)
        ind_x = X[rand_index:rand_index+batch_size]
        ind_y = y[rand_index:rand_index+batch_size]
        w = w - (alpha/batch_size) * (ind_x.T.dot(ind_x.dot(w) - ind_y) - (lamb/2)*np.sign
        cost_history[i] = cost_function_l1(ind_x,ind_y,w,lamb)
    return w, cost_history
```

Defining MSE, MAE AND CC FUNCTIONS

```
def mean_abs_error(Ypred,Yact):
    sum_error=abs(Yact - Ypred)
    ma_error=sum(sum_error)/Ypred.shape[0]
    return ma_error


def mean_square_error(Ypred,Yact):
  for i in range(Ypred.shape[0]):
    sum_error=(Yact - Ypred)**2
  ms_error=sum(sum_error)/Ypred.shape[0]
  return ms_error


def correcoff(Ypred,Yact):
    ypm=np.mean(Ypred)##mean of Ypred data
    yam=np.mean(Yact)##mean of Yactual data

    sum_numerator=((Yact - yam)*(Ypred-ypm))

    sum_d1=((Yact - yam)**2)
```

```
    sum_d2=((Ypred - ypm)**2)
    sum_d1=pow(sum(sum_d1),1/2)
    sum_d2=pow(sum(sum_d2),1/2)
    th_error=sum(sum_numerator)/(sum_d1*sum_d2)
    return th_error
```

LOADING THE DATA

```
data_X = pd.read_excel('/content/drive/MyDrive/nnfl data/Q1_Q2/xtr.xlsx',header=None)
data_Y = pd.read_excel('/content/drive/MyDrive/nnfl data/Q1_Q2/ytr.xlsx',header=None)
data_Xte = pd.read_excel('/content/drive/MyDrive/nnfl data/Q1_Q2/xte.xlsx',header=None)
data_Yte = pd.read_excel('/content/drive/MyDrive/nnfl data/Q1_Q2/yte.xlsx',header=None)
Xte=data_Xte.values
Yte=data_Yte.values
```

NORMALISING THE TRAINING DATA

```
datan_x=data_X.values
X=datan_x
m=X.shape[0]
xmin=np.min(X,axis=0)
xmax=np.max(X,axis=0)
X= (X-xmin)/(xmax-xmin)
```

ADDING BIAS VECTOR

```
pp=np.ones([m,1])
X=np.append(pp,X,axis=1)
```

```
X_te = np.hstack((np.ones((Xte.shape[0],1)) , Xte))
```

NORMALISING THE TRAINING OUTPUT

```
datan_y=data_Y.values
y=datan_y

ymin = np.min(y, axis = 0)
ymax = np.max(y, axis = 0)
y = (y- ymin)/(ymax-ymin)
print(y.shape)
```

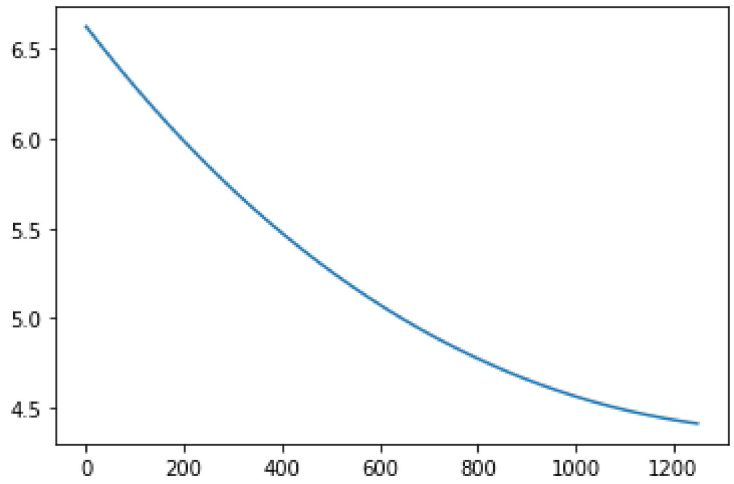
```
(55, 1)
```

INITIALISING THE WEIGHT VECTOR

```
w= np.zeros((X.shape[1]))
w1=np.zeros((X.shape[1]))###weight initialization
```

IMPLEMENTING BATCH GRADIENT DESCENT WITH LR

```
alpha=0.00025 ##learning rate
iters=1250 ###iterations
lamb=5
batch_w,J_his = batch_gradient_descent(X,y,w,alpha,iters)

plt.plot(range(iters),J_his)
plt.show()

bgd=batch_w[-1:]
print("WEIGHT VECTOR",bgd)

y_pred_bgd=X_te.dot(bgd.T)
a=mean_abs_error(y_pred_bgd,Yte)
print("MEAN ABSOLUTE ERROR",a)

b=mean_square_error(y_pred_bgd,Yte)
print("MEAN SQUARE ERROR:",b)

c=correcoff(y_pred_bgd,Yte)
print("CORRELATION COEFF:",c)
```

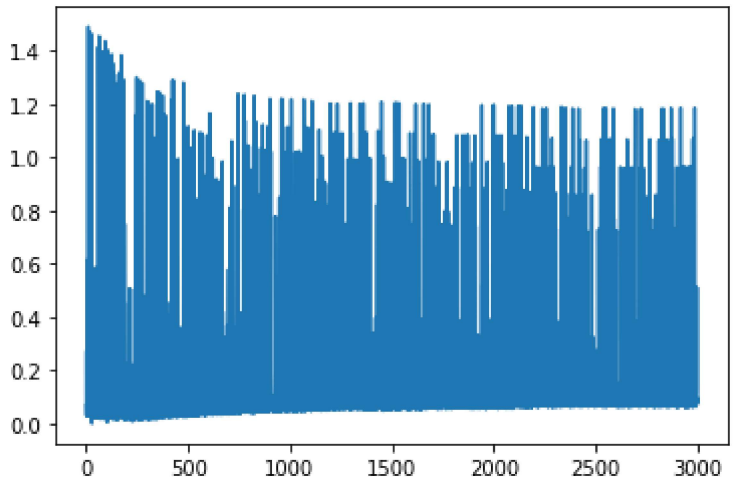



```
WEIGHT VECTOR [[0.18191255 0.0932603  0.10508698]]
MEAN ABSOLUTE ERROR [1.30998239]
MEAN SQUARE ERROR: [1.73572317]
CORRELATION COEFF: [-0.00825793]
```

IMPLEMENTING STOCHASTIC GRAIDENT DESCENT WITH RR[L2]

```
alpha=0.0008
iters=3000 ###iterations
lamb=1
w_n_l2,J_sgd_l2 = stochastic_gradient_descent_l2(X,y,w,alpha, iters,lamb)
```

```
plt.plot(range(iters),J_sgd_l2)
plt.show()

sgd_l2=w_n_l2[-1:]
print("WEIGHT VECTOR:",sgd_l2)

y_pred_sgd_l2=X_te.dot(sgd_l2.T)

a=mean_abs_error(y_pred_sgd_l2,Yte)
print("MEAN ABSOLUTE ERROR",a)

b=mean_square_error(y_pred_sgd_l2,Yte)
print("MEAN SQUARE ERROR:",b)

c=correcoff(y_pred_sgd_l2,Yte)
print("CORRELATION COEFF:",c)
```

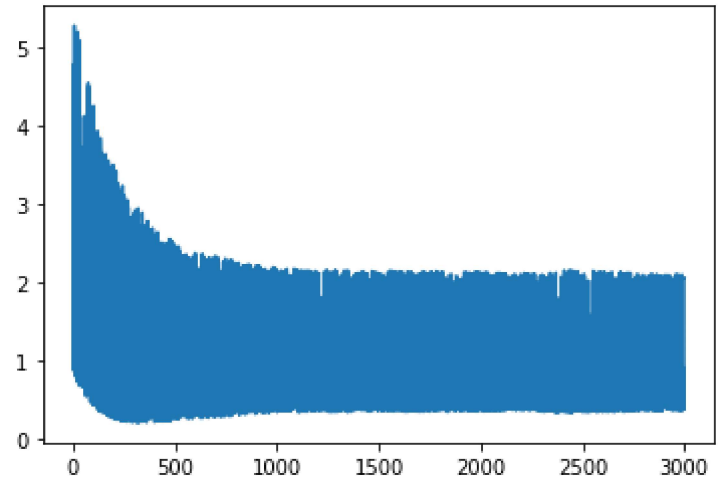



```
WEIGHT VECTOR: [[0.09020741 0.09020741 0.09020741]]
MEAN ABSOLUTE ERROR [1.03753298]
MEAN SQUARE ERROR: [1.09290369]
CORRELATION COEFF: [-0.00880095]
```

IMPLEMENTING MINI BATCH GRADIENT DESCENT WITH LAR[L1]

```
alpha=0.002
iters=3000 ###iterations
lamb=0.2
batch_size=30
mb_w_l1,J_mb_l1 = MB_gradient_descent_l1(X,y,w1,alpha, iters, batch_size,lamb)

plt.plot(range(iters),J_mb_l1)
plt.show()
print("WEIGHT VECTOR:",mb_w_l1)
y_pred_mbg_l1=X_te.dot(mb_w_l1)

a=mean_abs_error(y_pred_mbg_l1,Yte)
print("MEAN ABSOLUTE ERROR",a)

b=mean_square_error(y_pred_mbg_l1,Yte)
print("MEAN SQUARE ERROR:",b)
```

```
c=correcoff(y_pred_mbg_l1,Yte)
print("CORRELATION COEFF:",c)
```




```
WEIGHT VECTOR: [[0.23586686]
 [0.06197959]
 [0.10418553]]
MEAN ABSOLUTE ERROR [0.8123559]
MEAN SQUARE ERROR: [0.674275]
CORRELATION COEFF: [-0.00650422]
```

Q2

Q2

MOUNTING THE DRIVE

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

importing libraries

```
import pandas as pd
import math
import numpy as np
import matplotlib.pyplot as plt
import sklearn
```

Defining Cost Functions

```
def cost_function_l2(X,y,w,lamb):

    hypothesis = np.dot(X,w.T)

    J= (1/(2*len(y)))*np.sum((hypothesis-y)**2) + (lamb/2)*np.sum(w**2)
    return J


def cost_function_l1(X,y,w,lamb):
    w=w.ravel()
    hypothesis = np.dot(X,np.transpose(w))

    J= (1/(2*len(y)))*np.sum((hypothesis-y)**2) + (lamb/2)*np.sum(abs(w))
    return J


def cost_function(X,y,w):

    hypothesis = np.dot(X,w.T)
    J = (1/(2*len(y)))*np.sum((hypothesis-y)**2)

    return J
```

Defining gradient descent functions

```
def batch_gradient_descent(X,y,w,alpha,iters):
    cost_history = np.zeros(iters)
    for i in range(iters):
        hypothesis = np.dot(X,w.T)
        w = w - (alpha/len(y)) * np.dot(hypothesis - y, X)
        cost_history[i] = cost_function(X,y,w)
    return w, cost_history


def stochastic_gradient_descent_l2(X,y,w,alpha, iters,lamb):
    cost_history = np.zeros(iters)
    for i in range(iters):
        rand_index = np.random.randint(len(y)-1)
        ind_x = X[rand_index:rand_index+1]
        ind_y = y[rand_index:rand_index+1]
        w = w*(1-alpha*lamb) - alpha * (ind_x.T.dot(ind_x.dot(w) - ind_y))
        cost_history[i] = cost_function_l2(ind_x,ind_y,w,lamb)
    return w, cost_history


def MB_gradient_descent_l1(X,y,w,alpha, iters, batch_size,lamb):
    cost_history = np.zeros(iters)

    w= np.zeros(((X.shape[1]),1))

    for i in range(iters):

        rand_index = np.random.randint(len(y)-batch_size)
        ind_x = X[rand_index:rand_index+batch_size]
        ind_y = y[rand_index:rand_index+batch_size]
        w = w - (alpha/batch_size) * (ind_x.T.dot(ind_x.dot(w) - ind_y) - (lamb/2)*np.sign
        cost_history[i] = cost_function_l1(ind_x,ind_y,w,lamb)
    return w, cost_history
```

Defining MSE, MAE AND CC FUNCTIONS

```
def mean_abs_error(Ypred,Yact):
    sum_error=abs(Yact - Ypred)
    ma_error=sum(sum_error)/Ypred.shape[0]
    return ma_error


def mean_square_error(Ypred,Yact):
  for i in range(Ypred.shape[0]):
    sum_error=(Yact - Ypred)**2
  ms_error=sum(sum_error)/Ypred.shape[0]
  return ms_error


def correcoff(Ypred,Yact):
    ypm=np.mean(Ypred)##mean of Ypred data
    yam=np.mean(Yact)##mean of Yactual data

    sum_numerator=((Yact - yam)*(Ypred-ypm))
```

```
    sum_d1=((Yact - yam)**2)
    sum_d2=((Ypred - ypm)**2)
    sum_d1=pow(sum(sum_d1),1/2)
    sum_d2=pow(sum(sum_d2),1/2)
    th_error=sum(sum_numerator)/(sum_d1*sum_d2)
    return th_error
```

LOADING THE DATA

```
data_X = pd.read_excel('/content/drive/MyDrive/nnfl data/Q1_Q2/xtr.xlsx',header=None)
data_Y = pd.read_excel('/content/drive/MyDrive/nnfl data/Q1_Q2/ytr.xlsx',header=None)
data_Xte = pd.read_excel('/content/drive/MyDrive/nnfl data/Q1_Q2/xte.xlsx',header=None)
data_Yte = pd.read_excel('/content/drive/MyDrive/nnfl data/Q1_Q2/yte.xlsx',header=None)
Xte=data_Xte.values
Xte1=Xte[:,0].reshape(Xte.shape[0],1)
Xte2=Xte[:,1].reshape(Xte.shape[0],1)
k=Xte.shape[0]
Yte=data_Yte.values
Yte1=Yte[:,0].reshape(Yte.shape[0],1)
```

NORMALISING THE TRAINING DATA

```
datan_x=data_X.values
X1=datan_x[:,0].reshape(datan_x.shape[0],1)
X2=datan_x[:,1].reshape(datan_x.shape[0],1)
m=X1.shape[0]

x1min=np.min(X1,axis=0)
x1max=np.max(X1,axis=0)
X1= (X1-x1min)/(x1max-x1min)

n=X2.shape[0]

x2min=np.min(X2,axis=0)
x2max=np.max(X2,axis=0)
X2= (X2-x2min)/(x2max-x2min)
```

MAKING THE FEATURE MAPPING FOR PR

```
Z=np.ones([m,1])
Z=np.append(Z,X1,axis=1)

Z=np.append(Z,X2,axis=1)

Z=np.append(Z,X1**2,axis=1)
Z=np.append(Z,X2**2,axis=1)
Z=np.append(Z,X1*X2,axis=1)
```

```
Zte=np.ones([k,1])
Zte=np.append(Zte,Xte1,axis=1)

Zte=np.append(Zte,Xte2,axis=1)

Zte=np.append(Zte,Xte1**2,axis=1)
Zte=np.append(Zte,Xte2**2,axis=1)
Zte=np.append(Zte,Xte1*Xte2,axis=1)
```

Double-click (or enter) to edit

NORMALISING THE TRAINING OUTPUT

```
datan_y=data_Y.values
y=datan_y

ymin = np.min(y, axis = 0)
ymax = np.max(y, axis = 0)
y = (y- ymin)/(ymax-ymin)
print(y.shape)
```

```
(55, 1)
```

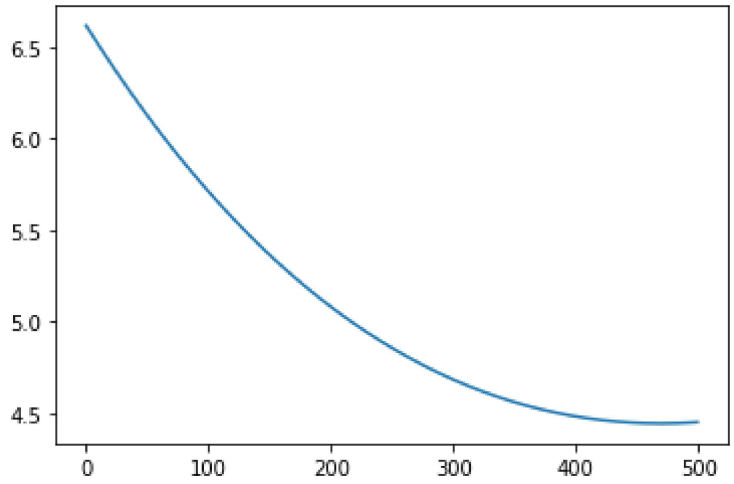
INITIALISING THE WEIGHT VECTOR

```
w= np.zeros((Z.shape[1]))
w1=np.zeros((Z.shape[1]))###weight initialization
```

IMPLEMENTING BATCH GRADIENT DESCENT WITH PR

```
alpha=0.0006 ##learning rate
iters=500 ###iterations
lamb=1
batch_w,J_his = batch_gradient_descent(Z,y,w,alpha,iters)

plt.plot(range(iters),J_his)
plt.show()

bgd=batch_w[-1:]
print("WEIGHT VECTOR",bgd)
```

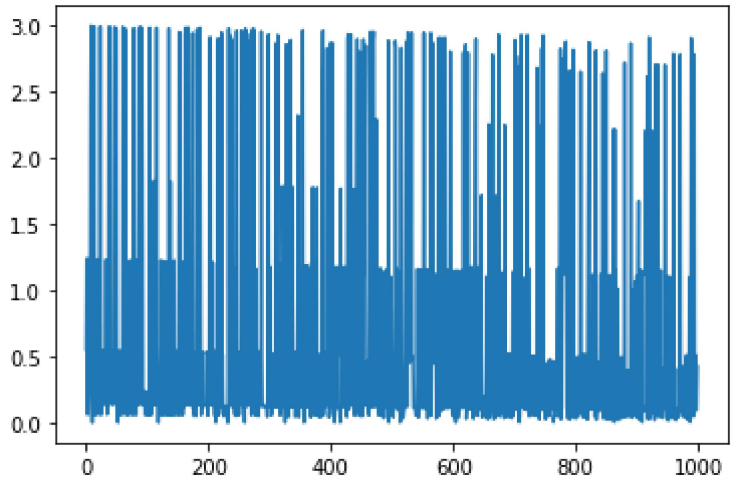
```
y_pred_bgd=Zte.dot(bgd.T)
a=mean_abs_error(y_pred_bgd,Yte)
print("MEAN ABSOLUTE ERROR",a)

b=mean_square_error(y_pred_bgd,Yte)
print("MEAN SQUARE ERROR:",b)

c=correcoff(y_pred_bgd,Yte)
print("CORRELATION COEFF:",c)
```




```
WEIGHT VECTOR [[0.17557825 0.09014305 0.10155515 0.06084765 0.07114284 0.06120531]]
MEAN ABSOLUTE ERROR [34.06934365]
MEAN SQUARE ERROR: [1172.82569505]
CORRELATION COEFF: [-0.00907119]
```

IMPLEMENTING STOCHASTIC GRAIDENT DESCENT WITH PR WITH L2 NORM REGULARISATION

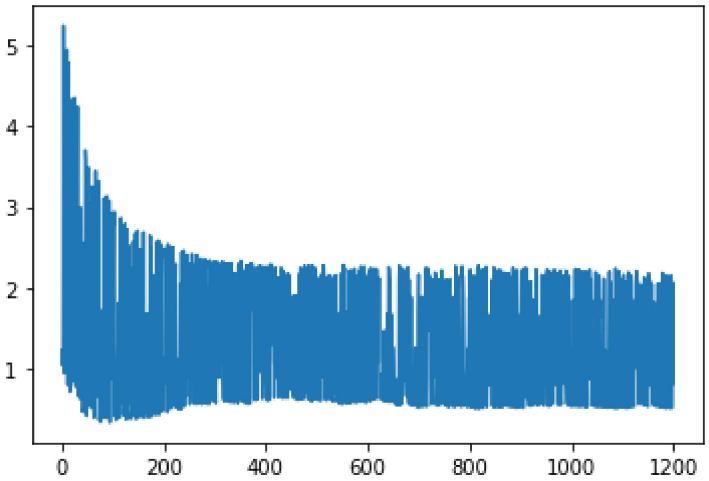
```
alpha=0.00006
iters=1000 ###iterations
lamb=0.05
w_n_l2,J_sgd_l2 = stochastic_gradient_descent_l2(Z,y,w,alpha, iters,lamb)

plt.plot(range(iters),J_sgd_l2)
plt.show()

sgd_l2=w_n_l2[-1:]

print("WEIGHT VECTOR:",sgd_l2)

y_pred_sgd_l2=Zte.dot(sgd_l2.T)

a=mean_abs_error(y_pred_sgd_l2,Yte)
print("MEAN ABSOLUTE ERROR",a)

b=mean_square_error(y_pred_sgd_l2,Yte)
print("MEAN SQUARE ERROR:",b)

c=correcoff(y_pred_sgd_l2,Yte)
print("CORRELATION COEFF:",c)
```




```
WEIGHT VECTOR: [[0.00834931 0.00834931 0.00834931 0.00834931 0.00834931 0.00834931]]
MEAN ABSOLUTE ERROR [3.22569445]
MEAN SQUARE ERROR: [10.61160124]
CORRELATION COEFF: [-0.00944736]
```

IMPLEMENTING MINI BATCH GRADIENT DESCENT WITH PR WITH L1 NORM REGULARISATION

```
alpha=0.005
iters=1200 ###iterations
lamb=0.5
batch_size=35
mb_w_l1,J_mb_l1 = MB_gradient_descent_l1(Z,y,w1,alpha, iters, batch_size,lamb)

plt.plot(range(iters),J_mb_l1)
plt.show()

print("WEIGHT VECTOR:",mb_w_l1)
y_pred_mbg_l1=Zte.dot(mb_w_l1)

a=mean_abs_error(y_pred_mbg_l1,Yte)
print("MEAN ABSOLUTE ERROR",a)

b=mean_square_error(y_pred_mbg_l1,Yte)
print("MEAN SQUARE ERROR:",b)

c=correcoff(y_pred_mbg_l1,Yte)
print("CORRELATION COEFF:",c)
```




```
WEIGHT VECTOR: [[ 0.24162462]
 [ 0.0483339 ]
 [ 0.09309818]
 [-0.00305662]
 [ 0.02442236]
 [ 0.00455333]]
MEAN ABSOLUTE ERROR [2.05192643]
MEAN SQUARE ERROR: [4.37638189]
CORRELATION COEFF: [0.00053929]
```

Q3

Q3

MOUNTING THE DRIVE

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

importing libraries

```
import pandas as pd
import math
import numpy as np
import matplotlib.pyplot as plt
import sklearn
np.random.seed(0)
from sklearn.metrics import confusion_matrix
```

LOADING THE DATA

```
data = pd.read_excel('/content/drive/MyDrive/nnfl data/Xtr_Q3.xlsx',header=None) ###loadin
```

```
data1 = pd.read_excel('/content/drive/MyDrive/nnfl data/Ytr_Q3.xlsx',header=None) ###loadi
```

```
data_Xte = pd.read_excel('/content/drive/MyDrive/nnfl data/Xte_Q3.xlsx',header=None)
data_Yte = pd.read_excel('/content/drive/MyDrive/nnfl data/Yte_Q3.xlsx',header=None)
```

NORMALISING THE TESTING DATA

```
Xte=data_Xte.values
Yte=data_Yte.values

xminte = np.min(Xte, axis = 0)
xmaxte = np.max(Xte, axis = 0)
Xte = (Xte- xminte)/(xmaxte-xminte)

Yte=Yte-1
Yte=Yte.ravel()
```

```
n=Xte.shape[0]
qq = np.ones([n, 1])
Xte = np.append(qq,Xte, axis=1)

print(Xte.shape,Yte.shape)
```

```
     (119, 61) (119,)
```

```
m=data.shape[0]
datan=data.values
data1n=data1.values

y=data1n[:,0] # class label

X=datan # feature matrix
```

NORMALISING THE TRAINING DATA

```
xmin = np.min(X, axis = 0)
xmax = np.max(X, axis = 0)
X = (X- xmin)/(xmax-xmin)
```

ADDING BIAS VECTOR

```
pp = np.ones([m, 1]) # vector containg ones as all elements
X = np.append(pp,X, axis=1) #Column of ones
y=y-1
print(X.shape,y.shape)
```

```
     (1589, 61) (1589,)
```

DEFINING SIGMOID FUNCTION

```
def sigmoid(z):
    return 1.0/(1 + np.exp(-z)) ###activation function
```

DEFINING COST FUNCTIONS

```
def cost_function(X,y,w): ###define cost function
    hypothesis = sigmoid(np.dot(X,w.T)) ###calculation of hypothesis for all instances
    J =-(1/m)*(np.sum(y*(np.log(hypothesis)) + (1-y)*np.log(1-hypothesis)))
    return J


def cost_function_l1(X,y,w,lamb):
```

```
   ###define cost function
  hypothesis = sigmoid(np.dot(X,w.T)) ###calculation of hypothesis for all instances
  J =-(1/m)*(np.sum(y*(np.log(hypothesis)) + (1-y)*np.log(1-hypothesis))) + (lamb/2)*np.su
  return J

def cost_function_l2(X,y,w,lamb):

   ###define cost function
  hypothesis = sigmoid(np.dot(X,w.T)) ###calculation of hypothesis for all instances
  J =-(1/m)*(np.sum(y*(np.log(hypothesis)) + (1-y)*np.log(1-hypothesis))) + (lamb/2)*np.su
  return J
```

DEFINING GRADIENT DESCENT FUNCTIONS

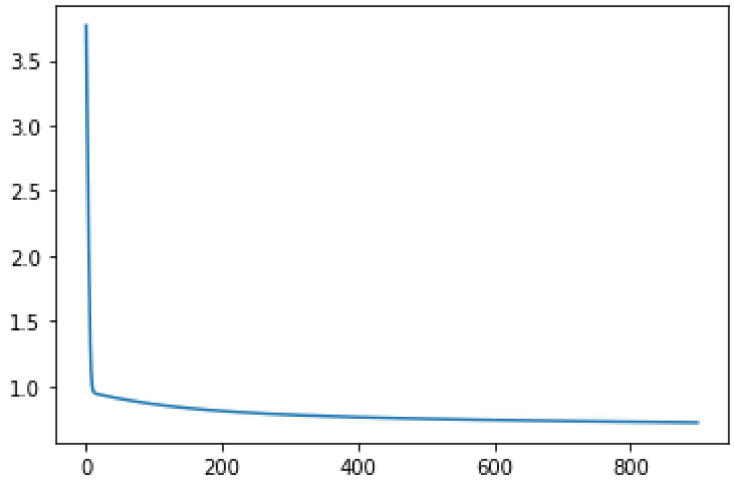
```
def batch_gradient_descent(X,y,w,alpha,iters):
    cost_history = np.zeros(iters) # cost function for each iteration
       #initalize our cost history list to store the cost function on every iteration
    for i in range(iters):

        hypothesis = sigmoid(np.dot(X,w.T))
        w = w - (alpha/len(y)) * np.dot(hypothesis - y,X)
        cost_history[i] = cost_function(X,y,w)
    return w,cost_history


def batch_gradient_descent_l1(X,y,w,alpha,iters,lamb):
  cost_history = np.zeros(iters)

 # cost function for each iteration
  #initalize our cost history list to store the cost function on every iteration
  for i in range(iters):
    hypothesis = sigmoid(np.dot(X,w.T))
    w = w - (alpha/len(y)) * np.dot(hypothesis - y, X) - ((alpha*lamb/2)*np.sign(w))
    cost_history[i] = cost_function_l1(X,y,w,lamb)
  return w,cost_history


def batch_gradient_descent_l2(X,y,w,alpha,iters,lamb):
  cost_history = np.zeros(iters)

 # cost function for each iteration
  #initalize our cost history list to store the cost function on every iteration
  for i in range(iters):
    hypothesis = sigmoid(np.dot(X,w.T))
    w = w*(1-alpha*lamb) - (alpha/len(y)) * np.dot(hypothesis - y, X)
    cost_history[i] = cost_function_l1(X,y,w,lamb)
  return w,cost_history
```

INITIALISING THE WEIGHT VECTOR

```
w = np.random.randn(1,X.shape[1]).ravel()
w1 = np.random.randn(1,X.shape[1]).ravel()
w2 = np.zeros((1,X.shape[1])).ravel()
```

IMPLEMENTING LOR WITH BATCH GRADIENT DESCENT

```
alpha=0.12 ##learning rate
iters=900 ###iterations
X_train=X
y_train=y
batch_w,J_his = batch_gradient_descent(X_train,y_train,w1,alpha,iters)
plt.plot(range(iters),J_his)
# print(J_his)
plt.show()
```




```
z_bgd = np.dot(Xte, batch_w.T)
h_bgd = sigmoid(z_bgd)
y_pred_bgd=h_bgd
y_pred_bgd=h_bgd>0.5
y_pred_bgd=y_pred_bgd.astype(int)
```

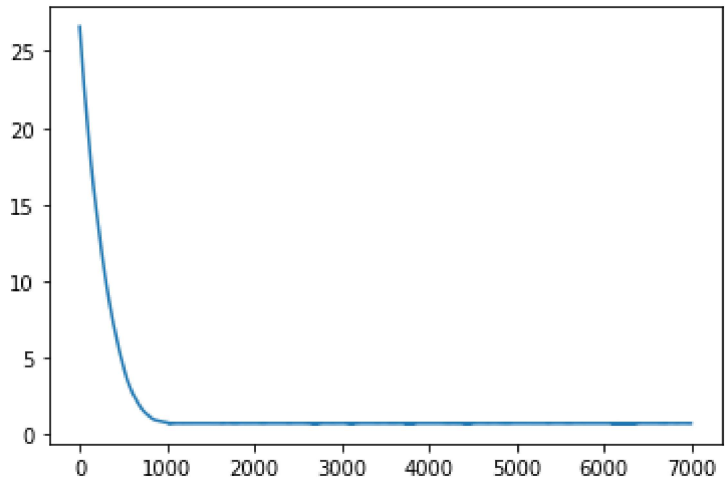
```
cm=confusion_matrix(Yte, y_pred_bgd)
print(cm)

tp = cm[1][1]
tn = cm[0][0]
fp = cm[0][1]
fn = cm[1][0]

Acc = (tp+tn)/(tp+tn+fp+fn)
SE = tp/(tp+fn)
SP = tn/(tn+fp)

print('Accuracy : ' + str(Acc))
```

```
print('sensitivity : ' + str(SE))
print('specificity : ' + str(SP))
```

```
[[56  2]
 [58  3]]
Accuracy : 0.4957983193277311
sensitivity : 0.04918032786885246
specificity : 0.9655172413793104
```

IMPLEMENTING LOR WITH BATCH GRADIENT DESCENT WITH L1 NORM REGULARISATION

```
alpha=0.005 ##learning rate
iters=7000 ###iterations
lamb=1
X_train=X
y_train=y
batch_w_l1,J_his_l1 = batch_gradient_descent_l1(X_train,y_train,w,alpha,iters,lamb)
plt.plot(range(iters),J_his_l1)
plt.show()
```




```
z_bgd_l1 = np.dot(Xte, batch_w_l1.T)
h_bgd_l1 = sigmoid(z_bgd_l1)
y_pred_bgd_l1=h_bgd_l1>0.5
y_pred_bgd_l1=y_pred_bgd_l1.astype(int)
```

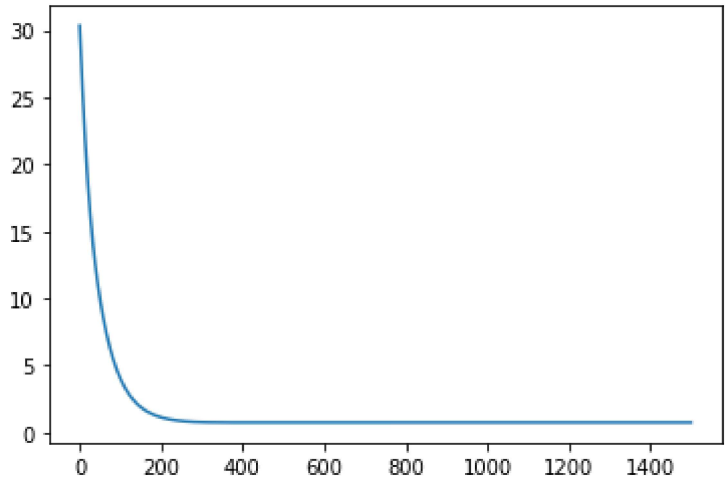
```
cm=confusion_matrix(Yte, y_pred_bgd_l1)
print(cm)

tp = cm[1][1]
tn = cm[0][0]
fp = cm[0][1]
fn = cm[1][0]

Acc = (tp+tn)/(tp+tn+fp+fn)
SE = tp/(tp+fn)
SP = tn/(tn+fp)
```

```
print('Accuracy : ' + str(Acc))
print('sensitivity : ' + str(SE))
print('specificity : ' + str(SP))
```

```
[[17 41]
 [15 46]]
Accuracy : 0.5294117647058824
sensitivity : 0.7540983606557377
specificity : 0.29310344827586204
```

IMPLEMENTING LOR WITH BATCH GRADIENT DESCENT WITH L2 NORM REGULARISATION

```
alpha=0.02 ##learning rate
iters=1500 ###iterations
lamb=1
X_train=X
y_train=y
batch_w_l2,J_his_l2 = batch_gradient_descent_l2(X_train,y_train,w1,alpha,iters,lamb)
plt.plot(range(iters),J_his_l2)
plt.show()
```




```
z_bgd_l2 = np.dot(Xte, batch_w_l2.T)
h_bgd_l2 = sigmoid(z_bgd_l2)
y_pred_bgd_l2=h_bgd_l2>0.5
y_pred_bgd_l2=y_pred_bgd_l2.astype(int)
```

```
cm=confusion_matrix(Yte, y_pred_bgd_l2)
print(cm)

tp = cm[1][1]
tn = cm[0][0]
fp = cm[0][1]
fn = cm[1][0]

Acc = (tp+tn)/(tp+tn+fp+fn)
```

```
SE = tp/(tp+fn)
SP = tn/(tn+fp)

print('Accuracy : ' + str(Acc))
print('sensitivity : ' + str(SE))
print('specificity : ' + str(SP))
```

```
[[32 26]
 [37 24]]
Accuracy : 0.47058823529411764
sensitivity : 0.39344262295081966
specificity : 0.5517241379310345
```

Q4 ONE VS ALL

Q4 ONE VS ALL

MOUNTING THE DRIVE

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

importing libraries

```
import pandas as pd
import math
import numpy as np
import matplotlib.pyplot as plt
import sklearn
np.random.seed(0)
```

```
from sklearn.model_selection import train_test_split
```

DEFINING SIGMOID FUNCTION

```
def sigmoid(x):
  x = x.astype(float)
  z = np.exp(-x)
  sig = 1 / (1 + z)
  return sig
```

DEFINING SET FUNCTION

```
def set(y):
    for i in range(len(y)):
        if(y[i]>=0.5):
            y[i] = 1
        if(y[i]<0.5):
            y[i] = 0
    return y
```

DEFINING GRADIENT DESCENT FUNCTIONS

```
def batch_gradient_descent(X,y,w,alpha,iters):

  cost_history = np.zeros(iters) # cost function for each iteration

  #initalize our cost history list to store the cost function on every iteration

  for i in range(iters):
    z = np.dot(X,w.T)
    hypothesis = sigmoid(z)
    w = w -  alpha * np.dot((hypothesis-y), X) #weight updation

  return w


def batch_gradient_descent_l1(X,y,w,alpha,iters,lamb):

  cost_history = np.zeros(iters) # cost function for each iteration

  #initalize our cost history list to store the cost function on every iteration

  for i in range(iters):
    z = np.dot(X,w.T)
    hypothesis = sigmoid(z)
    w = w -  alpha * np.dot((hypothesis-y), X) - ((alpha*lamb/2)*np.sign(w))#weight updati

  return w


def batch_gradient_descent_l2(X,y,w,alpha,iters,lamb):

  cost_history = np.zeros(iters) # cost function for each iteration

  #initalize our cost history list to store the cost function on every iteration

  for i in range(iters):
    z = np.dot(X,w.T)
    hypothesis = sigmoid(z)
    w = w*(1-alpha*lamb) -  alpha * np.dot((hypothesis-y), X) #weight updation

  return w
```

LOADING THE DATA

```
data = pd.read_excel('/content/drive/MyDrive/nnfl data/data.xlsx',header=None)


datan=data.values
X=datan[:,0:datan.shape[1]-1]
y=datan[:,datan.shape[1]-1]
print(data.shape)
```

```
(3412, 61)
```

SPLITTING THE DATA INTO TRAINING, VALIDATION AND TESTING using holdout validation

```
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2, shuffle = True, ran
X_train, X_val, y_train, y_val = train_test_split(X_train,y_train,test_size=0.125, shuffle
```

NOMRALISING TESTING AND TRAINING DATA, ADDING BIAS VECTOR

```
m=X_train.shape[0]
n=X_test.shape[0]

qq = np.ones([n, 1])

xmin1 = np.min(X_test, axis = 0)
xmax1 = np.max(X_test, axis = 0)

X_test = (X_test- xmin1)/(xmax1-xmin1)

X_test = np.append(qq,X_test, axis=1)
print(X_test.shape)
```

```
(683, 61)
```

```
xmin = np.min(X_train, axis = 0)
xmax = np.max(X_train, axis = 0)

X_train = (X_train- xmin)/(xmax-xmin)
print(X_train)
```

```
[[0.39328586 0.16654807 0.16214643 ... 0.23299547 0.81403191 0.93200908]
 [0.11627088 0.11346816 0.21845213 ... 0.15076686 0.67705128 0.63106166]
 [0.51419012 0.52397105 0.22901312 ... 0.46541911 0.83192827 0.79023879]
 ...
 [0.63425256 0.53253442 0.3768769  ... 0.64252927 0.71068596 0.75504019]
 [0.29051515 0.30674267 0.25506676 ... 0.3681679  0.877832   0.63315738]
 [0.56033294 0.29998019 0.37576393 ... 0.3886692  0.76985938 0.700262  ]]
```

```
pp = np.ones([m, 1])
X_train = np.append(pp,X_train, axis=1) #Column of ones

print(X_train)
```

```
[[1.         0.39328586 0.16654807 ... 0.23299547 0.81403191 0.93200908]
 [1.         0.11627088 0.11346816 ... 0.15076686 0.67705128 0.63106166]
 [1.         0.51419012 0.52397105 ... 0.46541911 0.83192827 0.79023879]
 ...
 [1.         0.63425256 0.53253442 ... 0.64252927 0.71068596 0.75504019]
 [1.         0.29051515 0.30674267 ... 0.3681679  0.877832   0.63315738]
 [1.         0.56033294 0.29998019 ... 0.3886692  0.76985938 0.700262  ]]
```

MAKING TRAINING MODELS FOR OVE VS ALL 4 CASES

```
y1_tr = [1 for i in range(len(y_train))]
y2_tr = [1 for i in range(len(y_train))]
y3_tr = [1 for i in range(len(y_train))]
y4_tr = [1 for i in range(len(y_train))]
for i in range(len(y_train)):
    if(y_train[i] != 1):
        y1_tr[i] = 0
    if(y_train[i] != 2):
        y2_tr[i] = 0
    if(y_train[i] != 3):
        y3_tr[i] = 0
    if(y_train[i] != 4):
        y4_tr[i] = 0
print(y4_tr)
```

```
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0,
```

IMPLEMETING ONE VS ALL FOR BATCH GRADIENT DESCENT MULTICLASS LOR

```
alpha=0.0025
iters=1000

w= np.zeros((X_train.shape[1])) ###weight initialization
w_m1 = batch_gradient_descent(X_train,y1_tr,w,alpha,iters)
y_p1 = np.dot(X_test,w_m1.T)
y_p1 = set(y_p1)

w= np.zeros((X_train.shape[1])) ###weight initialization
w_m2 = batch_gradient_descent(X_train,y2_tr,w,alpha,iters)
y_p2 = np.dot(X_test,w_m2.T)
y_p2 = set(y_p2)

w= np.zeros((X_train.shape[1])) ###weight initialization
w_m3 = batch_gradient_descent(X_train,y3_tr,w,alpha,iters)
y_p3 = np.dot(X_test,w_m3.T)
y_p3 = set(y_p3)

w= np.zeros((X_train.shape[1])) ###weight initialization
w_m4 = batch_gradient_descent(X_train,y4_tr,w,alpha,iters)
y_p4 = np.dot(X_test,w_m4.T)
y_p4 = set(y_p4)
```

```
cval = [0 for i in range(len(y_test))]
```

```
for i in range(len(y_test)):
    if (y_p1[i] == 1):
        cval[i] = 1.0
    if (y_p2[i] == 1):
        cval[i] = 2.0
    if (y_p3[i] == 1):
        cval[i] = 3.0
    if (y_p4[i] == 1):
        cval[i] = 4.0
```

GENERATING THE CONFUSION MATRIX

```
from sklearn.metrics import confusion_matrix
cm=confusion_matrix(y_test, cval)

print(cm)
```

```
[[  8   2 143   8]
 [  6   0 136  25]
 [  6   0 158  15]
 [  9   3 149  15]]
```

FINDING INDIVIDUAL CLASS ACCURACY AND OVERALL ACCURACY

```
confmat = np.asarray(cm)
Acc = (confmat[0][0] + confmat[1][1] + confmat[2][2])/sum(sum(confmat))
Acc1 = confmat[0][0]/sum(confmat[0])
Acc2 = confmat[1][1]/sum(confmat[1])
Acc3 = confmat[2][2]/sum(confmat[2])
Acc4 = confmat[3][3]/sum(confmat[3])
print('Overall Accuracy : ' + str(Acc))
print('Accuracy of class 1 : ' + str(Acc1))
print('Accuracy of class 2 : ' + str(Acc2))
print('Accuracy of class 3 : ' + str(Acc3))
print('Accuracy of class 4 : ' + str(Acc4))
```

```
Overall Accuracy : 0.2430453879941435
Accuracy of class 1 : 0.049689440993788817
Accuracy of class 2 : 0.0
Accuracy of class 3 : 0.88268156424581
Accuracy of class 4 : 0.08522727272727272
```

IMPLEMETING ONE VS ALL FOR BATCH GRADIENT DESCENT MULTICLASS LOR WITH L1

```
alpha=0.0025
iters=1000
lamb=1
w= np.zeros((X_train.shape[1])) ###weight initialization
w_m1 = batch_gradient_descent_l1(X_train,y1_tr,w,alpha,iters,lamb)
```

```
y_p1 = np.dot(X_test,w_m1.T)
y_p1 = set(y_p1)

w= np.zeros((X_train.shape[1])) ###weight initialization
w_m2 = batch_gradient_descent_l1(X_train,y2_tr,w,alpha,iters,lamb)
y_p2 = np.dot(X_test,w_m2.T)
y_p2 = set(y_p2)

w= np.zeros((X_train.shape[1])) ###weight initialization
w_m3 = batch_gradient_descent_l1(X_train,y3_tr,w,alpha,iters,lamb)
y_p3 = np.dot(X_test,w_m3.T)
y_p3 = set(y_p3)

w= np.zeros((X_train.shape[1])) ###weight initialization
w_m4 = batch_gradient_descent_l1(X_train,y4_tr,w,alpha,iters,lamb)
y_p4 = np.dot(X_test,w_m4.T)
y_p4 = set(y_p4)
```

```
cval = [0 for i in range(len(y_test))]
for i in range(len(y_test)):
    if (y_p1[i] == 1):
        cval[i] = 1.0
    if (y_p2[i] == 1):
        cval[i] = 2.0
    if (y_p3[i] == 1):
        cval[i] = 3.0
    if (y_p4[i] == 1):
        cval[i] = 4.0
```

GENERATING THE CONFUSION MATRIX

```
from sklearn.metrics import confusion_matrix
cm=confusion_matrix(y_test, cval)

print(cm)
```

```
     [[  9   2 144   6]
      [  6   0 137  24]
      [  6   1 157  15]
      [  8   3 153  12]]
```

FINDING INDIVIDUAL CLASS ACCURACY AND OVERALL ACCURACY

```
confmat = np.asarray(cm)
Acc = (confmat[0][0] + confmat[1][1] + confmat[2][2])/sum(sum(confmat))
Acc1 = confmat[0][0]/sum(confmat[0])
Acc2 = confmat[1][1]/sum(confmat[1])
Acc3 = confmat[2][2]/sum(confmat[2])
Acc4 = confmat[3][3]/sum(confmat[3])
```

```
print('Overall Accuracy : ' + str(Acc))
print('Accuracy of class 1 : ' + str(Acc1))
print('Accuracy of class 2 : ' + str(Acc2))
print('Accuracy of class 3 : ' + str(Acc3))
print('Accuracy of class 4 : ' + str(Acc4))
```

```
Overall Accuracy : 0.2430453879941435
Accuracy of class 1 : 0.055900621118012424
Accuracy of class 2 : 0.0
Accuracy of class 3 : 0.8770949720670391
Accuracy of class 4 : 0.06818181818181818
```

IMPLEMETING ONE VS ALL FOR BATCH GRADIENT DESCENT MULTICLASS LOR WITH L1

```
alpha=0.0025
iters=1000
lamb=1

w= np.zeros((X_train.shape[1])) ###weight initialization
w_m1 = batch_gradient_descent_l2(X_train,y1_tr,w,alpha,iters,lamb)
y_p1 = np.dot(X_test,w_m1.T)
y_p1 = set(y_p1)

w= np.zeros((X_train.shape[1])) ###weight initialization
w_m2 = batch_gradient_descent_l2(X_train,y2_tr,w,alpha,iters,lamb)
y_p2 = np.dot(X_test,w_m2.T)
y_p2 = set(y_p2)

w= np.zeros((X_train.shape[1])) ###weight initialization
w_m3 = batch_gradient_descent_l2(X_train,y3_tr,w,alpha,iters,lamb)
y_p3 = np.dot(X_test,w_m3.T)
y_p3 = set(y_p3)

w= np.zeros((X_train.shape[1])) ###weight initialization
w_m4 = batch_gradient_descent_l2(X_train,y4_tr,w,alpha,iters,lamb)
y_p4 = np.dot(X_test,w_m4.T)
y_p4 = set(y_p4)



cval = [0 for i in range(len(y_test))]
for i in range(len(y_test)):
    if (y_p1[i] == 1):
        cval[i] = 1.0
    if (y_p2[i] == 1):
        cval[i] = 2.0
    if (y_p3[i] == 1):
        cval[i] = 3.0
    if (y_p4[i] == 1):
        cval[i] = 4.0
```

GENERATING THE CONFUSION MATRIX

```
from sklearn.metrics import confusion_matrix
cm=confusion_matrix(y_test, cval)

print(cm)
```

```
[[  6   0 153   2]
 [  3   0 153  11]
 [  4   0 170   5]
 [  5   0 161  10]]
```

FINDING INDIVIDUAL CLASS ACCURACY AND OVERALL ACCURACY

```
confmat = np.asarray(cm)
Acc = (confmat[0][0] + confmat[1][1] + confmat[2][2])/sum(sum(confmat))
Acc1 = confmat[0][0]/sum(confmat[0])
Acc2 = confmat[1][1]/sum(confmat[1])
Acc3 = confmat[2][2]/sum(confmat[2])
Acc4 = confmat[3][3]/sum(confmat[3])
print('Overall Accuracy : ' + str(Acc))
print('Accuracy of class 1 : ' + str(Acc1))
print('Accuracy of class 2 : ' + str(Acc2))
print('Accuracy of class 3 : ' + str(Acc3))
print('Accuracy of class 4 : ' + str(Acc4))
```

```
Overall Accuracy : 0.25768667642752563
Accuracy of class 1 : 0.037267080745341616
Accuracy of class 2 : 0.0
Accuracy of class 3 : 0.9497206703910615
Accuracy of class 4 : 0.056818181818181816
```

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.

Q4 ONE VS ONE

MOUNTING THE DRIVE

```
from google.colab import drive
drive.mount('/content/drive')
```

```
     Mounted at /content/drive
```

IMPORTING LIBRARIES

```
import pandas as pd
import math
import numpy as np
import matplotlib.pyplot as plt
import sklearn
np.random.seed(0)
import statistics
```

```
from sklearn.model_selection import train_test_split
```

DEFINING SIGMOID FUNCTION

```
def sigmoid(x):
  x = x.astype(float)
  z = np.exp(-x)
  sig = 1 / (1 + z)
  return sig
```

DEFINING SET FUNCTION

```
def set(y,k,j):
    for i in range(len(y)):
        if(y[i]>=0.5):
            y[i] = j
        if(y[i]<0.5):
            y[i] = k
    return y
```

DEFINING GRADIENT DESCENT FUNCTIONS

```
def batch_gradient_descent_l1(X,y,w,alpha,iters,lamb):
```

```
  cost_history = np.zeros(iters) # cost function for each iteration

  #initalize our cost history list to store the cost function on every iteration

  for i in range(iters):
    z = np.dot(X,w.T)
    hypothesis = sigmoid(z)
    w = w -  alpha * np.dot((hypothesis-y), X) - ((alpha*lamb/2)*np.sign(w))#weight updati

  return w

def batch_gradient_descent_l2(X,y,w,alpha,iters,lamb):

  cost_history = np.zeros(iters) # cost function for each iteration

  #initalize our cost history list to store the cost function on every iteration

  for i in range(iters):
    z = np.dot(X,w.T)
    hypothesis = sigmoid(z)
    w = w*(1-alpha*lamb) -  alpha * np.dot((hypothesis-y), X) #weight updation

  return w


def batch_gradient_descent(X,y,w,alpha,iters):

  cost_history = np.zeros(iters) # cost function for each iteration

  #initalize our cost history list to store the cost function on every iteration

  for i in range(iters):
    z = np.dot(X,w.T)
    hypothesis = sigmoid(z)
    w = w -  alpha * np.dot((hypothesis-y), X) #weight updation

  return w
```

LOADING THE DATA

```
data = pd.read_excel('/content/drive/MyDrive/nnfl data/data.xlsx',header=None)


datan=data.values
X=datan[:,0:datan.shape[1]-1]
y=datan[:,datan.shape[1]-1]
print(data.shape)
```

```
(3412, 61)
```

SPLITTING THE DATA INTO TRAINING, VALIDATION AND TESTING I.E. HOLDOUT VALIDATION

```
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2, shuffle = True, ran
X_train, X_val, y_train, y_val = train_test_split(X_train,y_train,test_size=0.125, shuffle
```

NORMALISING THE DATA AND ADDING BIAS VECTOR

```
m=X_train.shape[0]
n=X_test.shape[0]

qq = np.ones([n, 1])

xmin1 = np.min(X_test, axis = 0)
xmax1 = np.max(X_test, axis = 0)

X_test = (X_test- xmin1)/(xmax1-xmin1)

X_test = np.append(qq,X_test, axis=1)


xmin = np.min(X_train, axis = 0)
xmax = np.max(X_train, axis = 0)

X_train = (X_train- xmin)/(xmax-xmin)



pp = np.ones([m, 1])
X_train = np.append(pp,X_train, axis=1) #Column of ones

print(X_train)
```

```
[[1.         0.4565523  0.70925588 ... 0.52205576 0.84817498 0.86645165]
 [1.         0.34247347 0.52862044 ... 0.20327721 0.81558568 0.71125864]
 [1.         0.59779369 0.24443946 ... 0.6853999  0.82794782 0.64855454]
 ...
 [1.         0.74693149 0.34984461 ... 0.21043456 0.84001243 0.84605655]
 [1.         0.23788388 0.25948086 ... 0.43523629 0.71647965 0.62122422]
 [1.         0.64374366 0.5491812  ... 0.5910701  0.8530399  0.66566898]]
```

```
x_tr=X_train
y_tr=y_train


print(y_tr)
```

```
[3. 2. 3. ... 4. 2. 3.]
```

MAKING MODELS FOR DIFFERENT OVE VS ONE CASES

```
y1_tr = [] # class 1 vs class 2
y2_tr = [] # class 1 vs class 3
y3_tr = [] # class 1 vs class 4
y4_tr = [] # class 2 vs class 3
y5_tr = [] # class 2 vs class 4
```

```
y6_tr = [] # class 3 vs class 4
x1_tr = []
x2_tr = []
x3_tr = []
x4_tr = []
x5_tr = []
x6_tr = []
```

ADDING VALUES TO EACH ONE VS ONE CLASS LIST

```
for i in range(len(y_tr)):
    if(y_tr[i] != 3 and y_tr[i] != 4 ):
        y1_tr.append(y_tr[i])
        x1_tr.append(x_tr[i])
    if(y_tr[i] != 2 and y_tr[i] != 4):
        y2_tr.append(y_tr[i])
        x2_tr.append(x_tr[i])
    if(y_tr[i] != 2 and y_tr[i] != 3):
        y3_tr.append(y_tr[i])
        x3_tr.append(x_tr[i])
    if(y_tr[i] != 1 and y_tr[i] != 4):
        y4_tr.append(y_tr[i])
        x4_tr.append(x_tr[i])
    if(y_tr[i] != 1 and y_tr[i] != 3):
        y5_tr.append(y_tr[i])
        x5_tr.append(x_tr[i])
    if(y_tr[i] != 1 and y_tr[i] != 2):
        y6_tr.append(y_tr[i])
        x6_tr.append(x_tr[i])


x1_tr = np.asarray(x1_tr)
x2_tr = np.asarray(x2_tr)
x3_tr = np.asarray(x3_tr)
x4_tr = np.asarray(x4_tr)
x5_tr = np.asarray(x5_tr)
x6_tr = np.asarray(x6_tr)
```

DEFINING LOW AND HIGH FOR EACH CLASS CASE

```
for i in range(len(y1_tr)):
    if y1_tr[i] == 1:
        y1_tr[i] = 0
    else:
        y1_tr[i] = 1

for i in range(len(y2_tr)):
    if y2_tr[i] == 1:
        y2_tr[i] = 0
```

```
    else:
        y2_tr[i] = 1

for i in range(len(y3_tr)):
    if y3_tr[i] == 1:
        y3_tr[i] = 0
    else:
        y3_tr[i] = 1
for i in range(len(y4_tr)):
    if y4_tr[i] == 2:
        y4_tr[i] = 0
    else:
        y4_tr[i] = 1
for i in range(len(y5_tr)):
    if y5_tr[i] == 2:
        y5_tr[i] = 0
    else:
        y5_tr[i] = 1
for i in range(len(y6_tr)):
    if y6_tr[i] == 3:
        y6_tr[i] = 0
    else:
        y6_tr[i] = 1
```

IMPLEMENTING BATCH GRADIENT DESCENT FOR MULTICLASS ONE VS ONE

```
alpha=0.008
iters=1000
w1= np.zeros((x1_tr.shape[1])) ###weight initialization
w_m1 = batch_gradient_descent(x1_tr,y1_tr,w1,alpha,iters)
y_p1 = np.dot(X_test,w_m1.T)
y_p1 = set(y_p1,1,2)


w2= np.zeros((x2_tr.shape[1])) ###weight initialization
w_m2 = batch_gradient_descent(x2_tr,y2_tr,w2,alpha,iters)
y_p2 = np.dot(X_test,w_m2.T)
y_p2 = set(y_p2,1,3)


w3= np.zeros((x3_tr.shape[1])) ###weight initialization
w_m3 = batch_gradient_descent(x3_tr,y3_tr,w3,alpha,iters)
y_p3 = np.dot(X_test,w_m3.T)
y_p3 = set(y_p3,1,4)


w4= np.zeros((x4_tr.shape[1])) ###weight initialization
w_m4 = batch_gradient_descent(x4_tr,y4_tr,w4,alpha,iters)
y_p4 = np.dot(X_test,w_m4.T)
y_p4 = set(y_p4,2,3)


w5= np.zeros((x5_tr.shape[1])) ###weight initialization
```

```
w_m5 = batch_gradient_descent(x5_tr,y5_tr,w5,alpha,iters)
y_p5 = np.dot(X_test,w_m5.T)
y_p5 = set(y_p5,2,4)


w6= np.zeros((x6_tr.shape[1])) ###weight initialization
w_m6 = batch_gradient_descent(x6_tr,y6_tr,w6,alpha,iters)
y_p6 = np.dot(X_test,w_m6.T)
y_p6 = set(y_p6,3,4)
```

DEFINING MODE FUNCTION

```
from collections import Counter
def my_mode(sample):
  return Counter(sample).most_common(1)[0][0]
```

CALCULATING MODE OF PREDICTED OUTPUT

```
from statistics import mode

cval = [0 for i in range(len(y_test))]

for i in range(len(y_test)):

  k=[y_p1[i],y_p2[i],y_p3[i],y_p4[i],y_p5[i],y_p6[i]]

  cval[i]=my_mode(k)
```

GENERATING CONFUSION MATRIX

```
for i in range(len(cval)):
    if (cval[i] == 0):
        cval[i] = 'None'

print(cval)
y_actual = pd.Series(y_test, name='Actual')
y_pred = pd.Series(cval, name='Predicted')
confmat = pd.crosstab(y_actual, y_pred)
print(confmat)
```

```
[3.0, 3.0, 3.0, 2.0, 4.0, 2.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 1.0, 3.0, 3.0, 3.0,
Predicted  1.0  2.0  3.0  4.0
Actual
1.0          3   19  122    7
2.0          4   23  141    5
3.0          8   39  128   15
4.0          4   21  136    8
```

FINDING ACCURACY

```
confmat = np.asarray(confmat)
Acc = (confmat[0][0] + confmat[1][1] + confmat[2][2])/sum(sum(confmat))
Acc1 = confmat[0][0]/sum(confmat[0])
Acc2 = confmat[1][1]/sum(confmat[1])
Acc3 = confmat[2][2]/sum(confmat[2])
Acc4 = confmat[3][3]/sum(confmat[3])
print('Overall Accuracy : ' + str(Acc))
print('Accuracy of class 1 : ' + str(Acc1))
print('Accuracy of class 2 : ' + str(Acc2))
print('Accuracy of class 3 : ' + str(Acc3))
print('Accuracy of class 4 : ' + str(Acc4))
```

```
Overall Accuracy : 0.22547584187408493
Accuracy of class 1 : 0.019867549668874173
Accuracy of class 2 : 0.1329479768786127
Accuracy of class 3 : 0.6736842105263158
Accuracy of class 4 : 0.047337278106508875
```

IMPLEMENTING BATCH GRADIENT DESCENT FOR MULTICLASS ONE VS ONE WITH L1 NORM

```
alpha=0.008
iters=1000
lamb=1
w1= np.zeros((x1_tr.shape[1])) ###weight initialization
w_m1 = batch_gradient_descent_l1(x1_tr,y1_tr,w1,alpha,iters,lamb)
y_p1 = np.dot(X_test,w_m1.T)
y_p1 = set(y_p1,1,2)


w2= np.zeros((x2_tr.shape[1])) ###weight initialization
w_m2 = batch_gradient_descent_l1(x2_tr,y2_tr,w2,alpha,iters,lamb)
y_p2 = np.dot(X_test,w_m2.T)
y_p2 = set(y_p2,1,3)


w3= np.zeros((x3_tr.shape[1])) ###weight initialization
w_m3 = batch_gradient_descent_l1(x3_tr,y3_tr,w3,alpha,iters,lamb)
y_p3 = np.dot(X_test,w_m3.T)
y_p3 = set(y_p3,1,4)


w4= np.zeros((x4_tr.shape[1])) ###weight initialization
w_m4 = batch_gradient_descent_l1(x4_tr,y4_tr,w4,alpha,iters,lamb)
y_p4 = np.dot(X_test,w_m4.T)
y_p4 = set(y_p4,2,3)


w5= np.zeros((x5_tr.shape[1])) ###weight initialization
```

```
w_m5 = batch_gradient_descent_l1(x5_tr,y5_tr,w5,alpha,iters,lamb)
y_p5 = np.dot(X_test,w_m5.T)
y_p5 = set(y_p5,2,4)


w6= np.zeros((x6_tr.shape[1])) ###weight initialization
w_m6 = batch_gradient_descent_l1(x6_tr,y6_tr,w6,alpha,iters,lamb)
y_p6 = np.dot(X_test,w_m6.T)
y_p6 = set(y_p6,3,4)
```

DEFINING MODE FUNCTION

```
from collections import Counter
def my_mode(sample):
  return Counter(sample).most_common(1)[0][0]
```

CALCULATING MODE OF PREDICTED OUTPUT

```
from statistics import mode

cval = [0 for i in range(len(y_test))]

for i in range(len(y_test)):
  k=[y_p1[i],y_p2[i],y_p3[i],y_p4[i],y_p5[i],y_p6[i]]

  cval[i]=my_mode(k)
```

GENERATING CONFUSION MATRIX

```
for i in range(len(cval)):
    if (cval[i] == 0):
        cval[i] = 'None'

print(cval)
y_actual = pd.Series(y_test, name='Actual')
y_pred = pd.Series(cval, name='Predicted')
confmat = pd.crosstab(y_actual, y_pred)
print(confmat)
```

```
[3.0, 3.0, 3.0, 2.0, 2.0, 2.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 1.0, 3.0, 3.0, 3.0,
Predicted  1.0  2.0  3.0  4.0
Actual
1.0          3   14  129    5
2.0          3   19  147    4
3.0          6   32  141   11
4.0          4   19  141    5
```

FINDING ACCURACY

```
confmat = np.asarray(confmat)
Acc = (confmat[0][0] + confmat[1][1] + confmat[2][2])/sum(sum(confmat))
Acc1 = confmat[0][0]/sum(confmat[0])
Acc2 = confmat[1][1]/sum(confmat[1])
Acc3 = confmat[2][2]/sum(confmat[2])
Acc4 = confmat[3][3]/sum(confmat[3])
print('Overall Accuracy : ' + str(Acc))
print('Accuracy of class 1 : ' + str(Acc1))
print('Accuracy of class 2 : ' + str(Acc2))
print('Accuracy of class 3 : ' + str(Acc3))
print('Accuracy of class 4 : ' + str(Acc4))
```

```
Overall Accuracy : 0.23865300146412885
Accuracy of class 1 : 0.019867549668874173
Accuracy of class 2 : 0.10982658959537572
Accuracy of class 3 : 0.7421052631578947
Accuracy of class 4 : 0.029585798816568046
```

IMPLEMENTING BATCH GRADIENT DESCENT FOR MULTICLASS ONE VS ONE WITH L2 NORM

```
alpha=0.001
iters=1000
lamb=0.008
w1= np.zeros((x1_tr.shape[1])) ###weight initialization
w_m1 = batch_gradient_descent_l2(x1_tr,y1_tr,w1,alpha,iters,lamb)
y_p1 = np.dot(X_test,w_m1.T)
y_p1 = set(y_p1,1,2)


w2= np.zeros((x2_tr.shape[1])) ###weight initialization
w_m2 = batch_gradient_descent_l2(x2_tr,y2_tr,w2,alpha,iters,lamb)
y_p2 = np.dot(X_test,w_m2.T)
y_p2 = set(y_p2,1,3)


w3= np.zeros((x3_tr.shape[1])) ###weight initialization
w_m3 = batch_gradient_descent_l2(x3_tr,y3_tr,w3,alpha,iters,lamb)
y_p3 = np.dot(X_test,w_m3.T)
y_p3 = set(y_p3,1,4)


w4= np.zeros((x4_tr.shape[1])) ###weight initialization
w_m4 = batch_gradient_descent_l2(x4_tr,y4_tr,w4,alpha,iters,lamb)
y_p4 = np.dot(X_test,w_m4.T)
y_p4 = set(y_p4,2,3)


w5= np.zeros((x5_tr.shape[1])) ###weight initialization
w_m5 = batch_gradient_descent_l2(x5_tr,y5_tr,w5,alpha,iters,lamb)
y_p5 = np.dot(X_test,w_m5.T)
y_p5 = set(y_p5,2,4)
```

```
w6= np.zeros((x6_tr.shape[1])) ###weight initialization
w_m6 = batch_gradient_descent_l2(x6_tr,y6_tr,w6,alpha,iters,lamb)
y_p6 = np.dot(X_test,w_m6.T)
y_p6 = set(y_p6,3,4)
```

DEFINING MODE FUNCTION

```
from collections import Counter
def my_mode(sample):
  return Counter(sample).most_common(1)[0][0]
```

Double-click (or enter) to edit

CALCULATING MODE OF PREDICTED OUTPUT

```
from statistics import mode

cval = [0 for i in range(len(y_test))]

for i in range(len(y_test)):
  k=[y_p1[i],y_p2[i],y_p3[i],y_p4[i],y_p5[i],y_p6[i]]

  cval[i]=my_mode(k)
```

GENERATING CONFUSION MATRIX

```
for i in range(len(cval)):
    if (cval[i] == 0):
        cval[i] = 'None'

print(cval)
y_actual = pd.Series(y_test, name='Actual')
y_pred = pd.Series(cval, name='Predicted')
confmat = pd.crosstab(y_actual, y_pred)
print(confmat)
```

```
[3.0, 3.0, 3.0, 3.0, 3.0, 2.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0,
Predicted  1.0  2.0  3.0  4.0
Actual
1.0          2   14  135    0
2.0          1   20  149    3
3.0          4   30  152    4
4.0          3   20  144    2
```

FINDING ACCURACY

```
confmat = np.asarray(confmat)
Acc = (confmat[0][0] + confmat[1][1] + confmat[2][2])/sum(sum(confmat))
Acc1 = confmat[0][0]/sum(confmat[0])
Acc2 = confmat[1][1]/sum(confmat[1])
Acc3 = confmat[2][2]/sum(confmat[2])
Acc4 = confmat[3][3]/sum(confmat[3])
print('Overall Accuracy : ' + str(Acc))
print('Accuracy of class 1 : ' + str(Acc1))
print('Accuracy of class 2 : ' + str(Acc2))
print('Accuracy of class 3 : ' + str(Acc3))
print('Accuracy of class 4 : ' + str(Acc4))
```

```
Overall Accuracy : 0.2547584187408492
Accuracy of class 1 : 0.013245033112582781
Accuracy of class 2 : 0.11560693641618497
Accuracy of class 3 : 0.8
Accuracy of class 4 : 0.011834319526627219
```

Q5

MOUNTING THE DRIVE

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

importing libraries

```
import pandas as pd
import math
import numpy as np
import matplotlib.pyplot as plt
import sklearn
np.random.seed(0)
```

DEFINING LIKELIHOOD FUNCTION

```
def likelihood_function(X, cov, mean):
    return ((1 / (np.power((2 * np.pi), X.shape[0] / 2) * np.sqrt(np.linalg.det(cov)))) *
```

DEFINING MEAN FUNCTION

```
def mean(A):
    m = []
    A=np.array(A)
    for i in range(A.shape[1]):
        m.append(sum(A[:,i])/len(A))
    return m
```

DEFINING Maximum likelihood (ML) FUNCTION

```
def ML(test_X,mn,cov):
  res=[]
  for i in range(len(mn)):
    res.append(likelihood_function(test_X,cov[i],mn[i]))
  return (np.argmax(res))
```

DEFINING maximum a posteriori (MAP) FUNCTION

```
def MAP(test_X,Py,mn,cov):
```

```
  res=[]
  for i in range(len(mn)):
    res.append(likelihood_function(test_X,cov[i],mn[i])*Py[i])

  return (np.argmax(res))
```

LOADING DATA

```
df=pd.read_excel('/content/drive/MyDrive/nnfl data/data.xlsx',header=None)


for i in range(df.shape[1]-1):
    df[i]=(df[i]-df[i].mean())/df[i].std()
```

SPLITTING DATA INTO TRAINING AND TESTING

```
train, test = np.split(df.sample(frac=1, random_state=10),[int(0.7 * len(df))])
train=np.asarray(train)
test=np.asarray(test)
train_X=train[:,0:train.shape[1]-1]
train_Y=train[:,train.shape[1]-1]
test_X=test[:,0:test.shape[1]-1]
test_Y=test[:,test.shape[1]-1]


Py=[]
mn=[]
cov=[]
for j in np.unique(train_Y):
  Py.append(len([i for (i, val) in enumerate(train_Y) if val == j])/len(train_Y))
  x = np.array([train_X[i] for (i, val) in enumerate(train_Y) if val == j])
  mn.append(mean(x));
  cov.append(np.cov(x.T))
mn=np.array(mn)
cov=np.array(cov)
```

IMPLEMENTING MAP

```
ypred=[]
for i in range(len(test_X)):
  ypred.append(MAP(test_X[i],Py,mn,cov)+1)

ypred=np.array(ypred)

df_confusion = pd.crosstab(test_Y, ypred)
df_confusion


df_confusion=np.array(df_confusion)
row_sum=0.0;
overall_sum=0.0;
```

```
dia_el=0.0
```

ACCURACY WITH MAP

```
for i in range(df_confusion.shape[0]):
  row_sum=np.sum(df_confusion[i,:])
  overall_sum +=row_sum
  acc=np.round(df_confusion[i,i]/row_sum,5)*100;
  dia_el+=df_confusion[i,i]
  print("Accuracy for class "+str(i+1)+" is {}%\n".format(acc))
```

```
Accuracy for class 1 is 27.426000000000002%

Accuracy for class 2 is 29.412%

Accuracy for class 3 is 21.132%

Accuracy for class 4 is 25.2%
```

```
ov_acc=np.round((dia_el/overall_sum),5)*100
print("Overall Accuracy for the model is {}%\n".format(ov_acc))
```

```
Overall Accuracy for the model is 25.781%
```

IMPLEMENTING ML

```
ypred=[]
for i in range(len(test_X)):
  ypred.append(ML(test_X[i], mn ,cov)+1)
```

```
ypred=np.array(ypred)
df_confusion = pd.crosstab(test_Y, ypred)
df_confusion
```

col_0	1	2	3	4
row_0				
1.0	64	47	52	74
2.0	62	80	53	77
3.0	73	63	57	72
4.0	60	56	71	63

ACCURACY WITH ML

```
df_confusion=np.array(df_confusion)
```

```
df_confusion=np.array(df_confusion)
row_sum=0.0;
overall_sum=0.0;
dia_el=0.0
for i in range(df_confusion.shape[0]):
  row_sum=np.sum(df_confusion[i,:])
  overall_sum +=row_sum
  acc=np.round(df_confusion[i,i]/row_sum,5)*100;
  dia_el+=df_confusion[i,i]
  print("Accuracy for class "+str(i+1)+" is {}%\n".format(acc))
```

```
Accuracy for class 1 is 27.004%

Accuracy for class 2 is 29.412%

Accuracy for class 3 is 21.509%

Accuracy for class 4 is 25.2%
```

```
ov_acc=np.round((dia_el/overall_sum),5)*100
print("Overall Accuracy for the model is {}%\n".format(ov_acc))
```

```
Overall Accuracy for the model is 25.781%
```