```python
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[:,1:] = matrix
    newMatrix[:,0] = np.ones(n)

    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

# train_x and train_y are numpy arrays
# function returns value of beta calculated using (0) the formula beta = (X^T*X)^ -1)*(X^T*Y)
def getBeta(train_x, train_y):
    n = train_x.shape[0] #total of data points
    p = train_x.shape[1] #total number of attributes

    beta = np.zeros(p)
    #========================#
    # STRART YOUR CODE HERE  #
    #========================#
    # 1. Calculate the transpose
    # 2. first_term = (X^T*X)^ -1)
    # 3. second_term = (X^T*Y)
    # 4. calculate beta = first_term * second_term
    train_x_transpose = np.transpose(train_x)
    first_term = np.linalg.inv(np.matmul(train_x_transpose, train_x))
    second_term = np.matmul(train_x_transpose, train_y)
    beta = np.matmul(first_term, second_term)
    #========================#
    #   END YOUR CODE HERE   #
    #========================#
    return beta

# train_x and train_y are numpy arrays
# lr (learning rate) is a scalar
# function returns value of beta calculated using (1) batch gradient descent
def getBetaBatchGradient(train_x, train_y, lr, num_iter):
    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)
```

```python
    for iter in range(0, num_iter):
       deriv = np.zeros(p)
       for i in range(n):
           #========================#
           # STRART YOUR CODE HERE  #
           #========================#
            x_i = train_x[i]
            y_i = train_y[i]
            x_i_transpose = np.transpose(x_i)
            A = np.matmul(x_i_transpose, beta)
            deriv += x_i * (A - y_i)
           #========================#
           #   END YOUR CODE HERE   #
           #========================#
       deriv = deriv / n
       beta = beta - deriv.dot(lr)
    return beta

# train_x and train_y are numpy arrays
# lr (learning rate) is a scalar
# function returns value of beta calculated using (2) stochastic gradient descent
def getBetaStochasticGradient(train_x, train_y, lr):
    n = train_x.shape[0] #total of data points
    p = train_x.shape[1] #total number of attributes

    beta = np.random.rand(p)

    epoch = 100;
    for iter in range(epoch):
        indices = list(range(n))
        rd.shuffle(indices)
        for i in range(n):
            idx = indices[i]
           #========================#
           # STRART YOUR CODE HERE  #
           #========================#
            y_i = train_y[idx]
            x_i = train_x[idx]
            x_i_transpose = np.transpose(x_i)
            A = np.matmul(x_i_transpose, beta)
            beta = beta + (lr * (y_i - A) * x_i)
           #========================#
           #   END YOUR CODE HERE   #
           #========================#
    return beta


# Linear Regression implementation
class LinearRegression(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 - closed form, 1 - batch gradient, 2 - stochastic gradient)
    # Performs z-score normalization if z_score is 1
    def __init__(self,lr=0.005, num_iter=1000):
        self.lr = lr
        self.num_iter = num_iter
        self.train_x = 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
        means = self.train_x.mean(0)
        std = self.train_x.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
        print('Learning Algorithm Type: ', algType)
        if(algType == '0'):
            beta = getBeta(newTrain_x, self.train_y.values)
            #print('Beta: ', beta)

        elif(algType == '1'):
            beta = getBetaBatchGradient(newTrain_x, self.train_y.values, self.lr, self.num_iter)
            #print('Beta: ', beta)
        elif(algType == '2'):
            self.lr = 0.0005  #lr to 0.0005 so that the beta does converge
            beta = getBetaStochasticGradient(newTrain_x, self.train_y.values, self.lr)
            #print('Beta: ', beta)
        else:
            print('Incorrect beta_type! Usage: 0 - closed form solution, 1 - batch gradient descent, 2 - stochastic gradient descent')


        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
    # Computes MSE
    def predict(self,x, beta):
        newTest_x = addAllOneColumn(x)
        self.predicted_y = newTest_x.dot(beta)
        return self.predicted_y


    # predicted_y and test_y are the predicted and actual y values respectively as numpy arrays
    # function prints the mean squared error value for the test dataset
    def compute_mse(self,predicted_y, y):
        mse = np.sum((predicted_y - y)**2)/predicted_y.shape[0]
```