```python
# -*- coding: utf-8 -*-

import pandas as pd
import numpy as np
import sys
import random as rd

#insert an all-one column as the first column
def addAllOneColumn(matrix):
    n = matrix.shape[0] #total of data points
    p = matrix.shape[1] #total number of attributes

    newMatrix = np.zeros((n,p+1))
    newMatrix[:,0] = np.ones(n)
    newMatrix[:,1:] = matrix


    return newMatrix

# Reads the data from CSV files, converts it into Dataframe and returns x and y dataframes
def getDataframe(filePath):
    dataframe = pd.read_csv(filePath)
    y = dataframe['y']
    x = dataframe.drop('y', axis=1)
    return x, y

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

# compute average logL
def compute_avglogL(X,y,beta):
    eps = 1e-50
    n = y.shape[0]
    avglogL = 0
    #=======================#
    # STRART YOUR CODE HERE  #
    #=======================#
    total_logL = 0
    for i in range(n):
        # logL = yiXi^TB - log(1 + exp{xi^TB})
        XiT_beta = np.dot(np.transpose(X[i]), beta)
        left_oper = np.multiply(y[i], XiT_beta)
        right_oper = np.log(1 + np.exp(XiT_beta))
        total_logL += (left_oper - right_oper)
    avglogL = total_logL / n
    #=======================#
    #   END YOUR CODE HERE   #
    #=======================#
    return avglogL


# train_x and train_y are numpy arrays
# lr (learning rate) is a scalar
# function returns value of beta calculated using (0) batch gradient descent
def getBeta_BatchGradient(train_x, train_y, lr, num_iter, verbose):
    beta = np.random.rand(train_x.shape[1])

    n = train_x.shape[0] #total of data points
```

```
    p = train_x.shape[1] #total number of attributes


    beta = np.random.rand(p)
    #update beta interatively
    for iter in range(0, num_iter):
        #=======================#
        # STRART YOUR CODE HERE  #
        #=======================#

        """
        logL = yiXi^TB - log(1 + exp{xi^TB})
        deriv = sum yixij - sum x_ij exp{b^Txo}/ (1+exp{bTxi})
        """

        deriv = np.zeros(p)
        for j in range(p):
            jth_deriv = 0
            for i in range(n):
                betaT_xi = np.dot(beta, train_x[i])
                yi_xij = train_y[i] * train_x[i][j]
                pi_xij = train_x[i][j] * np.exp(betaT_xi) / (1 + np.exp(betaT_xi))
                jth_deriv = jth_deriv + yi_xij - pi_xij
            deriv[j] += jth_deriv
        beta = beta + np.multiply(lr, deriv)
        #=======================#
        #   END YOUR CODE HERE   #
        #=======================#
        if(verbose == True and iter % 1000 == 0):
            avgLogL = compute_avglogL(train_x, train_y, beta)
            print(f'average logL for iteration {iter}: {avgLogL} \t')
    return beta

# train_x and train_y are numpy arrays
# function returns value of beta calculated using (1) Newton-Raphson method
def getBeta_Newton(train_x, train_y, num_iter, verbose):
    n = train_x.shape[0] #total of data points
    p = train_x.shape[1] #total number of attributes

    beta = np.random.rand(p)
    for iter in range(0, num_iter):
        #=======================#
        # STRART YOUR CODE HERE  #
        #=======================#

        # calculate hessian matrix
        # -sum X_ij X_in p_i(beta)(1 - p_i(beta))
        hessian = np.zeros((p, p))
        for row in range(p):
            for col in range(p):
                for i in range(n):
                    betaT_xi = np.dot(beta, train_x[i])
                    pi_beta = np.exp(betaT_xi) / (1 + np.exp(betaT_xi))
                    hessian[row][col] -= (train_x[i][row] * train_x[i][col] * pi_beta
* (1 - pi_beta))

        # calculate first derivative same as getBeta_BatchGradient
        deriv = np.zeros(p)
        for j in range(p):
            jth_deriv = 0
```

```
            for i in range(n):
                betaT_xi = np.dot(beta, train_x[i])
                yi_xij = train_y[i] * train_x[i][j]
                pi_xij = train_x[i][j] * np.exp(betaT_xi) / (1 + np.exp(betaT_xi))
                jth_deriv = jth_deriv + yi_xij - pi_xij
            deriv[j] += jth_deriv

        beta = beta - np.matmul(np.linalg.inv(hessian), deriv)
        #=======================#
        #   END YOUR CODE HERE   #
        #=======================#
        if(verbose == True and iter % 500 == 0):
            avgLogL = compute_avglogL(train_x, train_y, beta)
            print(f'average logL for iteration {iter}: {avgLogL} \t')
    return beta



# Logistic Regression implementation
class LogisticRegression(object):
    # Initializes by reading data, setting hyper-parameters
    # Learns the parameter using (0) Batch gradient (1) Newton-Raphson
    # Performs z-score normalization if isNormalized is 1
    # Print intermidate training loss if verbose = True
    def __init__(self,lr=0.005, num_iter=10000, verbose = True):
        self.lr = lr
        self.num_iter = num_iter
        self.verbose = verbose
        self.train_x = pd.DataFrame()
        self.train_y = pd.DataFrame()
        self.test_x = pd.DataFrame()
        self.test_y = pd.DataFrame()
        self.algType = 0
        self.isNormalized = 0


    def load_data(self, train_file, test_file):
        self.train_x, self.train_y = getDataframe(train_file)
        self.test_x, self.test_y = getDataframe(test_file)

    def normalize(self):
        # Applies z-score normalization to the dataframe and returns a normalized
dataframe
        self.isNormalized = 1
        data = np.append(self.train_x, self.test_x, axis = 0)
        means = data.mean(0)
        std = data.std(0)
        self.train_x = (self.train_x - means).div(std)
        self.test_x = (self.test_x - means).div(std)

    # Gets the beta according to input
    def train(self, algType):
        self.algType = algType
        newTrain_x = addAllOneColumn(self.train_x.values) #insert an all-one column
as the first column
        if(algType == '0'):
            beta = getBeta_BatchGradient(newTrain_x, self.train_y.values, self.lr,
self.num_iter, self.verbose)
            #print('Beta: ', beta)

```

```
176         elif(algType == '1'):
177             beta = getBeta_Newton(newTrain_x, self.train_y.values, self.num_iter,
    self.verbose)
178             #print('Beta: ', beta)
179         else:
180             print('Incorrect beta_type! Usage: 0 - batch gradient descent, 1 -
    Newton-Raphson method')
181 
182         train_avglogL = compute_avglogL(newTrain_x, self.train_y.values, beta)
183         print('Training avgLogL: ', train_avglogL)
184 
185         return beta
186 
187     # Predict on given data x with learned parameter beta
188     def predict(self, x, beta):
189         newTest_x = addAllOneColumn(x)
190         self.predicted_y = (sigmoid(newTest_x.dot(beta))>=0.5)
191         return self.predicted_y
192 
193     # predicted_y and y are the predicted and actual y values respectively as numpy
    arrays
194     # function returns the accuracy
195     def compute_accuracy(self,predicted_y, y):
196         acc = np.sum(predicted_y == y)/predicted_y.shape[0]
197         return acc
198 
199 
```