Birla Institute of Technology & Science, Pilani
Hyderabad Campus

Assignment 1

Course No.: BITS F312

Course Title: Neural Network and Fuzzy Logic
Instructor-in-Charge: Dr. Rajesh Kumar Tripathy
Assignment by: Saksham Yadav

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.m
```

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

Linear Regression with L2 norm regularization commented

```
def cost_function(X,y,w):
  hypothesis = np.dot(X,w.T) ###calculation of hypothesis for all instances
  J = (1/(2*len(y))) * np.sum((hypothesis - y) ** 2) ####as mention in the class notes
  # J = (1/(2*len(y))) * np.sum((hypothesis - y) ** 2)+(lamb/2)*np.sum(w**2) ####as mentio
  return J


def batch_gradient_descent(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 = np.dot(X,w.T)
    #w = (w*(1-alpha*lamb)) -(alpha/len(y)) * np.dot(hypothesis - y, X)
    w = w - (alpha/len(y)) * np.dot(hypothesis - y, X)
    #cost_history[i] = cost_function(X,y,w,lamb)
    cost_history[i] = cost_function(X,y,w)
  return w,cost_history


def MB_gradient_descent(X,y,w,alpha, iters, batch_size):
  cost_history = np.zeros(iters)
  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))
    cost_history[i] = cost_function(ind_x,ind_y,w)
  return w, cost_history


def stochastic_gradient_descent(X,y,w,alpha, iters):
  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 - alpha * (ind_x.T.dot(ind_x.dot(w) - ind_y))
    cost_history[i] = cost_function(ind_x,ind_y,w)
  return w, cost_history
```

Linear Regression with L2 norm regularization

```
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 batch_gradient_descent_l2(X,y,w,alpha,iters,lamb):
    cost_history = np.zeros(iters)
    for i in range(iters):
        hypothesis = np.dot(X,w.T)
        w = (w*(1-alpha*lamb)) - (alpha/len(y)) * (np.dot(hypothesis - y, X))
        cost_history[i] = cost_function_l2(X,y,w,lamb)
    return w, cost_history


def MB_gradient_descent_l2(X,y,w,alpha, iters, batch_size,lamb):
    cost_history = np.zeros(iters)
    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*int((1-(alpha*lamb))) - (alpha/batch_size) * (np.dot((np.dot(ind_x,w) - ind_
        cost_history[i] = cost_function_l2(ind_x,ind_y,w,lamb)
    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
```

Linear Regression with Least Angle Regression Model

```
def cost_function_l1(X,y,w,lamb):
    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 batch_gradient_descent_l1(X,y,w,alpha,iters,lamb):
    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) - (lamb/2)*np.sign(w))
        cost_history[i] = cost_function_l2(X,y,w,lamb)
    return w, cost_history
```

```
def stochastic_gradient_descent_l1(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 - alpha * (ind_x.T.dot(ind_x.dot(w) - ind_y) - (lamb/2)*np.sign(w))
        cost_history[i] = cost_function_l1(ind_x,ind_y,w,lamb)
    return w, cost_history


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

        # print((ind_x.T@(ind_x@w - ind_y)).sum(axis=1).shape)
        w = w - (alpha/batch_size) * (ind_x.T@(ind_x@w - ind_y)).sum(axis=1)
        cost_history[i] = cost_function(ind_x,ind_y,w)
    return w, cost_history
```

TrainingData

```
dataxtr = pd.read_csv('/content/drive/MyDrive/xtr.csv',header=None)
datаytr = pd.read_csv('/content/drive/MyDrive/ytr.csv',header=None)
```

TestingData

```
dataxte = pd.read_csv('/content/drive/MyDrive/xte.csv',header=None)
datayte = pd.read_csv('/content/drive/MyDrive/yte.csv',header=None)
```

```
data_XTraining=dataxtr.values
X=data_XTraining[:,:]
# print(X)
m=X.shape[0]
xmin=np.min(X,axis=0)
xmax=np.max(X,axis=0)
# print(xmin , xmax)
```

```
X = (X- xmin)/(xmax-xmin) #Normalization
# print(X)

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


X.shape
```

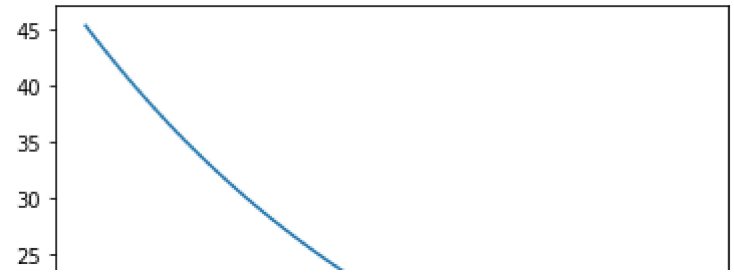
```
(55, 3)
```

```
data_YTraining=dataytr.values
Y=data_YTraining[:,:]
# print(Y)
m=X.shape[0]
ymin=np.min(Y,axis=0)
ymax=np.max(Y,axis=0)
# print(ymin , ymax)
y = (Y- ymin)/(ymax-ymin) #Normalization
#print(Y)


datayte_=datayte.values
k=datayte_.shape[0]
Yte=np.ones([k,1])
Yte=np.append(Yte,dataxte_,axis=1)


#Linear regression starts from here
w= np.zeros(X.shape[1]) ##weight initialization
#w=[0.5, 0.5, 0.5]
w1=np.zeros((X.shape[1]))
print(w)
```

```
[0. 0. 0.]
```

Batch Gradient Descent

```
alpha=0.005 ##learning rate
iters=100 ###iterations
lamb=5
batch_w,J_his = batch_gradient_descent(X,Y,w,alpha,iters,lamb)


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



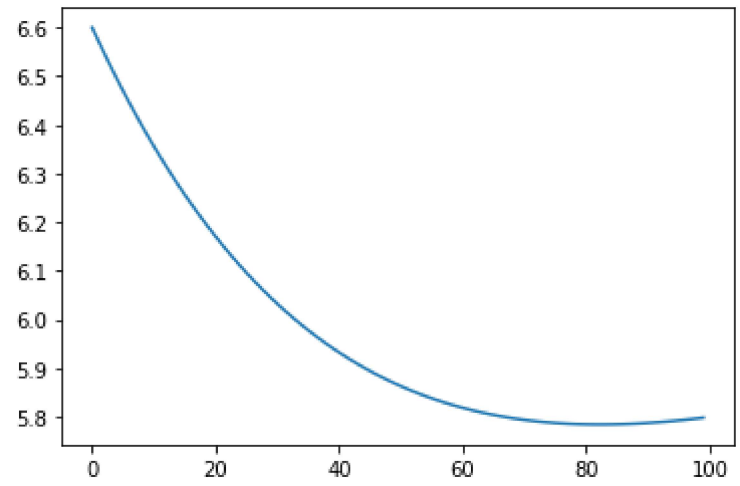

```
print(batch_w[0])
```

```
[0.45651057 0.23564335 0.26581355]
```

```
# print(y)
```

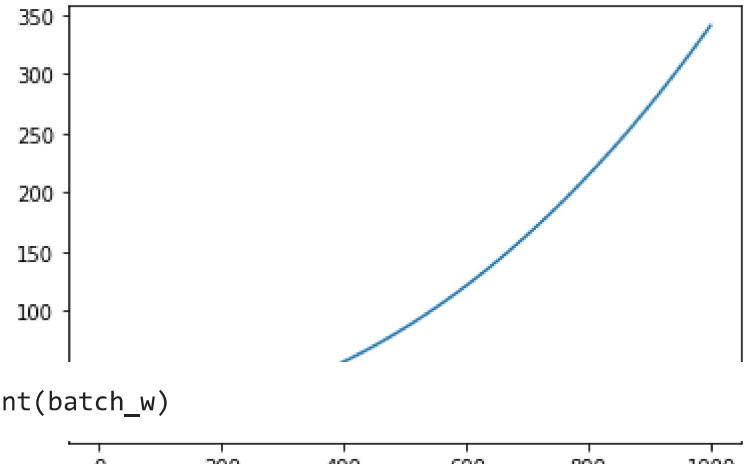
Batch Gradient Descent with L2 norm regularization

```
alpha=0.002 ##learning rate
iters=100 ###iterations
lamb=3.5
batch_w_l2,J_his_l2 = batch_gradient_descent_l2(X,y,w,alpha,iters,lamb)
plt.plot(range(iters),J_his_l2)
plt.show()
```



Batch Gradient Descent with L1 norm regularization

```
alpha=0.004 ##learning rate
iters=1000 ###iterations
lamb=3
batch_w,J_his = batch_gradient_descent_l1(X,y,w,alpha,iters,lamb)

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

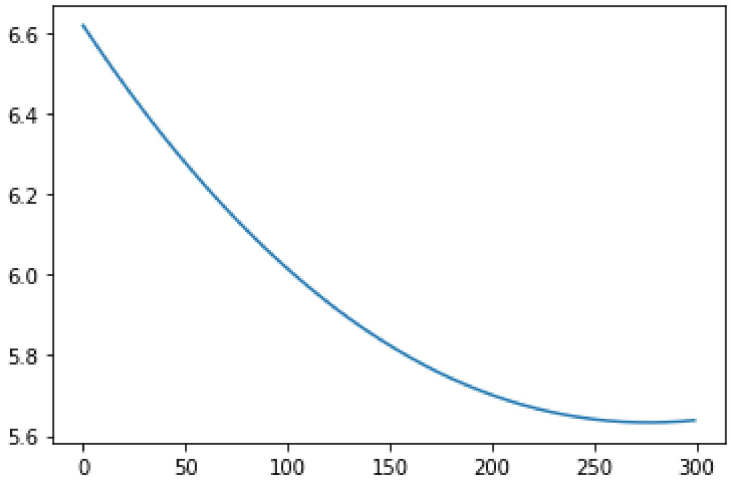



```
# print(batch_w)
```

```
bgd=batch_w[-1:]
print(bgd)
```

```
[[1.78991329 1.15029434 1.18273904]]
```

```
alpha=0.0005#learning rate
iters=300 ###iterations
lamb=3
batch_w_l1,J_his_l1 = batch_gradient_descent_l1(X,y,w,alpha,iters,lamb)

plt.plot(range(iters),J_his_l1)
plt.show()
```



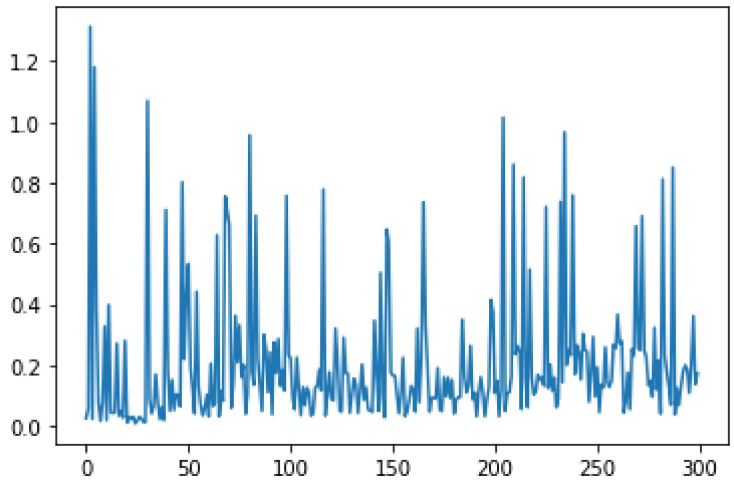

```
# print(batch_w_l1)
```

```
bgd_l1=batch_w_l1[-1:]
print(bgd_l1)
```

```
[[0.09544143 0.05084858 0.05689333]]
```

```
alpha=0.05
iters=300 ###iterations
lamb=1
w_n,J_sgd = stochastic_gradient_descent(X,y,w,alpha, iters)
```

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



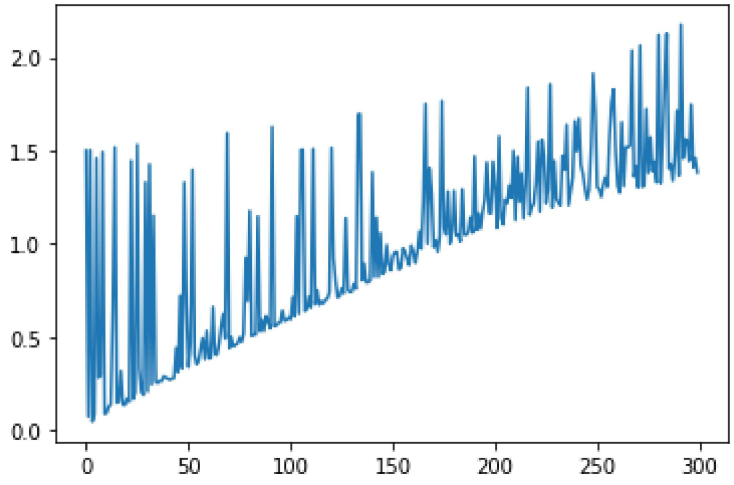

```
print(w_n)
```

```
[[ 0.27939833  0.27939833  0.27939833]
 [-0.02090169 -0.02090169 -0.02090169]
 [ 0.09714084  0.09714084  0.09714084]]
```

```
sgd=w_n[-1:]
print(sgd)
```

```
[[0.09714084 0.09714084 0.09714084]]
```

```
alpha=0.002
iters=300 ###iterations
lamb=1
w_n_l1,J_sgd_l1 = stochastic_gradient_descent_l1(X,y,w,alpha, iters,lamb)

plt.plot(range(iters),J_sgd_l1)
plt.show()
```



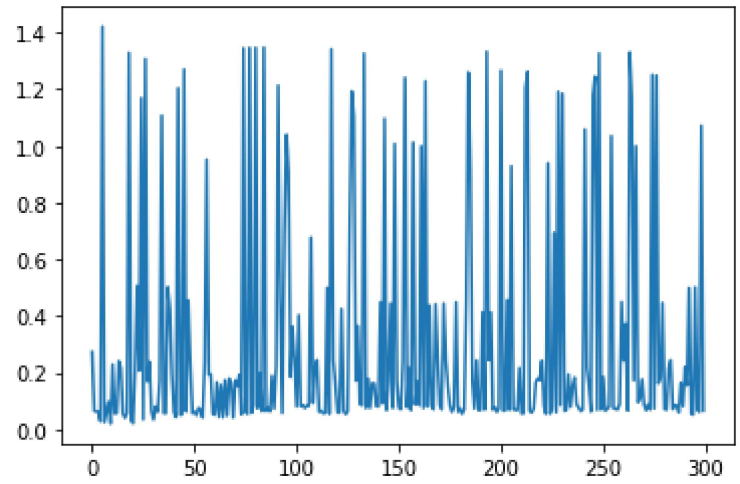

```
print(w_n_l1)
```

```
      [[0.32180733 0.32180733 0.32180733]
       [0.29396355 0.29396355 0.29396355]
       [0.30018099 0.30018099 0.30018099]]
```

```
sgd_l1=w_n_l1[-1:]
print(sgd_l1)
```

```
      [[0.30018099 0.30018099 0.30018099]]
```

```
alpha=0.01
iters=300 ###iterations
lamb=3
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()
```




```
print(w_n_l2)
```

```
      [[0.08481359 0.08481359 0.08481359]
       [0.04128165 0.04128165 0.04128165]
       [0.05071442 0.05071442 0.05071442]]
```

```
sgd_l2=w_n_l2[-1:]
print(sgd_l2)
```

```
      [[0.05071442 0.05071442 0.05071442]]
```

```
alpha=0.05
iters=300 ###iterations
lamb=3
batch_size=25
mb_w,J_mb = MB_gradient_descent(X,y,w1,alpha, iters, batch_size)

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

Performance Measures for Regression(Self Defined functions)

Have defined these functions but have not used them because sklearn libraries were allowed to be used which I have used to find the three errors.

Mean Absolute Error

```
def mean_abs_error(Ypre,Yact):
  sum_err=abs(Yact - Ypre)
  ma_err=sum_err/Ypre.shape[0]
  return ma_err
```

Mean Square Error

```
def mean_square_error(Ypre,Yact):
  sum_error=((Yact - Ypre)**2)
  ms_err=sum_error/Ypre.shape[0]
  return ms_err
```

Correlation Coefficient

```
def correlation_coeff(test_instances,Ypred,Yact):
  ypm=np.mean(Ypred)##mean of Ypred data
  yam=np.mean(Yact)##mean of Yactual data
  num=((Yact - yam)*(Ypred-ypm))
  d1=pow(((Yact - yam)**2),1/2)
  d2=pow(((Ypred - ypm)**2),1/2)
  c_c=num/(d1*d2)
  return c_c
```

Finding Errors

Using sklearn libraries(was allowed)

```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import matthews_corrcoef


dataxte_=dataxte.values
k=dataxte_.shape[0]
Xte=np.ones([k,1])
Xte=np.append(Xte,dataxte_,axis=1)
print(Xte)
print(bgd.T)
```

```
[[ 1.   17.4   8.58]
 [ 1.   17.86  9.08]
 [ 1.   18.3   9.58]
 [ 1.   16.97  8.08]
 [ 1.   16.46  7.58]
 [ 1.   17.4   8.58]
 [ 1.   17.85  9.08]
 [ 1.   18.3   9.58]
 [ 1.   16.9   8.08]
 [ 1.   16.42  7.58]]
[[1.78991329]
 [1.15029434]
 [1.18273904]]
```

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



y_pred_bgd=Xte.dot(bgd.T)
print(mean_squared_error(y_pred_bgd, Yte))
print(mean_absolute_error(y_pred_bgd, Yte))
# print(matthews_corrcoef(y_pred_bgd, Yte))


# y_pred_bgd_l1=Xte.dot(bgd_l1.T)
# print(mean_squared_error(y_pred_bgd_l1, Yte))
# print(mean_absolute_error(y_pred_bgd_l1, Yte))
# print(matthews_corrcoef(y_pred_bgd_l1, Yte))

# y_pred_bgd_l2=Xte.dot(bgd_l2.T)
# print(mean_squared_error(y_pred_bgd_l2, Yte))
# print(mean_absolute_error(y_pred_bgd_l2, Yte))
# print(matthews_corrcoef(y_pred_bgd_l2, Yte))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-359-5a96d7ea412b> in <module>()
      1 y_pred_bgd=Xte.dot(bgd.T)
----> 2 print(mean_squared_error(y_pred_bgd, Yte))
      3 print(mean_absolute_error(y_pred_bgd, Yte))
      4 # print(matthews_corrcoef(y_pred_bgd, Yte))
      5

                                  1 frames
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_regression.py in
_check_reg_targets(y_true, y_pred, multioutput, dtype)
```

Stochastic Gradient descent

```
      107                 y_true.shape[1], y_pred.shape[1]
```

```
y_pred_sgd=Xte.dot(sgd.T)
print(mean_squared_error(y_pred_sgd, Yte))
print(mean_absolute_error(y_pred_sgd, Yte))
print(matthews_corrcoef(y_pred_sgd, Yte))

y_pred_sgd_l1=Xte.dot(sgd_l1.T)
print(mean_squared_error(y_pred_sgd_l1, Yte))
print(mean_absolute_error(y_pred_sgd_l1, Yte))
print(matthews_corrcoef(y_pred_sgd_l1, Yte))

y_pred_sgd_l2=Xte.dot(sgd_l2.T)
print(mean_squared_error(y_pred_sgd_l2, Yte))
print(mean_absolute_error(y_pred_sgd_l2, Yte))
print(matthews_corrcoef(y_pred_sgd_l2, Yte))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-357-d5771ea10559> in <module>()
      1 y_pred_sgd=Xte.dot(sgd.T)
----> 2 print(mean_squared_error(y_pred_sgd, Yte))
      3 print(mean_absolute_error(y_pred_sgd, Yte))
      4 print(matthews_corrcoef(y_pred_sgd, Yte))
      5

                                  1 frames
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_regression.py in
_check_reg_targets(y_true, y_pred, multioutput, dtype)
    105         raise ValueError(
    106             "y_true and y_pred have different number of output ({0}!=
{1})".format(
--> 107                 y_true.shape[1], y_pred.shape[1]
    108             )
    109         )

ValueError: y_true and y_pred have different number of output (1!=3)
```

Mini Batch Gradient Descent

0s completed at 3:00 AM

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

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

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

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

```
pip install openpyxl==3.0.9
```

```
Requirement already satisfied: openpyxl==3.0.9 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: et-xmlfile in /usr/local/lib/python3.7/dist-packages (
```

TRAINING DATA

```
dataxtr = pd.read_csv('/content/drive/MyDrive/xtr.csv',header=None)
dataytr = pd.read_csv('/content/drive/MyDrive/ytr.csv',header=None)
```

TESTING DATA

```
dataxte = pd.read_csv('/content/drive/MyDrive/xte.csv',header=None)
datayte = pd.read_csv('/content/drive/MyDrive/yte.csv',header=None)
```

```
X=dataxtr.values
Y=dataytr.values
Xte=dataxte.values
Yte=datayte.values
Yte1=Yte[:,0].reshape(Yte.shape[0],1)
k=Xte.shape[0]
```

```
datan_x=dataxtr.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]
# print(m)
# print(X1)
```

```
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)
# print(X1)
# print(X2)


datan_y=dataytr.values
y=datan_y

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



# print(ztr.shape)
```

Making the Z matrix

for training data

```
ztr=np.ones([m,1])
ztr=np.append(ztr,X1,axis=1)
ztr=np.append(ztr,X2,axis=1)
ztr=np.append(ztr,X1**2,axis=1)
ztr=np.append(ztr,X2**2,axis=1)
ztr=np.append(ztr,X1*X2,axis=1)
print(ztr)
```

```
       [1.00000000e+00 9.62962963e-01 1.00000000e+00 9.27297668e-01
        1.00000000e+00 9.62962963e-01]
       [1.00000000e+00 2.59259259e-01 4.52554745e-01 6.72153635e-02
        2.04805797e-01 1.17329008e-01]
       [1.00000000e+00 2.11640212e-02 2.70072993e-01 4.47915792e-04
        7.29394214e-02 5.71583053e-03]
       [1.00000000e+00 5.13227513e-01 6.13138686e-01 2.63402480e-01
        3.75939048e-01 3.14679643e-01]
       [1.00000000e+00 7.56613757e-01 8.02919708e-01 5.72464377e-01
        6.44680058e-01 6.07500097e-01]
       [1.00000000e+00 9.89417989e-01 9.81751825e-01 9.78947958e-01
        9.63836646e-01 9.71362917e-01]
       [1.00000000e+00 2.59259259e-01 4.30656934e-01 6.72153635e-02
        1.85465395e-01 1.11651798e-01]
       [1.00000000e+00 0.00000000e+00 2.51824818e-01 0.00000000e+00
        6.34157387e-02 0.00000000e+00]
       [1.00000000e+00 5.13227513e-01 6.13138686e-01 2.63402480e-01
        3.75939048e-01 3.14679643e-01]
       [1.00000000e+00 7.51322751e-01 7.99270073e-01 5.64485877e-01
```

```
 6.38832650e-01 6.00509790e-01]
[1.00000000e+00 9.78835979e-01 9.78102190e-01 9.58119873e-01
 9.56683894e-01 9.57401614e-01]
[1.00000000e+00 2.69841270e-01 4.37956204e-01 7.28143109e-02
 1.91805637e-01 1.18178658e-01]
[1.00000000e+00 1.58730159e-02 2.55474453e-01 2.51952633e-04
 6.52671959e-02 4.05515004e-03]
[1.00000000e+00 5.13227513e-01 8.61313869e-01 2.63402480e-01
 7.41861580e-01 4.42049975e-01]
[1.00000000e+00 5.13227513e-01 7.44525547e-01 2.63402480e-01
 5.54318291e-01 3.82110995e-01]
[1.00000000e+00 5.13227513e-01 6.05839416e-01 2.63402480e-01
 3.67041398e-01 3.10933457e-01]
[1.00000000e+00 5.13227513e-01 6.13138686e-01 2.63402480e-01
 3.75939048e-01 3.14679643e-01]
[1.00000000e+00 5.13227513e-01 6.05839416e-01 2.63402480e-01
 3.67041398e-01 3.10933457e-01]
[1.00000000e+00 5.13227513e-01 6.13138686e-01 2.63402480e-01
 3.75939048e-01 3.14679643e-01]
[1.00000000e+00 5.13227513e-01 5.91240876e-01 2.63402480e-01
 3.49565773e-01 3.03441084e-01]
[1.00000000e+00 7.67195767e-01 7.73722628e-01 5.88589345e-01
 5.98646705e-01 5.93596725e-01]
[1.00000000e+00 1.00000000e+00 9.56204380e-01 1.00000000e+00
 9.14326815e-01 9.56204380e-01]
[1.00000000e+00 2.80423280e-01 4.08759124e-01 7.86372162e-02
 1.67084022e-01 1.14625574e-01]
[1.00000000e+00 2.64550265e-02 2.26277372e-01 6.99868425e-04
 5.12014492e-02 5.98617387e-03]
[1.00000000e+00 5.13227513e-01 5.91240876e-01 2.63402480e-01
 3.49565773e-01 3.03441084e-01]
[1.00000000e+00 7.51322751e-01 7.73722628e-01 5.64485877e-01
 5.98646705e-01 5.81315413e-01]
[1.00000000e+00 9.84126984e-01 9.56204380e-01 9.68505921e-01
 9.14326815e-01 9.41026532e-01]
[1.00000000e+00 2.64550265e-01 4.08759124e-01 6.99868425e-02
 1.67084022e-01 1.08137334e-01]
[1.00000000e+00 1.58730159e-02 2.26277372e-01 2.51952633e-04
 5.12014492e-02 3.59170432e-03]]
```

for testing data

```
zte=np.ones([m,1])
zte=np.append(zte,X1,axis=1)
zte=np.append(zte,X2,axis=1)
zte=np.append(zte,X1**2,axis=1)
zte=np.append(zte,X2**2,axis=1)
zte=np.append(zte,X1*X2,axis=1)
# print(zte)


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
```

```
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(X,y,w,alpha, iters):
    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 - alpha * (ind_x.T.dot(ind_x.dot(w) - ind_y))
        cost_history[i] = cost_function(ind_x,ind_y,w)
    return w, cost_history


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

        # print((ind_x.T@(ind_x@w - ind_y)).sum(axis=1).shape)
        w = w - (alpha/batch_size) * (ind_x.T@(ind_x@w - ind_y)).sum(axis=1)
        cost_history[i] = cost_function(ind_x,ind_y,w)
    return w, cost_history


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

    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 batch_gradient_descent_l1(X,y,w,alpha,iters,lamb):
    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) - (lamb/2)*np.sign(w))
        cost_history[i] = cost_function_l1(X,y,w,lamb)
    return w, cost_history


def cost_function_l1_mbg(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 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_mbg(ind_x,ind_y,w,lamb)
    return w, cost_history


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


def cost_function_l2_mbg(X,y,w,lamb):
    w=w.ravel()#flatten the array into one dimensional array
    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 batch_gradient_descent_l2(X,y,w,alpha,iters,lamb):
    cost_history = np.zeros(iters)
    for i in range(iters):
        hypothesis = np.dot(X,w.T)
        w = (w*(1-alpha*lamb)) - (alpha/len(y)) * (np.dot(hypothesis - y, X))
        cost_history[i] = cost_function_l2(X,y,w,lamb)
    return w, cost_history


def stochastic_gradient_descent_l1(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 - alpha * (ind_x.T.dot(ind_x.dot(w) - ind_y) - (lamb/2)*np.sign(w))
```

```
        cost_history[i] = cost_function_l1(ind_x,ind_y,w,lamb)
    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_l2(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*((1-(alpha*lamb))) - (alpha/batch_size) * (ind_x.T.dot(ind_x.dot(w) - ind_y)
        cost_history[i] = cost_function_l2_mbg(ind_x,ind_y,w,lamb)
    return w, cost_history
```

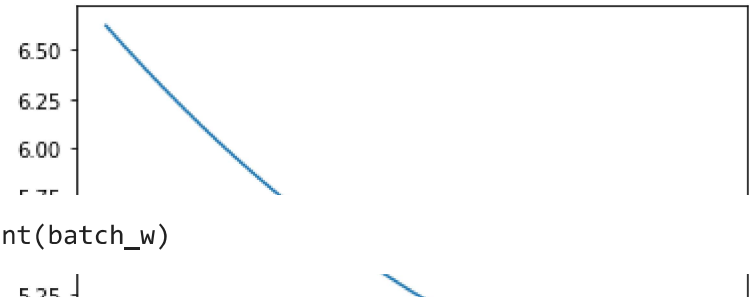
Weight initialization

```
w= np.zeros((ztr.shape[1]))
w1= np.zeros((ztr.shape[1]))
print(w.shape)
```

```
(6,)
```

Batch Gradient Descent

```
alpha=0.0002#learning rate
iters=1000 ###iterations
lamb=0.2
batch_w,J_his = batch_gradient_descent(ztr,y,w,alpha,iters)

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



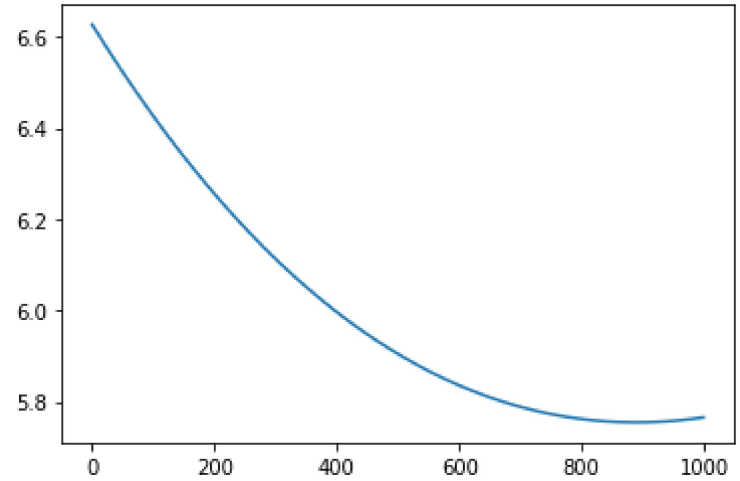

```
# print(batch_w)
```



```
bgd=batch_w[-1:]
print(bgd)
```

```
[[0.12040141 0.06170125 0.06961951 0.04162253 0.04871827 0.04190613]]
```

```
alpha=0.0002#learning rate
iters=1000 ###iterations
lamb=0.2
batch_w_l1,J_his_l1 = batch_gradient_descent_l1(ztr,y,w,alpha,iters,lamb)

plt.plot(range(iters),J_his_l1)
plt.show()
```




```
print(batch_w_l1)
```

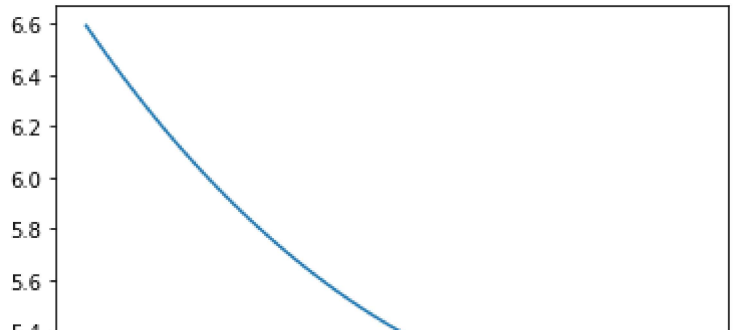
```
[[ 0.03322797  0.01730444  0.01931353  0.01180157  0.01367408  0.0118069 ]
 [ 0.02804292  0.01471878  0.01627816  0.01006477  0.0115858   0.01000649]
 [ 0.02919655  0.01529409  0.01695352  0.01045121  0.01205044  0.01040708]
 [ 0.03363514  0.01750747  0.01955188  0.01193794  0.01383805  0.01194827]
 [ 0.02906543  0.01522871  0.01687676  0.01040729  0.01199763  0.01036155]
 [ 0.03347595  0.01742809  0.0194587   0.01188463  0.01377394  0.011893  ]
 [-0.01400077 -0.00716497 -0.0083481  -0.0049323  -0.00585696 -0.0050999 ]
 [ 0.02899941  0.01519578  0.01683811  0.01038517  0.01197104  0.01033863]
 [ 0.03296895  0.01717528  0.0191619   0.01171482  0.01356976  0.01171696]
 [ 0.02893308  0.0151627   0.01679928  0.01036296  0.01194433  0.0103156 ]
 [ 0.02919655  0.01529409  0.01695352  0.01045121  0.01205044  0.01040708]
 [ 0.02543449  0.01341738  0.01475076  0.00919038  0.01053468  0.00910018]
 [ 0.0268821   0.0141386   0.01559777  0.00967462  0.01111711  0.00960223]
 [ 0.03372022  0.01755049  0.01960207  0.01196703  0.01387285  0.01197835]
 [ 0.07519525  0.03878705  0.04359464  0.02629112  0.03065326  0.02641365]
 [ 0.07025343  0.03632168  0.04070101  0.02463478  0.02866204  0.02469679]
 [ 0.06609717  0.03424835  0.03826748  0.02324187  0.02698749  0.02325298]
```

```
[ 0.06965782  0.03602339  0.04035152  0.02443399  0.02842102  0.02448882]
[ 0.07768569  0.04002893  0.04505256  0.02712529  0.03165626  0.02727838]
[ 0.01475473  0.0079078   0.008595    0.00547005  0.00620868  0.00538127]
[ 0.01101486  0.00604216  0.00640525  0.00421667  0.00470187  0.00408209]
[ 0.00659604  0.00383777  0.00381794  0.00273569  0.00292146  0.002547  ]
[ 0.01790647  0.00948002  0.0104404   0.00652631  0.00747853  0.00647614]
[ 0.02037634  0.01071207  0.01188655  0.00735402  0.00847362  0.00733412]
[ 0.02906543  0.01522871  0.01687676  0.01040729  0.01199763  0.01036155]
[ 0.02550473  0.01345245  0.01479191  0.00921395  0.01056301  0.00912461]
[ 0.02099711  0.01120379  0.01215262  0.00770324  0.00874685  0.0075587 ]
[ 0.03231037  0.01684744  0.01877673  0.01149479  0.01330504  0.01148881]
[ 0.03474592  0.01806235  0.02020278  0.01231098  0.01428629  0.01233485]
[ 0.01497594  0.0080181   0.00872449  0.00554414  0.00629777  0.00545808]
[ 0.01104993  0.00605959  0.00642574  0.00422836  0.00471595  0.00409422]
[ 0.00664558  0.00386242  0.00384691  0.00275223  0.00294137  0.00256416]
[ 0.01818408  0.00961848  0.01060293  0.00661932  0.00759035  0.00657255]
[ 0.0206547   0.01085091  0.01204953  0.00744728  0.00858576  0.0074308 ]
[ 0.0721188   0.03725194  0.04179303  0.02525965  0.02941335  0.02534453]
[ 0.06828086  0.03533736  0.03954585  0.02397339  0.02786701  0.02401127]
[ 0.06394853  0.03317613  0.03700919  0.0225214   0.02612147  0.02250624]
[ 0.07521025  0.03879411  0.04360314  0.02629572  0.03065892  0.02641847]
[ 0.0777088   0.04004046  0.04506609  0.02713304  0.03166557  0.02728641]
[ 0.04116189  0.02144539  0.02386225  0.01460279  0.01689477  0.01457295]
[ 0.14232863  0.07318645  0.08241523  0.04949578  0.05781697  0.0497794 ]
[ 0.18646931  0.09575192  0.10796804  0.06471237  0.07567061  0.06513994]
[ 0.11497594  0.05917731  0.06659443  0.04004628  0.04675004  0.04025937]
[ 0.11504073  0.05920962  0.06663236  0.04006799  0.04677613  0.04028187]
[ 0.16497594  0.08475692  0.0955294   0.05729735  0.06697617  0.05766002]
[ 0.18659797  0.09581608  0.10804336  0.06475547  0.07572243  0.06518462]
[ 0.18268672  0.09386491  0.10575323  0.06344461  0.07414653  0.06382586]
[ 0.17826838  0.09166076  0.10316621  0.06196379  0.07236632  0.06229095]
[ 0.1896426   0.09733488  0.10982604  0.06577583  0.07694912  0.06624228]
[ 0.19211014  0.09856577  0.11127084  0.06660276  0.07794329  0.06709946]
[ 0.1151694   0.05927379  0.06670769  0.04011109  0.04682796  0.04032655]
[ 0.11140589  0.05739634  0.06450408  0.03884976  0.0453116   0.03901914]
[ 0.10701298  0.05520488  0.06193194  0.03737747  0.04354164  0.03749306]
[ 0.11830927  0.06084011  0.06854615  0.04116338  0.04809303  0.04141731]
[ 0.12072715  0.06204622  0.06996186  0.04197366  0.04906718  0.04225723]]
```

```
bgd_l1=batch_w_l1[-1:]
print(bgd_l1)
```

```
[[0.12072715 0.06204622 0.06996186 0.04197366 0.04906718 0.04225723]]
```

```
alpha=0.002 ##learning rate
iters=100 ###iterations
lamb=1
batch_w_l2,J_his_l2 = batch_gradient_descent_l2(ztr,y,w,alpha,iters,lamb)

plt.plot(range(iters),J_his_l2)
plt.show()
```




```
print(batch_w_l2)
```

```
[[ 0.03019495  0.01555516  0.01741088  0.01050073  0.01222506  0.01050986]
 [ 0.02570683  0.0133155   0.01478163  0.00899582  0.01041567  0.00894951]
 [ 0.02670512  0.01381368  0.01536647  0.00933057  0.01081815  0.0092966 ]
 [ 0.03054761  0.01573114  0.01761747  0.01061897  0.01236723  0.01063246]
 [ 0.02659165  0.01375706  0.01529999  0.00929252  0.0107724   0.00925714]
 [ 0.03040973  0.01566234  0.0175367   0.01057274  0.01231164  0.01058453]
 [-0.01183937 -0.00590957 -0.00693683 -0.00397135 -0.00477443 -0.0041175 ]
 [ 0.02653452  0.01372855  0.01526652  0.00927336  0.01074937  0.00923728]
 [ 0.02997062  0.01544323  0.01727947  0.01042552  0.01213463  0.01043188]
 [ 0.02647712  0.0136999   0.0152329   0.00925412  0.01072622  0.00921733]
 [ 0.02670512  0.01381368  0.01536647  0.00933057  0.01081815  0.0092966 ]
 [ 0.02345245  0.01219002  0.01346061  0.0082394   0.00950635  0.00816528]
 [ 0.02470764  0.01281586  0.01419556  0.00865975  0.01001189  0.00860116]
 [ 0.03061921  0.01576734  0.01765974  0.01064345  0.01239653  0.01065779]
 [ 0.06829966  0.03505974  0.03945709  0.02365597  0.02764107  0.02377181]
 [ 0.06402585  0.03292624  0.03695284  0.02222212  0.02591734  0.02228522]
 [ 0.06043265  0.03113262  0.03484748  0.02101675  0.02446824  0.02103549]
 [ 0.06351567  0.03267064  0.03665325  0.02205005  0.02571073  0.02210689]
 [ 0.07045669  0.03613613  0.04072072  0.02437923  0.02851066  0.02452171]
 [ 0.01361046  0.00711627  0.00778764  0.00481395  0.00552125  0.00473954]
 [ 0.01037696  0.00550218  0.00589302  0.00372922  0.0042172   0.0036149 ]
 [ 0.00655771  0.00359566  0.00365517  0.00244796  0.00267687  0.00228649]
 [ 0.0163366   0.00847709  0.00938497  0.00572848  0.00662069  0.00568771]
 [ 0.01847437  0.0095442   0.01063756  0.00644562  0.00748283  0.00643124]
 [ 0.02659165  0.01375706  0.01529999  0.00929252  0.0107724   0.00925714]
 [ 0.02351303  0.01222028  0.01349613  0.00825975  0.0095308   0.00818637]
 [ 0.01961684  0.01027539  0.01121321  0.0069527   0.00795948  0.00683123]
 [ 0.02939836  0.01515811  0.01694455  0.01023409  0.01190434  0.01023336]
 [ 0.03150654  0.01621045  0.01817978  0.01094129  0.01275454  0.01096658]
 [ 0.01380194  0.00721181  0.00789981  0.00487815  0.00559844  0.0048061 ]
 [ 0.01040754  0.0055174   0.00591091  0.00373943  0.00422948  0.00362549]
 [ 0.00660077  0.00361711  0.00368037  0.00246236  0.0026942   0.00230142]
 [ 0.01657689  0.00859701  0.00952575  0.00580906  0.00671758  0.00577127]
 [ 0.01871549  0.00966455  0.01077882  0.00652649  0.00758006  0.00651509]
 [ 0.06564     0.03373175  0.03789846  0.02276339  0.02656811  0.02284642]
 [ 0.06232173  0.03207534  0.03595416  0.02165021  0.02522986  0.02169228]
 [ 0.05857714  0.03020611  0.03376007  0.02039401  0.02371966  0.02038986]
 [ 0.06831398  0.03506655  0.03946525  0.02366043  0.02764653  0.02377647]
 [ 0.07047669  0.03614612  0.04073244  0.02438594  0.02851873  0.02452867]
 [ 0.0376568   0.01944169  0.02168987  0.0131286   0.01525107  0.0131086 ]
 [ 0.12938294  0.06635538  0.07477894  0.04476629  0.05235501  0.04503015]
 [ 0.16937218  0.08679962  0.09792861  0.05855267  0.06853006  0.05894655]
 [ 0.10451854  0.05362171  0.06039745  0.03617732  0.04229536  0.03637666]
 [ 0.10457461  0.05364969  0.0604303   0.03619612  0.04231797  0.03639615]
 [ 0.14987684  0.07682666  0.08664627  0.0518269   0.06064383  0.05216193]
 [ 0.16948353  0.08685518  0.09799384  0.05859001  0.06857495  0.05898526]
 [ 0.16610196  0.08516715  0.09601244  0.05745557  0.06721114  0.05780908]
```

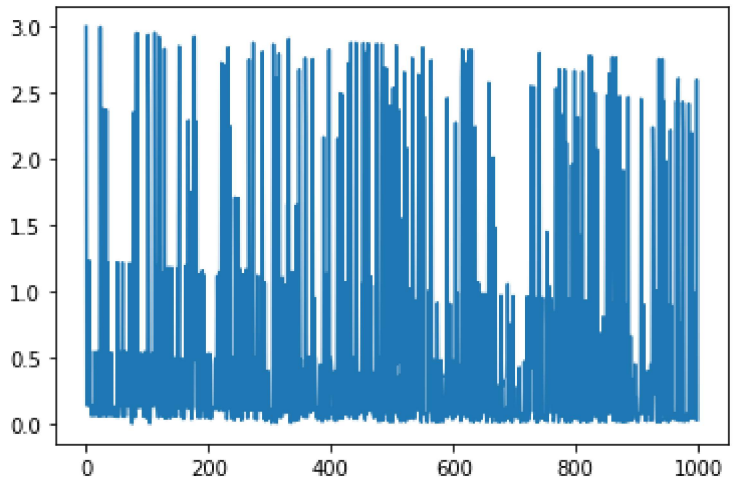
```
 [ 0.16228307  0.08326082  0.09377481  0.05617443  0.06567097  0.05648081]
 [ 0.17211713  0.08816981  0.09953695  0.05947349  0.06963707  0.05990125]
 [ 0.17425303  0.089236    0.10078845  0.06019001  0.07049847  0.06064413]
 [ 0.10468596  0.05370526  0.06049553  0.03623346  0.04236286  0.03643486]
 [ 0.10143203  0.05208095  0.05858893  0.03514185  0.04105055  0.0353031 ]
 [ 0.09763506  0.05018557  0.05636414  0.03386807  0.03951922  0.03398245]
 [ 0.10740192  0.05506101  0.06208691  0.03714458  0.04345821  0.03737951]
 [ 0.10949492  0.05610578  0.06331326  0.0378467   0.0443023   0.03810747]]
```

```
bgd_l2=batch_w_l2[-1:]
print(bgd_l2)
```

```
[[0.10949492 0.05610578 0.06331326 0.0378467  0.0443023  0.03810747]]
```

Stochastic Gradient Descent

```
alpha=0.0002#learning rate
iters=1000 ###iterations
lamb=0.2
w_n,J_sgd = stochastic_gradient_descent(ztr,y,w.T,alpha, iters)

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



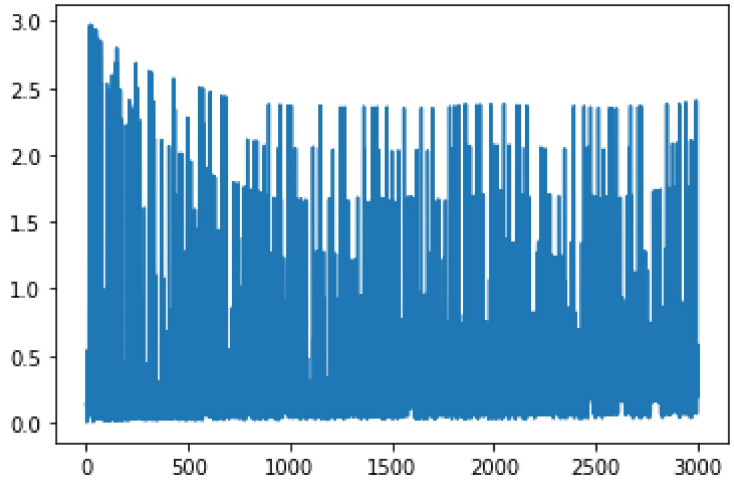

```
print(w_n)
```

```
[[0.06557465 0.06557465 0.06557465 0.06557465 0.06557465 0.06557465]
 [0.03282488 0.03282488 0.03282488 0.03282488 0.03282488 0.03282488]
 [0.037912   0.037912   0.037912   0.037912   0.037912   0.037912  ]
 [0.0217596  0.0217596  0.0217596  0.0217596  0.0217596  0.0217596 ]
 [0.02565183 0.02565183 0.02565183 0.02565183 0.02565183 0.02565183]
 [0.02221456 0.02221456 0.02221456 0.02221456 0.02221456 0.02221456]]
```

```
sgd=w_n[-1:]
print(sgd)
```

```
[[0.02221456 0.02221456 0.02221456 0.02221456 0.02221456 0.02221456]]
```

```
alpha=0.001
iters=3000 ###iterations
lamb=0.005
w_n_l1,J_sgd_l1 = stochastic_gradient_descent_l1(ztr,y,w,alpha, iters,lamb)

plt.plot(range(iters),J_sgd_l1)
plt.show()
```



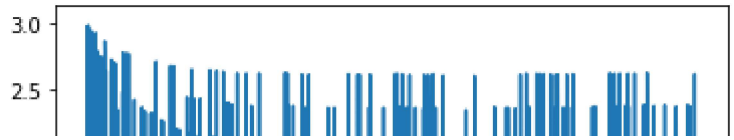

```
print(w_n_l1)
```

```
[[0.22891475 0.22891475 0.22891475 0.22891475 0.22891475 0.22891475]
 [0.06718753 0.06718753 0.06718753 0.06718753 0.06718753 0.06718753]
 [0.10236949 0.10236949 0.10236949 0.10236949 0.10236949 0.10236949]
 [0.02847351 0.02847351 0.02847351 0.02847351 0.02847351 0.02847351]
 [0.0472459  0.0472459  0.0472459  0.0472459  0.0472459  0.0472459 ]
 [0.03467277 0.03467277 0.03467277 0.03467277 0.03467277 0.03467277]]
```

```
sgd_l1=w_n_l1[-1:]
print(sgd_l1)
```

```
[[0.03467277 0.03467277 0.03467277 0.03467277 0.03467277 0.03467277]]
```

```
alpha=0.001
iters=3000 ###iterations
lamb=1
w_n_l2,J_sgd_l2 = stochastic_gradient_descent_l2(ztr,y,w,alpha, iters,lamb)
plt.plot(range(iters),J_sgd_l2)
plt.show()
```




```
print(w_n_l2)
```

```
[[0.14310135 0.14310135 0.14310135 0.14310135 0.14310135 0.14310135]
 [0.05971401 0.05971401 0.05971401 0.05971401 0.05971401 0.05971401]
 [0.07555136 0.07555136 0.07555136 0.07555136 0.07555136 0.07555136]
 [0.03423445 0.03423445 0.03423445 0.03423445 0.03423445 0.03423445]
 [0.04522525 0.04522525 0.04522525 0.04522525 0.04522525 0.04522525]
 [0.03715684 0.03715684 0.03715684 0.03715684 0.03715684 0.03715684]]
```

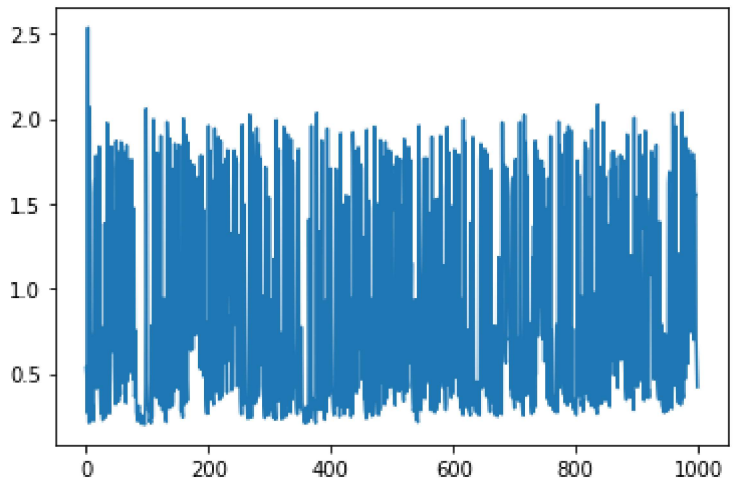

```
sgd_l2=w_n_l2[-1:]
print(sgd_l2)
```

```
[[0.03715684 0.03715684 0.03715684 0.03715684 0.03715684 0.03715684]]
```

MiniBatch Gradient Descent

```
alpha=0.005
iters=1000 ##iterations
lamb=3
batch_size=25
print(ztr.shape)
print(w1.shape)
mb_w,J_mb = MB_gradient_descent(ztr,y,w1,alpha, iters, batch_size)

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

```
(55, 6)
(6,)
```

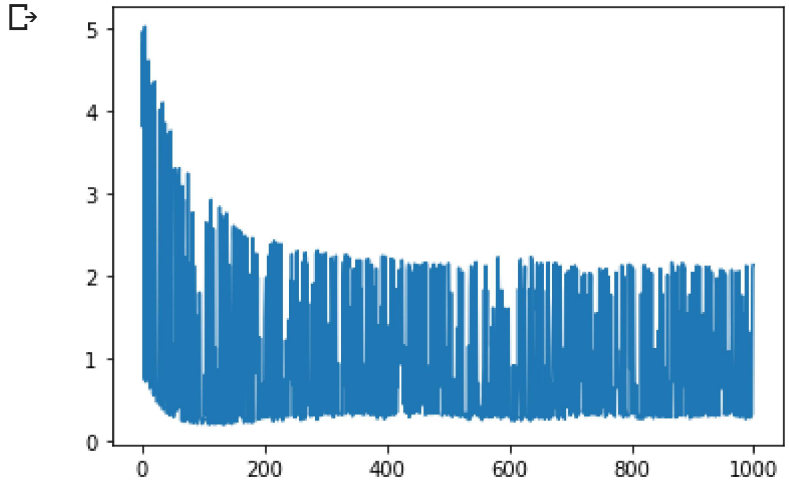



```
print(mb_w)
```

```
[ 0.01800889  0.23203559  0.34295655 -0.13013481  0.0745416  -0.02518848]
```

```
alpha=0.005
```

```
iters=1000
lamb=0.005
batch_size=30
mb_w_l1,J_mb_l1 = MB_gradient_descent_l1(ztr,y,w1,alpha, iters, batch_size,lamb)

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




```
print(mb_w_l1)
```

```
[[ 0.23105754]
 [ 0.04614605]
 [ 0.08808821]
 [-0.00482229]
 [ 0.02160498]
 [ 0.0003552 ]]
```

```
# alpha=0.001
# iters=1000 ###iterations
# lamb=0.005
# batch_size=25
# print(w1.shape)
# print(ztr.shape)
# mb_w_l2,J_mb_l2 = MB_gradient_descent_l2(ztr,y,w1,alpha, iters, batch_size,lamb)

# plt.plot(range(iters),J_mb_l2)
# plt.show()


# print(mb_w_l2)
```

Performance Measures for Regression(Self Defined functions)

Have defined these functions but have not used them because sklearn libraries were allowed to be used which I have used to find the three errors.

Mean Absolute Error

```
def mean_abs_error(Ypre,Yact):
  sum_err=abs(Yact - Ypre)
  ma_err=sum_err/Ypre.shape[0]
  return ma_err
```

Mean Square Error

```
def mean_square_error(Ypre,Yact):
  sum_error=((Yact - Ypre)**2)
  ms_err=sum_error/Ypre.shape[0]
  return ms_err
```

Correlation Coefficient

```
def correlation_coeff(test_instances,Ypred,Yact):
  ypm=np.mean(Ypred)##mean of Ypred data
  yam=np.mean(Yact)##mean of Yactual data
  num=((Yact - yam)*(Ypred-ypm))
  d1=pow(((Yact - yam)**2),1/2)
  d2=pow(((Ypred - ypm)**2),1/2)
  c_c=num/(d1*d2)
  return c_c
```

Finding Errors

Using sklearn libraries(was allowed)

```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import matthews_corrcoef


Xte1=Xte[:,0].reshape(Xte.shape[0],1)
Xte2=Xte[:,1].reshape(Xte.shape[0],1)
k=Xte.shape[0]
print(Xte2)
# print(Xte2.shape)
```

```
     [[8.58]
      [9.08]
      [9.58]
      [8.08]
      [7.58]
      [8.58]
      [9.08]
      [9.58]
```

```
       [8.08]
       [7.58]]


# Xte1=np.append(Xte1,Xte1,axis=1)

# Xte1=np.append(Xte1,Xte2,axis=1)
Xte1=np.append(Xte,Xte1**2,axis=1)
Xte1=np.append(Xte,Xte2**2,axis=1)
Xte1=np.append(Xte,Xte1*Xte2,axis=1)
Xte1=np.append(np.ones([k,1]),Xte1,axis=1)
print(Xte1)
```

```
     [[  1.        17.4       8.58     149.292     73.6164   631.628712]
      [  1.        17.86      9.08     162.1688    82.4464   748.613312]
      [  1.        18.3       9.58     175.314     91.7764   879.217912]
      [  1.        16.97      8.08     137.1176    65.2864   527.514112]
      [  1.        16.46      7.58     124.7668    57.4564   435.519512]
      [  1.        17.4       8.58     149.292     73.6164   631.628712]
      [  1.        17.85      9.08     162.078     82.4464   748.613312]
      [  1.        18.3       9.58     175.314     91.7764   879.217912]
      [  1.        16.9       8.08     136.552     65.2864   527.514112]
      [  1.        16.42      7.58     124.4636    57.4564   435.519512]]
```

```
Xte1.shape
print(Xte1)
print(bgd.T)
```

```
     [[  1.        17.4       8.58     149.292     73.6164   631.628712]
      [  1.        17.86      9.08     162.1688    82.4464   748.613312]
      [  1.        18.3       9.58     175.314     91.7764   879.217912]
      [  1.        16.97      8.08     137.1176    65.2864   527.514112]
      [  1.        16.46      7.58     124.7668    57.4564   435.519512]
      [  1.        17.4       8.58     149.292     73.6164   631.628712]
      [  1.        17.85      9.08     162.078     82.4464   748.613312]
      [  1.        18.3       9.58     175.314     91.7764   879.217912]
      [  1.        16.9       8.08     136.552     65.2864   527.514112]
      [  1.        16.42      7.58     124.4636    57.4564   435.519512]]
     [[0.12040141]
      [0.06170125]
      [0.06961951]
      [0.04162253]
      [0.04871827]
      [0.04190613]]
```

```
y_pred_bgd=Xte1.dot(bgd.T)
print(mean_squared_error(y_pred_bgd, Yte))
print(mean_absolute_error(y_pred_bgd, Yte))
# print(matthews_corrcoef(y_pred_bgd, Yte))


y_pred_bgd_l1=Xte1.dot(bgd_l1.T)
print(mean_squared_error(y_pred_bgd_l1, Yte))
print(mean_absolute_error(y_pred_bgd_l1, Yte))
# print(matthews_corrcoef(y_pred_bgd_l1, Yte))
```

```
y_pred_bgd_l2=Xte1.dot(bgd_l2.T)
print(mean_squared_error(y_pred_bgd_l2, Yte))
print(mean_absolute_error(y_pred_bgd_l2, Yte))
# print(matthews_corrcoef(y_pred_bgd_l2, Yte))
```

```
1451.208834920268
37.242967368074865
1475.7561322470854
37.55691224167781
1191.460677421232
33.740143694028866
```

Stochastic Gradient descent

```
y_pred_sgd=Xte1.dot(sgd.T)
print(mean_squared_error(y_pred_sgd, Yte))
print(mean_absolute_error(y_pred_sgd, Yte))
# print(matthews_corrcoef(y_pred_sgd, Yte))

y_pred_sgd_l1=Xte1.dot(sgd_l1.T)
print(mean_squared_error(y_pred_sgd_l1, Yte))
print(mean_absolute_error(y_pred_sgd_l1, Yte))
# print(matthews_corrcoef(y_pred_sgd_l1, Yte))

y_pred_sgd_l2=Xte1.dot(sgd_l2.T)
print(mean_squared_error(y_pred_sgd_l2, Yte))
print(mean_absolute_error(y_pred_sgd_l2, Yte))
# print(matthews_corrcoef(y_pred_sgd_l2, Yte))
```

```
359.48963297053274
18.491852086462014
921.53602271369
29.644654777000063
1064.669590639896
31.86843736823422
```

✓ 0s completed at 8:10 AM

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

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.m
```

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

```
pip install openpyxl==3.0.9
```

```
Requirement already satisfied: openpyxl==3.0.9 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: et-xmlfile in /usr/local/lib/python3.7/dist-packages (
```

Training Data

```
dataxtr = pd.read_excel('/content/drive/MyDrive/Q3_data/Xtr.xlsx',header=None)
dataytr = pd.read_excel('/content/drive/MyDrive/Q3_data/Ytr.xlsx',header=None)
```

Testing Data

```
dataxte = pd.read_excel('/content/drive/MyDrive/Q3_data/Xte.xlsx',header=None)
datayte = pd.read_excel('/content/drive/MyDrive/Q3_data/Yte.xlsx',header=None)
```

```
datanX=dataxtr.values
X=datanX[:,0:3] #feature matrix
datanY=dataytr.values
y=datanY[:,:]
```

```
datanXte=dataxte.values
Xte=datanXte[:,0:3] #feature matrix
datanYt=datayte.values
Yte=datanYt[:,:]
```

Normalization

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

```
# print(X)


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


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

Logistical Regression

```
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))) ####as mention i
  return J


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 MB_gradient_descent(X,y,w,alpha, iters, batch_size):
  cost_history = np.zeros(iters)
  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(sigmoid(ind_x.dot(w)) - ind_y))
    cost_history[i] = cost_function(ind_x,ind_y,w)
  return w, cost_history


def stochastic_gradient_descent(X,y,w,alpha, iters):
  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 - alpha * (ind_x.T.dot(sigmoid(ind_x.dot(w)) - ind_y))
    cost_history[i] = cost_function(ind_x,ind_y,w)
    return w,cost_history
```

Logistical Regression with L1 norm regularization

```
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.si
  return J


def batch_gradient_descent_l1(X,y,w,alpha,iters,lamb):
  cost_history = np.zeros(iters) # cost function for each iteration
  for i in range(iters):
    hypothesis = sigmoid(np.dot(X,w.T))
    w = w - (alpha/len(y)) * np.dot(hypothesis - y, X)- (lamb/2)*np.sign(w)
    cost_history[i] = cost_function(X,y,w)
  return w,cost_history


##doubtful

def MB_gradient_descent_l1(X,y,w,alpha, iters, batch_size):
  cost_history = np.zeros(iters)
  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(sigmoid(ind_x.dot(w)) - ind_y))
    cost_history[i] = cost_function(ind_x,ind_y,w)
  return w, cost_history




w= np.zeros(X.shape[1]) ##weight initialization
#w=[0.5, 0.5, 0.5]
print(w)

     [0. 0. 0. 0.]


alpha=0.0002 ##learning rate
iters=1000 ###iterations
batch_w,J_his = batch_gradient_descent(X,y,w,alpha,iters)
plt.plot(range(iters),J_his)
plt.show()
```

```
# alpha=0.02
# iters=2000
# batch_size=25
# mini_batch_w,J_mini_batch = MB_gradient_descent(X,y,w,alpha,iters, batch_size)
# plt.plot(range(iters),J_mini_batch)
# plt.show()


n_epochs=2000
alpha=0.02
w_n,J_sgd = stochastic_gradient_descent(X,y,w, alpha, n_epochs)
plt.plot(range(n_epochs),J_sgd)
plt.show()
```

```
z_btch = np.dot(X, batch_w.T)
# z_mbtch = np.dot(X, mini_batch_w.T)
z_ = np.dot(X, w_n.T)
# print(z_btch)
# print(z_)


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

```
def sensitivity(cm):
  tp=cm[1][1]
  tn=cm[0][0]
  fp=cm[0][1]
```

```
    fn=cm[1][0]
    se=tp/(tp+fn)
    return se


def specificity(cm):
    tp=cm[1][1]
    tn=cm[0][0]
    fp=cm[0][1]
    fn=cm[1][0]
    sp=tn/(tn+fp)
    return sp


def accuracy(cm):
    tp=cm[1][1]
    tn=cm[0][0]
    fp=cm[0][1]
    fn=cm[1][0]
    ac=(tp+tn)/(tp+tn+fp+fn)
    return ac


def precision(cm):
    tp=cm[1][1]
    tn=cm[0][0]
    fp=cm[0][1]
    fn=cm[1][0]
    pr=tp/(tp+fp)
    return pr
```

Executing (33s) Cell › batch_gradient_descent() › sigmoid()

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

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

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

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

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

```
XTr=dataxtr.values
```

```
print(XTr)# class_label=dataxtr[:,60]
print(dataxtr[:,60])
```

```
[[242.75152634 281.80129772 250.03240457 ...   1.28197209   1.8448603
    1.        ]
 [216.58595112 297.05788313 300.9384782  ...   1.59858185   1.62539544
    1.        ]
 [265.73553593 339.27113445 269.81730525 ...   1.84815977   1.70613365
    1.        ]
 ...
 [446.22719796 219.93690991 181.60575345 ...   1.41851014   1.77229548
    4.        ]
 [511.40643675 215.37971012 170.59895734 ...   1.60929772   1.66489041
    4.        ]
 [757.96751558 268.43124317 189.75528024 ...   1.79003949   1.56063308
    4.        ]]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-17e90f722fc2> in <module>()
      1 print(XTr)# class_label=dataxtr[:,60]
----> 2 print(dataxtr[:,60])

                                3 frames
/usr/local/lib/python3.7/dist-packages/pandas/_libs/index.pyx in
pandas._libs.index.IndexEngine.get_loc()

TypeError: '(slice(None, None, None), 60)' is an invalid key

SEARCH STACK OVERFLOW
```

```
X=XTr[:,0:59]
print(X)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-a463093f3181> in <module>()
```

```
----> 1 X=XTr[:,0:59]
      2 print(X)

NameError: name 'XTr' is not defined
```

SEARCH STACK OVERFLOW

0s completed at 3:06 AM

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

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

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

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

```
not able to complete
```

not able to complete

Google colab links:

Assignment 1	https://colab.research.google.com/drive/1XVoBHPdoOF0r0QGmQa3ZTd8MOVsTA4-9?usp=sharing
Assignment 2	https://colab.research.google.com/drive/1JuICXfmId0oftSqwtfYCni_qlUFL8P1b?usp=sharing
Assignment 3	https://colab.research.google.com/drive/1npJfpEV-tWUrno7Mbx6D-zY72izxRlol?usp=sharing
Assignment 4	https://colab.research.google.com/drive/1-px4IjgIIVx_qNm4DQ9pUCbRRhdGGcZn?usp=sharing
Assignment 5	https://colab.research.google.com/drive/1KGqo5vb8M_ASmYQSGu73FiH2PMcc_Q7R?usp=sharing