# import necessary libraries  
import csv  
import time  
  
# import time  
import numpy as np  
  
# import statistics  
#  
import math  
  
# import operator  
import matplotlib.pyplot as plt  
  
ADataPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/A.csv"  
APDataPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/AP.csv"  
AODataPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/AO.csv"  
USPSDataPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/usps.csv"  
ALabelPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/labels-A.csv"  
APLabelPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/labels-AP.csv"  
AOLabelPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/labels-AO.csv"  
USPSLabelPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/labels-usps.csv"  
IRLSDataPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/irlstest.csv"  
IRLSLabelPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/labels-irlstest.csv"  
  
  
class GLM:  
 # read data  
 def readData(self, path):  
 file = open(path)  
 csvReader = csv.reader(file)  
 data = []  
 for row in csvReader:  
 data.append(row)  
 file.close()  
 return data  
  
 # add intercept column in the beginning of the matrix  
 def addIntercept(self, data):  
 newData = np.concatenate((np.ones((len(data), 1)), data), axis=1)  
 return newData  
  
 # calculate first derivative vector g  
 def calculateG(self, phiT, d, alpha, w):  
 g = np.array(phiT @ d - alpha \* w)  
 return g  
  
 # calculate second derivative matrix H  
 def calculateHInverse(self, phiT, phi, R, alpha):  
 # I for H matrix  
 I = np.eye(len(((-phiT @ R) @ phi)))  
 H = -((phiT @ R) @ phi) - (alpha \* I)  
 HInverse = np.linalg.inv(H)  
 return HInverse  
  
 # sigmoid  
 def sigmoid(self, x):  
 return 1 / (1 + np.exp(-x))  
  
 # Logistic regression function to calculate d and R matrices  
 def logisticReg(self, phi, t, w):  
 # print('wT shape:' + str(wT.shape))  
  
 wT\_phi = phi @ w  
  
 # print('wT\_phi shape' + str(wT\_phi.shape))  
  
 y = np.array(self.sigmoid(wT\_phi))  
 # print('y shape' + str(y.shape))  
  
 # d vector  
 # print(y)  
 d = np.array(t.reshape(-1, ) - y)  
 # print('d shape' + str(d.shape))  
  
 # R matrix  
 R = y \* (1 - y)  
 # print('R shape before diagonal' + str(R.shape))  
 R = np.diag(R)  
 # print('R shape after diagonal' + str(R.shape))  
  
 return [d, R]  
  
 # poisson data function to calculate d and R matrices  
 def poissonReg(self, phi, t, w):  
  
 # print('t shape: '+str(t.shape))  
 wT\_phi = phi @ w  
 # print('wT\_phi shape' + str(wT\_phi.shape))  
  
 y = np.array(np.exp(wT\_phi))  
 # print('y shape' + str(y.shape))  
  
 # d vector  
 d = t.reshape(-1, ) - y  
 # print('d shape' + str(d.shape))  
  
 # R matrix  
 R = y  
 # print('R shape before diagonal' + str(R.shape))  
 R = np.diag(R)  
 # print('R shape after diagonal' + str(R.shape))  
  
 return [d, R]  
  
 # ordinal data function to calculate d and R matrices  
 def ordinalReg(self, phi, t, w):  
 s = 1  
 k = 5  
 a = phi @ w  
 y = np.empty((len(t), k + 1))  
 ordinals\_phi = [-np.inf, -2, -1, 0, 1, np.inf]  
  
 # creating y matrix  
 for i in range(len(a)):  
 for j in range(6):  
 X = s \* (ordinals\_phi[j] - a[i])  
 yij = self.sigmoid(X)  
 y[i, j] = yij  
  
 d = np.empty(len(t), )  
 R = np.empty(len(t), )  
  
 # filling up d and R  
 for i in range(y.shape[0]):  
 yi\_j = y[i, int(t[i])]  
 yi\_jMinus1 = y[i, int(t[i]) - 1]  
 d[i] = yi\_j + yi\_jMinus1 - 1  
 R[i] = s \*\* 2 \* ((yi\_j \* (1 - yi\_j)) + (yi\_jMinus1 \* (1 - yi\_jMinus1)))  
  
 # # print('R shape before diagonal' + str(R.shape))  
 R = np.diag(R)  
 # print('R shape after diagonal' + str(R.shape))  
  
 return [d, R]  
  
 # implement Newton-Raphson method and update weight vectors  
 def MyGLM(self, phi, phiT, t, alpha, w\_old, counter, distribution):  
  
 if distribution == "Logistic":  
 dRList = self.logisticReg(phi, t, w\_old)  
 d = dRList[0]  
 R = dRList[1]  
  
 g = self.calculateG(phiT, d, alpha, w\_old)  
 # print('g vector shape: ' + str(g.shape))  
 HInverse = self.calculateHInverse(phiT, phi, R, alpha)  
 # print('h vector shape: ' + str(HInverse.shape))  
 w\_new = w\_old - (HInverse @ g)  
  
 while (np.linalg.norm((w\_new - w\_old), 2)) / ((np.linalg.norm(w\_old, 2) + 1e-10)) > 0.001 or counter <100:  
 dRList = self.logisticReg(phi, t, w\_old)  
 d = dRList[0]  
 R = dRList[1]  
 g = self.calculateG(phiT, d, alpha, w\_old)  
 HInverse = self.calculateHInverse(phiT, phi, R, alpha)  
 w\_new = w\_old - (HInverse @ g)  
 w\_old = w\_new  
 counter += 1  
 return w\_new  
  
  
  
 elif distribution == "Poisson":  
  
 dRList = self.poissonReg(phi, t, w\_old)  
  
 d = dRList[0]  
  
 R = dRList[1]  
  
 g = self.calculateG(phiT, d, alpha, w\_old)  
  
 # print('g vector shape: ' + str(g.shape))  
  
 HInverse = self.calculateHInverse(phiT, phi, R, alpha)  
  
 w\_new = w\_old - (HInverse @ g)  
  
 while (np.linalg.norm((w\_new - w\_old), 2)) / ((np.linalg.norm(w\_old, 2) + 1e-10)) > 0.001 or counter <100:  
 dRList = self.poissonReg(phi, t, w\_old)  
  
 d = dRList[0]  
  
 R = dRList[1]  
  
 g = self.calculateG(phiT, d, alpha, w\_old)  
  
 HInverse = self.calculateHInverse(phiT, phi, R, alpha)  
  
 w\_new = w\_old - (HInverse @ g)  
  
 w\_old = w\_new  
  
 counter += 1  
  
 return w\_new  
  
  
  
 elif distribution == "Ordinal":  
 dRList = self.ordinalReg(phi, t, w\_old)  
 d = dRList[0]  
 R = dRList[1]  
  
 g = self.calculateG(phiT, d, alpha, w\_old)  
 # print('g vector shape: ' + str(g.shape))  
 HInverse = self.calculateHInverse(phiT, phi, R, alpha)  
 w\_new = w\_old - (HInverse @ g)  
  
 while (np.linalg.norm((w\_new - w\_old), 2)) / ((np.linalg.norm(w\_old, 2) + 1e-10)) > 0.001 or counter <100:  
 dRList = self.ordinalReg(phi, t, w\_old)  
 d = dRList[0]  
 R = dRList[1]  
 g = self.calculateG(phiT, d, alpha, w\_old)  
 HInverse = self.calculateHInverse(phiT, phi, R, alpha)  
 w\_new = w\_old - (HInverse @ g)  
 # print(w\_new[1])  
 w\_old = w\_new  
 counter += 1  
 return w\_new  
  
 def calculateError\_Logistic(self, Wmap, phi, t):  
 Wmap\_phi = phi @ Wmap  
 probOfT1 = np.array(self.sigmoid(Wmap\_phi))  
 # print(probOfT1)  
 t\_hat = []  
 for i in range(len(probOfT1)):  
 if probOfT1[i] >= 0.5:  
 t\_hat.append(1)  
 else:  
 t\_hat.append(0)  
 t\_hat = np.array(t\_hat).astype(int)  
 t = t.astype(int)  
  
 # print(t\_hat)  
 # print(t.reshape(-1,))  
  
 error = []  
 for i in range(len(t\_hat)):  
 if t\_hat[i] == int(t[i]):  
 error.append(0)  
 else:  
 error.append(1)  
 # print(np.mean(error))  
  
 return error  
  
 def calculateError\_Poisson(self, Wmap, phi, t):  
 a = phi @ Wmap  
 lambdaa = np.exp(a)  
  
 t\_hat = []  
 for i in range(len(lambdaa)):  
 t\_hat.append(np.floor(lambdaa[i]))  
  
 t\_hat = np.array(t\_hat)  
 # print(t\_hat)  
 t = t.astype(int)  
  
 error = []  
 for i in range(len(t\_hat)):  
 error.append(int(abs(t\_hat[i] - t[i])))  
  
 return error  
  
 def calculateError\_Ordinal(self, Wmap, phi, t):  
 a = phi @ Wmap  
  
 s = 1  
 k = 5  
 y = np.empty((len(t), k + 1))  
 p = np.empty((len(t), k))  
 ordinals\_phi = [-np.inf, -2, -1, 0, 1, np.inf]  
  
 for i in range(len(a)):  
 for j in range(6):  
 X = s \* (ordinals\_phi[j] - a[i])  
 yij = self.sigmoid(X)  
 y[i, j] = yij  
  
 for k in range(1, len(y[i])):  
 p[i, k - 1] = y[i, k] - y[i, (k - 1)]  
 p\_max = p.argmax(axis=1)  
 t\_hat = []  
 t\_hat.append(p\_max)  
 # print(t\_hat)  
 t = t.astype(int)  
  
 error = []  
 for i in range(len(t\_hat)):  
 error.append(abs(t\_hat[i] - t[i]))  
 return error  
  
  
g = GLM()  
  
# alpha  
alpha = 10  
  
# counter  
counter = 0  
  
start = time.time()  
  
# =======================================================LOGISTIC REGRESSION=============================================  
  
  
# phi(x)  
phi = np.array(g.readData(ADataPath))  
phi = phi.astype(np.float64) # converting elements into type float for matrix multiplication  
  
# add intercept column inn data  
phi = g.addIntercept(phi)  
# print('phi shape:' + str(phi.shape))  
# print(phi)  
  
# t  
t = np.array(g.readData(ALabelPath)).astype(np.float64)  
# print('t shape:' + str(t.shape))  
  
noOfIterations = 1  
  
TotalMeanErrors\_30 = []  
  
for i in range(30):  
 dummy = np.c\_[t, phi]  
 # print(dummy.shape)  
 np.random.shuffle(dummy)  
 x = dummy[:, 1:dummy.shape[1]]  
 # print(x.shape)  
 y = dummy[:, 0:1]  
 # print(y.shape)  
  
 train\_phi = x[:int(2 \* len(x) / 3)]  
 train\_t = y[:int(2 \* len(y) / 3)]  
 test\_phi = x[int(2 \* len(y) / 3):]  
 test\_t = y[int(2 \* len(y) / 3):]  
 # print(train\_phi.shape)  
 # print(train\_t.shape)  
 # print(test\_phi.shape)  
 # print(test\_t.shape)  
  
 train\_data\_size = np.arange(0.1, 1.1, 0.1)  
 error = []  
 errorList = []  
 for s in train\_data\_size:  
 segmented\_Train\_Phi = train\_phi[:int(s \* len(train\_phi))]  
 segmented\_Train\_t = train\_t[:int(s \* len(train\_t))]  
 segmented\_Train\_Phi\_Transpose = segmented\_Train\_Phi.T  
 w = np.zeros([len(segmented\_Train\_Phi\_Transpose)])  
 Wmap = g.MyGLM(segmented\_Train\_Phi, segmented\_Train\_Phi\_Transpose, segmented\_Train\_t, alpha, w, counter,  
 "Logistic")  
  
 # calculation of error  
 error = g.calculateError\_Logistic(Wmap, test\_phi, test\_t)  
 errorList.append(error)  
  
 MeanErrorList\_innerLoop = []  
 StdList = []  
 for i in range(len(errorList)):  
 mean = sum(errorList[i]) / len(errorList[i])  
 MeanErrorList\_innerLoop.append(mean)  
 variance = sum([((x - MeanErrorList\_innerLoop[i]) \*\* 2) for x in MeanErrorList\_innerLoop]) / len(  
 MeanErrorList\_innerLoop)  
 std = variance \*\* 0.5  
 # std = np.std(errorList[i])  
 StdList.append(std)  
  
end = time.time()  
print('Average No of Iterations ' + str(noOfIterations \* 85))  
print("Runtime: " + str(end - start) + " seconds")  
  
plt.xlabel("DataSize")  
plt.ylabel("Mean Error")  
plt.title("Logistic Regression")  
plt.errorbar(train\_data\_size, MeanErrorList\_innerLoop, StdList)  
plt.show()  
  
# ======================================================POISSON REGRESSION==============================================  
#  
# phi(x)  
# phi = np.array(g.readData(APDataPath))  
# phi = phi.astype(np.float64) # converting elements into type float for matrix multiplication  
#  
# # add intercept column inn data  
# phi = g.addIntercept(phi)  
# # print('phi shape:' + str(phi.shape))  
#  
#  
# # t  
# t = np.array(g.readData(APLabelPath)).astype(int).reshape(-1, )  
# # print('t shape:' + str(t.shape))  
# # print(t)  
#  
# for i in range(30):  
# dummy = np.c\_[t, phi]  
# # print(dummy.shape)  
# np.random.shuffle(dummy)  
# x = dummy[:, 1:dummy.shape[1]]  
# # print(x.shape)  
# y = dummy[:, 0:1]  
# # print(y.shape)  
#  
# train\_phi = x[:int(2 \* len(x) / 3)]  
# train\_t = y[:int(2 \* len(y) / 3)]  
# test\_phi = x[int(2 \* len(y) / 3):]  
# test\_t = y[int(2 \* len(y) / 3):]  
# # print(train\_phi.shape)  
# # print(train\_t.shape)  
# # print(test\_phi.shape)  
# # print(test\_t.shape)  
#  
# train\_data\_size = np.arange(0.1, 1.1, 0.1)  
# # print(train\_data\_size)  
# errorList = []  
# for s in train\_data\_size:  
# segmented\_Train\_Phi = train\_phi[:int(s \* len(train\_phi))]  
# segmented\_Train\_t = train\_t[:int(s \* len(train\_t))]  
# segmented\_Train\_Phi\_Transpose = segmented\_Train\_Phi.T  
# w = np.zeros([len(segmented\_Train\_Phi\_Transpose)])  
# Wmap = g.MyGLM(segmented\_Train\_Phi, segmented\_Train\_Phi\_Transpose, segmented\_Train\_t, alpha, w, counter,  
# "Poisson")  
#  
# # calculation of error  
# error = g.calculateError\_Poisson(Wmap, test\_phi, test\_t)  
# errorList.append(error)  
#  
# MeanErrorList\_innerLoop = []  
# StdList = []  
# for i in range(len(errorList)):  
# mean = sum(errorList[i]) / len(errorList[i])  
# MeanErrorList\_innerLoop.append(mean)  
# variance = sum([((x - MeanErrorList\_innerLoop[i]) \*\* 2) for x in MeanErrorList\_innerLoop]) / len(  
# MeanErrorList\_innerLoop)  
# res = variance \*\* 0.5  
# StdList.append(res)  
#  
# end = time.time()  
# print("Runtime: " + str(end - start) + " seconds")  
#  
# plt.xlabel("DataSize")  
# plt.ylabel("Mean Error")  
# plt.title("Poisson Regression")  
# plt.errorbar(train\_data\_size, MeanErrorList\_innerLoop, StdList)  
# plt.show()  
  
  
# =====================================================ORDINAL REGRESSION===============================================  
#  
# #phi(x)  
# phi = np.array(g.readData(AODataPath))  
# phi = phi.astype(np.float64) # converting elements into type float for matrix multiplication  
#  
# # add intercept column inn data  
# phi = g.addIntercept(phi)  
# # print('phi shape:' + str(phi.shape))  
#  
# phiT = phi.T  
# w = np.zeros([len(phiT)])  
# # print('w shape ' + str(w.shape))  
#  
# # t  
# t = np.array(g.readData(AOLabelPath)).astype(int).reshape(-1, )  
# # print('t shape:' + str(t.shape))  
# # print(t)  
#  
# noOfIterations = 1  
#  
# for i in range(30):  
# dummy = np.c\_[t, phi]  
# # print(dummy.shape)  
# np.random.shuffle(dummy)  
# x = dummy[:, 1:dummy.shape[1]]  
# # print(x.shape)  
# y = dummy[:, 0:1]  
# # print(y.shape)  
#  
# train\_phi = x[:int(2 \* len(x) / 3)]  
# train\_t = y[:int(2 \* len(y) / 3)]  
# test\_phi = x[int(2 \* len(y) / 3):]  
# test\_t = y[int(2 \* len(y) / 3):]  
# # print(train\_phi.shape)  
# # print(train\_t.shape)  
# # print(test\_phi.shape)  
# # print(test\_t.shape)  
#  
# train\_data\_size = np.arange(0.1, 1.1, 0.1)  
# # print(train\_data\_size)  
# errorList = []  
# for s in train\_data\_size:  
# segmented\_Train\_Phi = train\_phi[:int(s \* len(train\_phi))]  
# segmented\_Train\_t = train\_t[:int(s \* len(train\_t))]  
# segmented\_Train\_Phi\_Transpose = segmented\_Train\_Phi.T  
# w = np.zeros([len(segmented\_Train\_Phi\_Transpose)])  
# Wmap = g.MyGLM(segmented\_Train\_Phi, segmented\_Train\_Phi\_Transpose, segmented\_Train\_t, alpha, w, counter,  
# "Ordinal")  
# # print(Wmap[i])  
# # calculation of error  
# error = g.calculateError\_Poisson(Wmap, test\_phi, test\_t)  
# errorList.append(error)  
#  
# MeanErrorList\_innerLoop = []  
# StdList = []  
# for i in range(len(errorList)):  
# mean = sum(errorList[i]) / len(errorList[i])  
# MeanErrorList\_innerLoop.append(mean)  
# variance = sum([((x - MeanErrorList\_innerLoop[i]) \*\* 2) for x in MeanErrorList\_innerLoop]) / len(  
# MeanErrorList\_innerLoop)  
# res = variance \*\* 0.5  
# StdList.append(res)  
#  
# end = time.time()  
# print("Runtime: " + str(end - start) + " seconds")  
# print('Average No of Iterations ' + str(int(noOfIterations\*99)))  
# plt.xlabel("DataSize")  
# plt.ylabel("Mean Error")  
# plt.title("Ordinal Regression")  
# plt.errorbar(train\_data\_size,MeanErrorList\_innerLoop, StdList)  
# plt.show()