```
"""
This file contains the classes that implement Linear Regression
"""
import numpy as np
from preprocessing import NormalScaler
import matplotlib.pyplot as plt
import pandas as pd
from mpl_toolkits.mplot3d import Axes3D

class LinearRegression:
    """
    This class implements Linear Regression using both
    Batch Gradient Descent and Stochastic Gradient descent.
    Class attributes:
      W        : current set of weights
      W_arr    : list of weights at each iteration
      Cost     : current cost
      cost_arr : list of costs at each iteration
    """
    def init_weights(self, s):
        """
        This method initializes the weight matrix
        as a column vector with shape = (X rows+1, 1)
        """
        np.random.seed(2)
        self.W = np.random.randn(s[1],1)
        self.W_arr = []
        self.cost_arr = []
        self.cost = 0

    def get_cost(self, X, y, W):
        """
        This function returns the cost with the given set of weights
        using the formula
             J = \frac{1}{2}\sum_{i=0}^{m}\left(h_\omega\left(x^i\right)-y^i\right)^2
        """
        total_cost = sum(np.square(np.matmul(X,W)-y.reshape(-1,1)))[0]
        return (0.5/X.shape[0])*total_cost

    def add_bias(self, X):
        """
        This function adds bias (a column of ones) to the feature vector X.
        """
        bias = np.ones((X.shape[0],1))
        return np.concatenate((bias,X), axis=1)

    def get_h_i(self, X, i, W):
        """
        This function returns the hypothesis of ith feature vector
        with the given weights W.
             h_w\left(x^i\right)=\sum_{j=0}^{n}w_j{x^i}_j=x^iw
        """
```

```
        h_i = 0
        h_i = np.matmul(X[i].reshape(1,-1),W)
        return h_i[0][0]

    def batch_grad_descent(self, X, y, alpha, max_iter):
        """
        This function implements the Batch Gradient Descent algorithm.
        It runs for multiple iterations until either the weights converge or
        iterations reach max_iter. At each iteration the weights are updated using
        the following rule
            repeat until convergence{
                    w_j^{t+1} = w_j^t - α Σ_{i=1}^{m} (h_w(x^i) - y^i) x_j^i
            }
        """
        W_new = self.W.copy()
        for iteration in range(max_iter):
            temp = np.matmul(X,self.W) - y.reshape(-1,1)
            for j in range(X.shape[1]):
                W_new[j][0] = self.W[j][0] - (alpha/X.shape[0])*(sum(temp*X[:,j:j+1])[0])
            self.W = W_new.copy()
            self.cost_arr.append(self.get_cost(X, y, self.W))
            self.W_arr.append(self.W)
        return W_new

    def stochastic_grad_descent(self, X, y, alpha, max_iter):
        """
        This function implements the Stochastic Gradient Descent algorithm.
        It runs for multiple iterations until either the weights converge or
        iterations reach max_iter. Weights are updated for every row of the
        training set.

            repeat until convergence{
                    for
                    w_j^{t+1} = w_j^t - α Σ_{i=1}^{m} (h_w(x^i) - y^i) x_j^i
            }
        """

        mat = np.concatenate((X,y.reshape(-1,1)), axis=1)
        for iteration in range(max_iter):
            W_new = self.W.copy()
            np.random.shuffle(mat)
            X = mat[:,0:3]
            y = mat[:,3]
            for i in range(X.shape[0]):
                temp = np.matmul(X[i,:],self.W) - y[i]
                for j in range(X.shape[1]):
                    W_new[j][0] = self.W[j][0] - (alpha)*(temp[0]*X[i,j])
                self.W = W_new.copy()
            self.cost_arr.append(self.get_cost(X, y, self.W))
            self.W_arr.append(self.W)
        return self.W


    def train(self, X, y, alpha, max_iter=100, option="batch"):
        X = self.add_bias(X)
```

```
        self.init_weights(X.shape)
        if option=="batch":
            self.batch_grad_descent(X,y,alpha,max_iter)
        elif option=="stochastic":
            self.stochastic_grad_descent(X,y,alpha,max_iter)
        self.cost = self.cost_arr[-1]
        return self.cost_arr

    def test(self,X,W=""):
        if W=="":W = self.W

        X = self.add_bias(X)
        y_pred = np.ones(X.shape[0])
        for i in range(X.shape[0]):
            for j in range(X.shape[1]):
                y_pred[i] += X[i][j]*W[j][0]
        return y_pred


if __name__ == "__main__":
    model = LinearRegression()

    data = pd.read_csv("./data.csv", header=None)
    X = data.loc[:,0:1].values
    y = data.loc[:,2].values
    mscaler = NormalScaler()
    mscaler.fit(X[:,0])
    X[:,0] = mscaler.transform(X[:,0])
    mscaler.fit(X[:,1])
    X[:,1] = mscaler.transform(X[:,1])
    arr = model.train(X,y,0.19,250,"batch")

    print("weights: ",model.W)
    print("Total Cost: ",model.cost)
    W_arr = np.array(model.W_arr)
    res = 100

    xx = np.linspace(np.min(W_arr[:,1])-10, np.max(W_arr[:,1])+10, res)
    yy = np.linspace(np.min(W_arr[:,2])-10, np.max(W_arr[:,2])+10, res)
    minw0 = W_arr[-1][0][0]

    r = np.ndarray((res,res))
    s = np.ndarray((res,res))
    z = np.ndarray((res,res))

    for i in range(res):
        for j in range(res):
            z[i][j] = model.get_cost(model.add_bias(X), y, np.array([minw0,xx[i],yy[j]]).r
eshape(-1,1))
            r[i][j] = xx[i]
            s[i][j] = yy[j]

    ax = plt.axes(projection='3d')
    ax.plot_surface(r, s, z,cmap='coolwarm')
    ax.plot(W_arr[:,1], W_arr[:,2], model.cost_arr,c='red')
    plt.show()
```

```
    plt.contour(r,s,z.reshape(res,res),levels=25)
    plt.scatter(W_arr[:,1].ravel(),W_arr[:,2].ravel(),c=model.cost_arr)
    plt.show()

    plt.plot(model.cost_arr)
    plt.show()
```