```
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.