```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  #
    #========================#
    for i in range(n):

        x_transpose = np.transpose(X[i])
        x_transpose_dot_beta = np.dot(x_transpose, beta)
        first_term = y[i] * x_transpose_dot_beta
        second_term = 1 + np.exp(x_transpose_dot_beta)
        avglogL += first_term - np.log(second_term)

    avglogL = avglogL/ 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  #
        #=========================#
        for i in range(n):

            beta_transpose_dot_x = np.dot(np.transpose(beta), train_x[i])
            sigmoid_res = sigmoid(beta_transpose_dot_x)
            diff = train_y[i] - sigmoid_res
            gradient = np.dot(diff, train_x[i])
            beta +=  gradient * lr
        #=========================#
        #   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)
    ########## Please Fill Missing Lines Here ##########
    for iter in range(0, num_iter):
        #=========================#
        # STRART YOUR CODE HERE  #
        #=========================#
        beta_XT = np.dot(beta, np.transpose(train_x))
        sigmoid_res = sigmoid(beta_XT)
        diff = train_y - sigmoid_res
        # first deriv
        first_deriv = np.dot(diff, train_x)
        # second deriv
        prob_mul = sigmoid_res * (1 - sigmoid_res)
        x_mul = np.array([x*y for (x,y) in zip(train_x, prob_mul)])
        second_deriv = -1 * np.dot(np.transpose(x_mul), train_x)
        beta -= np.dot(np.linalg.inv(second_deriv), first_deriv)/n
        #=========================#
        #   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



# Linear Regression implementation
class LogisticRegression(object):
    # Initializes by reading data, setting hyper-parameters, and forming linear model
    # Forms a linear model (learns the parameter) according to type of beta (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)

        elif(algType == '1'):
            beta = getBeta_Newton(newTrain_x, self.train_y.values, self.num_iter, self.verbose)
            #print('Beta: ', beta)
        else:
            print('Incorrect beta_type! Usage: 0 - batch gradient descent, 1 - Newton-Raphson method')

        train_avglogL = compute_avglogL(newTrain_x, self.train_y.values, beta)
        print('Training avgLogL: ', train_avglogL)

        return beta

    # Predicts the y values of all test points
    # Outputs the predicted y values to the text file named "logistic-regression-output_algType_isNormalized" inside "output" folder
    def predict(self, x, beta):
        newTest_x = addAllOneColumn(x)
        self.predicted_y = (sigmoid(newTest_x.dot(beta))>=0.5)
        return self.predicted_y

    # predicted_y and y are the predicted and actual y values respectively as numpy arrays
    # function prints the accuracy
    def compute_accuracy(self,predicted_y, y):
        acc = np.sum(predicted_y == y)/predicted_y.shape[0]
        return acc

```