```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  #
    #========================#

    x_transpose = np.transpose(train_x)
    left_operand = np.linalg.inv(np.matmul(x_transpose, train_x))
    right_operand = np.matmul(x_transpose, train_y)
    beta = np.matmul(left_operand, right_operand)

    #========================#
    #   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)
    #update beta interatively
    for iter in range(0, num_iter):
       deriv = np.zeros(p)
```

```python
       for i in range(n):
           #========================#
           # STRART YOUR CODE HERE  #
           #========================#

            # gradient of OLS(1/2 (yi-xi)^2)
            cur_deriv = train_x[i] * (np.dot(beta, np.transpose(train_x[i])) - train_y[i])
            deriv = np.add(deriv, cur_deriv)

           #========================#
           #   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  #
           #========================#

           # use np.multiply instead of * here to avoid overflow
           # needs to multiply lr first to avoid overflow
            coefficient = lr * (train_y[idx] - np.dot(np.transpose(train_x[idx]), beta))
            cur_update = np.multiply(coefficient, train_x[idx])
            beta = np.add(beta, cur_update)

           #========================#
           #   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.train_y = pd.DataFrame()
```

```python
        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'):
            # change learning rate to 0.0005 to converge
            beta = getBetaStochasticGradient(newTrain_x, self.train_y.values, 0.0005)
            #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 on given data and learned beta
    def predict(self,x, beta):
        newTest_x = addAllOneColumn(x)
        self.predicted_y = newTest_x.dot(beta)
        return self.predicted_y


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