# Linear Regression

## Linear Regression

"""

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

"""

        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\_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\_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{

randomly shuffle the feature matrix rows

for each feature vector {

update all weights j -> 0 to n+1

}

}

        """

        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"):

"""

        This function initiates the training process.

It runs batch gradient descent by default and can also run

Stochastic gradient descent if the argument is passed.

returns the cost list which has costs at every training iteration.

        """

*# adding bias column to feature matrix X.*

        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=""):

"""

        This function takes a feature matrix as test data and

predicts the target values using the trained weights.

returns the predicted target values.

        """

        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 input*

    data = pd.read\_csv("./data.csv", header=None)

    X = data.loc[:,0:1].values

    y = data.loc[:,2].values

*# data preprocessing (Normal scaling)*

    mscaler = NormalScaler()

    mscaler.fit(X[:,0])

    X[:,0] = mscaler.transform(X[:,0])

    mscaler.fit(X[:,1])

    X[:,1] = mscaler.transform(X[:,1])

*# Training the model by choosing alpha and max\_iter values.*

*# gradient descent algorithm can be set as either ‘batch’ or ‘stochastic’*

*# in this function call.*

    arr = model.train(X,y,0.19,250,"batch")

    print("weights: ",model.W)

    print("Total Cost: ",model.cost)

*# visualization of cost function.*

    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]]).reshape(-1,1))

            r[i][j] = xx[i]

            s[i][j] = yy[j]

*# 3d surface plot of cost function and learning curve*

    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()

*# 2d contour plot of cost function*

    plt.title("2d contour plot of cost function")

    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()

*# 2d line plot of cost vs iteration*

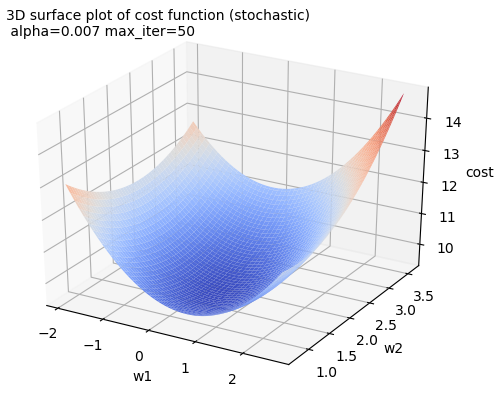
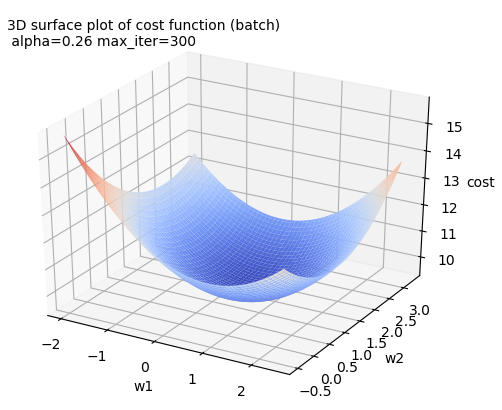
    plt.plot(model.cost\_arr)

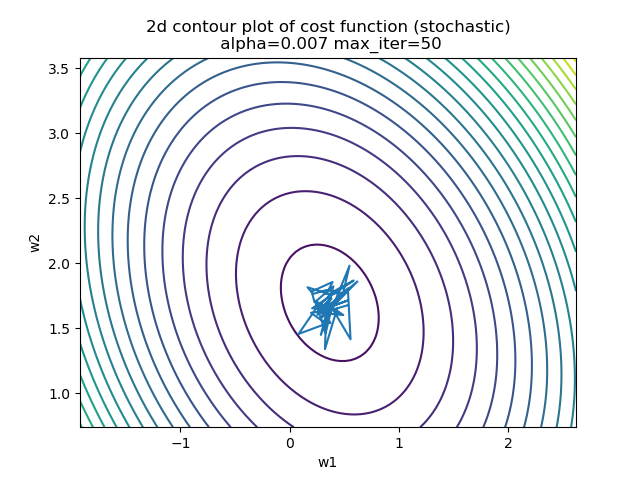
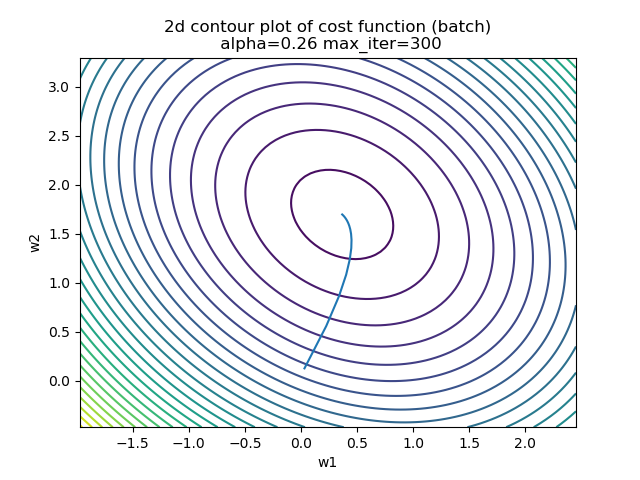
    plt.show()

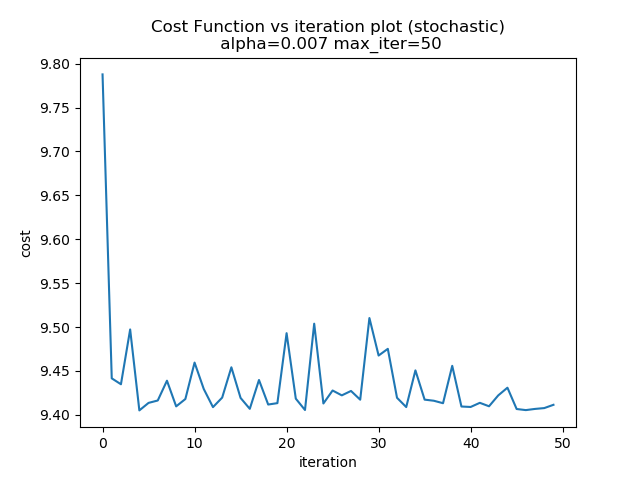
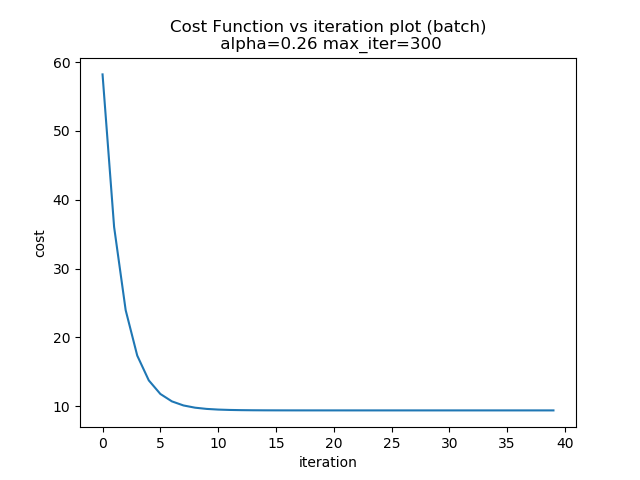
### Results:

|  |  |
| --- | --- |
|  |  |
| Batch Gradient Descent | Stochastic Gradient Descent |
| weights:  w0 = 14.9047221  w1 = 0.36752656  w2 = 1.6965344  Total Cost: 9.403235190130703 | weights:  w0 = 14.95107574  w1 = 0.39873559  w2 = 1.75867644  Total Cost: 9.407225072389698 |
|  |  |

### Plots:







## Ridge Regression

**class** RidgeRegression:

    """

    This class implements Linear Regression with regularization 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(11)

        self.W = np.random.randn(s[1],1)

        self.W\_arr = []

        self.cost\_arr = []

        self.cost = 0

        self.eta = 0

**def** get\_cost(self, X, y, W):

        """

        This function returns the cost with the given set of weights

        using the formula. Regularization term (sum of squares of weights)

        is added to the cost.

         J

    """

        reg = 0

        for i in range(1,W.shape[0]):

            reg += W[i][0]\*\*2

        total\_cost = sum(np.square(np.matmul(X,W)-y.reshape(-1,1)))[0]

        return (0.5/X.shape[0])\*total\_cost + 0.5\*self.eta\*reg

**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\_i = 0

        for j in range(X.shape[1]):

            h\_i += X[i][j]\*W[j][0]

        return h\_i

**def** batch\_grad\_descent(self, X, y, alpha, eta, 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{

            }

        """

        self.eta = eta

        for \_ in range(max\_iter):

            W\_new = np.ndarray(self.W.shape)

            for j in range(X.shape[1]):

                grad = 0

                for i in range(X.shape[0]):

                    grad += (self.get\_h\_i(X, i, self.W) - y[i])\*X[i][j]

                W\_new[j][0] = self.W[j][0]\*(1-eta\*alpha) - (alpha/X.shape[0])\*grad

            self.W = W\_new.copy()

            self.cost\_arr.append(self.get\_cost(X, y, self.W))

            self.W\_arr.append(self.W)

            if len(self.W\_arr)>1:

                if sum(abs(self.W\_arr[-2]-self.W\_arr[-1]))<0.0001:

                    break

        return W\_new

**def** stochastic\_grad\_descent(self, X, y, alpha, eta, 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{

                randomly shuffle the feature matrix rows

                for each feature vector  {

                    update all weights j -> 0 to n+1

                }

            }

        """

        mat = np.concatenate((X,y.reshape(-1,1)), axis=1)

        for \_ 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]\*(1-eta\*alpha) - (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)

            if len(self.W\_arr)>1:

                if sum(abs(self.W\_arr[-2]-self.W\_arr[-1]))<0.0001:

                    break

        return self.W

**def** train(self, X, y, alpha, eta, max\_iter=100, option="batch"):

        """

        This function initiates the training process.

        It runs batch gradient descent by default and can also run

        Stochastic gradient descent if the argument is passed.

        returns the cost list which has costs at every training iteration.

        """

*# adding bias column to feature matrix X.*

        X = self.add\_bias(X)

        self.init\_weights(X.shape)

        if option=="batch":

            self.batch\_grad\_descent(X,y,alpha,eta,max\_iter)

        elif option=="stochastic":

            self.stochastic\_grad\_descent(X,y,alpha,eta,max\_iter)

        self.cost = self.cost\_arr[len(self.cost\_arr)-1]

        return self.cost\_arr

**def** test(self,X):

        """

        This function takes a feature matrix as test data and

        predicts the target values using the trained weights.

        returns the predicted target values.

        """

        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]\*self.W[j][0]

        return y\_pred

if \_\_name\_\_ == "\_\_main\_\_":

    model = RidgeRegression()

*# data input*

    data = pd.read\_excel("./data.xlsx", header=None)

    X = data.loc[:,0:1].values

    y = data.loc[:,2].values

*# data preprocessing (MinMax scaling)*

    mscaler = NormalScaler()

    mscaler.fit(X[:,0])

    X[:,0] = mscaler.transform(X[:,0])

    mscaler.fit(X[:,1])

    X[:,1] = mscaler.transform(X[:,1])

*# Training the model by choosing alpha and max\_iter values.*

*# gradient descent algorithm can be set as either ‘batch’ or ‘stochastic’*

*# in this function call.*

    alpha = 0.1

    eta = 0.1

    max\_iter = 150

    algo = 'batch'

    arr = model.train(X,y,alpha,eta,max\_iter,algo)

    print("weights: ",model.W)

    print("Total Cost: ",model.cost)

*# visualization of cost function.*

    W\_arr = np.array(model.W\_arr)

    res = 100

    bounds = 2

    xx = np.linspace(np.min(W\_arr[:,1])-bounds, np.max(W\_arr[:,1])+bounds, res)

    yy = np.linspace(np.min(W\_arr[:,2])-bounds, np.max(W\_arr[:,2])+bounds, 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]]).reshape(-1,1))

            r[i][j] = xx[i]

            s[i][j] = yy[j]

*# 3d surface plot of cost function and learning curve*

    ax = plt.axes(projection='3d')

    ax.plot\_surface(r, s, z,cmap='coolwarm')

    ax.text2D(0.05, 0.95, "3D surface plot of cost function ({3})\n alpha={0} eta={1} max\_iter={2}".format(alpha,eta,max\_iter,algo), transform=ax.transAxes)

    ax.set\_xlabel("w1")

    ax.set\_ylabel("w2")

    ax.set\_zlabel("cost")

    plt.savefig("./Results/ridge\_reg/{3}\_{0}\_{1}\_{2}\_surf.png".format(alpha,eta,max\_iter,algo))

    plt.show()

*# 2d contour plot of cost function*

    plt.figure()

    plt.title("2d contour plot of cost function ({3})\n alpha={0} eta={1} max\_iter={2}".format(alpha,eta,max\_iter,algo))

    plt.xlabel("w1")

    plt.ylabel("w2")

    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.savefig("./Results/ridge\_reg/{3}\_{0}\_{1}\_{2}\_contour.png".format(alpha,eta,max\_iter,algo))

    plt.show()

*# 2d line plot of cost vs iteration*

    plt.figure()

    plt.plot(model.cost\_arr)

    plt.title("Cost Function vs iteration plot ({3})\n alpha={0} eta={1} max\_iter={2}".format(alpha,eta,max\_iter,algo))

    plt.xlabel("iteration")

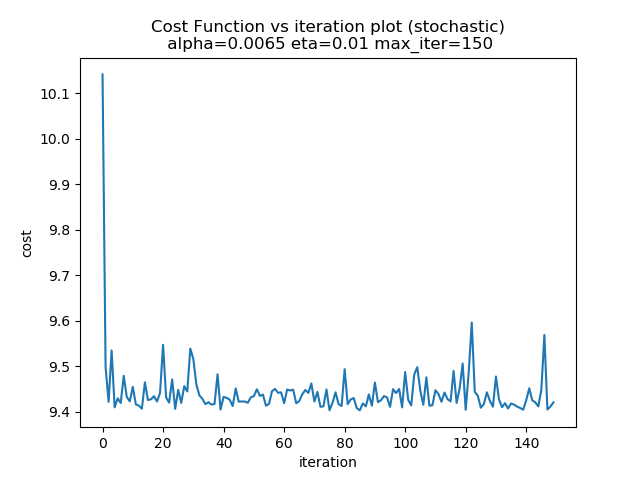
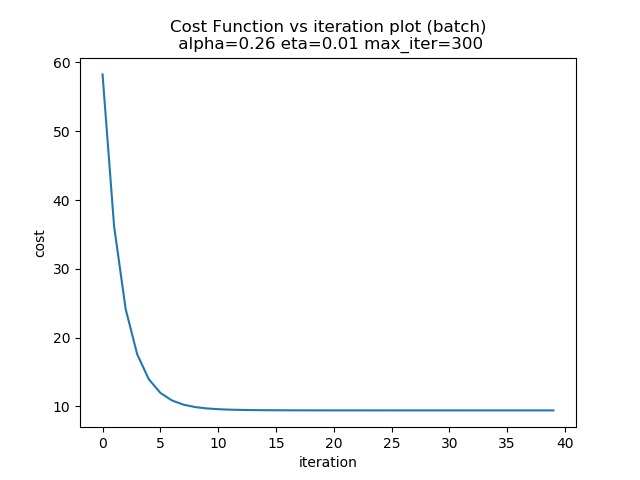
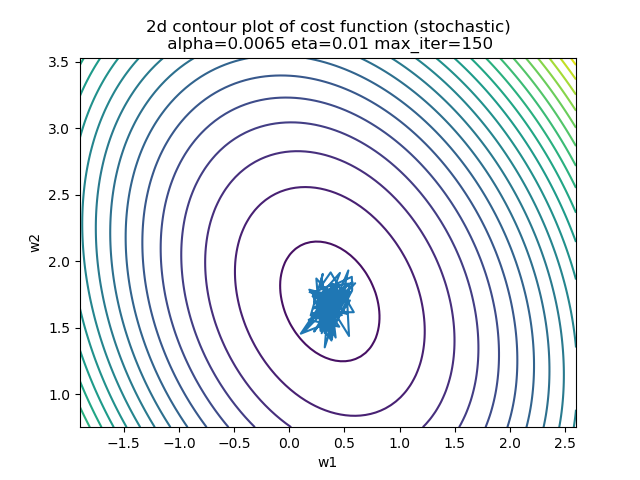
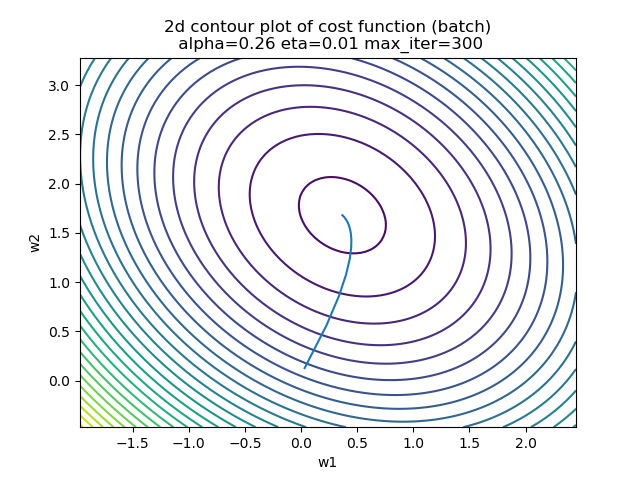
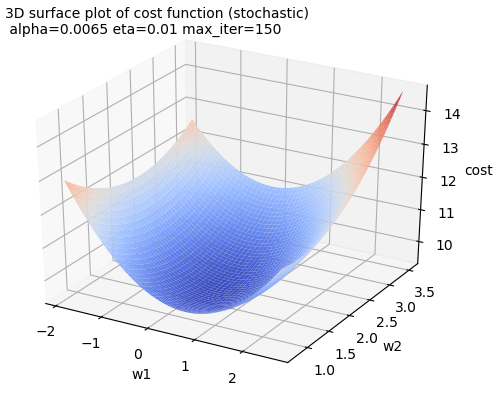
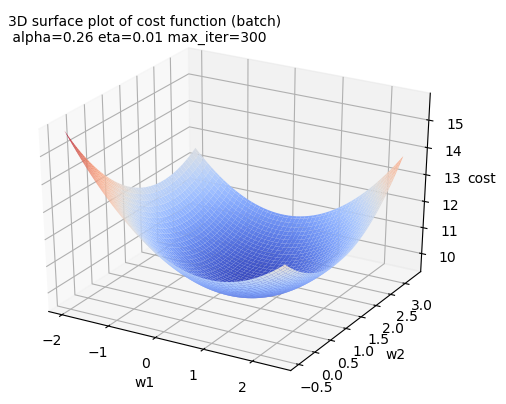
    plt.ylabel("cost")

    plt.savefig("./Results/ridge\_reg/{3}\_{0}\_{1}\_{2}\_cost\_iter.png".format(alpha,eta,max\_iter,algo))

    plt.show()

Results:

|  |  |
| --- | --- |
|  |  |
| Batch Gradient Descent | Stochastic Gradient Descent |
| weights:  w0 = 14.75716074  w1 = 0.36824758  w2 = 1.67956428  Total Cost: 9.429059509401734 | weights:  w0 = 14.8376096  w1 = 0.18609906  w2 = 1.76646287  Total Cost: 9.421065885290048 |
|  |  |
|  |  |



**class** LinearRegression\_Vectorized:

    """

    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(11)

        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

        """

        return (0.5/X.shape[0])\*np.sum(np.square(np.matmul(X,W)-y.reshape(-1,1)))

**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** train(self, X, y):

        """

        This function uses the vectorized version of linear regression

        and obtains the optimal weights with the given feature matrix

        and target values.

        returns the optimal weights.

        """

        X = self.add\_bias(X)

        self.init\_weights(X.shape)

        self.W = np.matmul(np.matmul(np.linalg.inv(np.matmul(X.T,X)), X.T), y.reshape(-1,1))

        self.cost = self.get\_cost(X,y,self.W)

        return self.W

**def** test(self,X):

        """

        This function takes a feature matrix as test data and

        predicts the target values using the trained weights.

        returns the predicted target values.

        """

        X = self.add\_bias(X)

        return np.matmul(X,self.W)

if \_\_name\_\_ == "\_\_main\_\_":

    model = LinearRegression\_Vectorized()

*# data input*

    data = pd.read\_excel("./data.xlsx", header=None)

    X = data.loc[:,0:1].values

    y = data.loc[:,2].values

*# data preprocessing (Normal Scaling)*

    print(X[:,0].mean())

    mscaler = NormalScaler()

    mscaler.fit(X[:,0])

    X[:,0] = mscaler.transform(X[:,0])

    mscaler.fit(X[:,1])

    X[:,1] = mscaler.transform(X[:,1])

*# training the model*

    arr = model.train(X,y)

    print("weights: ",model.W)

    print("Total Cost: ",model.cost)

*# visualization of cost function.*

    res = 100

    bounds = 10

    xx = np.linspace(model.W[1]-bounds, model.W[1]+bounds, res)

    yy = np.linspace(model.W[2]-bounds, model.W[2]+bounds, res)

    minw0 = model.W[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]]).reshape(-1,1))

            r[i][j] = xx[i]

            s[i][j] = yy[j]

*# 3d surface plot of cost function and learning curve*

    ax = plt.axes(projection='3d')

    ax.plot\_surface(r, s, z,cmap='coolwarm')

    ax.text2D(0.05, 0.95, "3D surface plot of cost function", transform=ax.transAxes)

    ax.set\_xlabel("w1")

    ax.set\_ylabel("w2")

    ax.set\_zlabel("cost")

    plt.savefig("./Results/lin\_reg\_vec/surf.png")

    plt.show()

Least Angle regression

**class** LeastAngleRegression:

    """

    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(11)

        self.W = np.random.randn(s[1],1)

        self.W\_arr = []

        self.cost\_arr = []

        self.cost = 0

        self.eta = 0

**def** get\_cost(self, X, y, W):

        """

        This function returns the cost with the given set of weights

        using the formula

         J

        """

        total\_cost = sum(np.square(np.matmul(X,W)-y.reshape(-1,1)))[0]

        reg = 0

        for i in range(1,W.shape[0]):

            reg += abs(W[i][0])

        return (0.5/X.shape[0])\*total\_cost + 0.5\*self.eta\*reg

**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 (x^i )=∑\_(j=0)^n▒〖w\_j 〖x^i〗\_j 〗=x^i w

        """

        h\_i = np.matmul(X[i].reshape(1,-1),W)

        return h\_i[0][0]

**def** batch\_grad\_descent(self, X, y, alpha, eta, 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{

            }

        """

        self.eta = eta

        for \_ in range(max\_iter):

            W\_new = np.ndarray(self.W.shape)

            for j in range(X.shape[1]):

                grad = 0

                for i in range(X.shape[0]):

                    grad += (self.get\_h\_i(X, i, self.W) - y[i])\*X[i][j]

                W\_new[j][0] = self.W[j][0]-0.5\*eta\*alpha\*np.sign(self.W[j][0]) - (alpha/X.shape[0])\*grad

            self.W = W\_new.copy()

            self.cost\_arr.append(self.get\_cost(X, y, self.W))

            self.W\_arr.append(self.W)

            if len(self.W\_arr)>1:

                if sum(abs(self.W\_arr[-2]-self.W\_arr[-1]))<0.0001:

                    break

        return W\_new

**def** stochastic\_grad\_descent(self, X, y, alpha,eta, 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{

                randomly shuffle the feature matrix rows

                for each feature vector x^i {

                    update all weights j -> 0 to n+1

                }

            }

        """

        mat = np.concatenate((X,y.reshape(-1,1)), axis=1)

        for \_ 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]-0.5\*eta\*alpha\*np.sign(self.W[j][0]) - (alpha/X.shape[0])\*temp\*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)

            if len(self.W\_arr)>1:

                if sum(abs(self.W\_arr[-2]-self.W\_arr[-1]))<0.0001:

                    break

        return self.W

**def** train(self, X, y, alpha, eta, max\_iter=100, option="batch"):

        """

        This function initiates the training process.

        It runs batch gradient descent by default and can also run

        Stochastic gradient descent if the argument is passed.

        returns the cost list which has costs at every training iteration.

        """

*# adding bias column to feature matrix X.*

        X = self.add\_bias(X)

        self.init\_weights(X.shape)

        if option=="batch":

            self.batch\_grad\_descent(X,y,alpha,eta,max\_iter)

        elif option=="stochastic":

            self.stochastic\_grad\_descent(X,y,alpha,eta,max\_iter)

        self.cost = self.cost\_arr[len(self.cost\_arr)-1]

        return self.cost\_arr

**def** test(self,X):

        """

        This function takes a feature matrix as test data and

        predicts the target values using the trained weights.

        returns the predicted target values.

        """

        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]\*self.W[j][0]

        return y\_pred

if \_\_name\_\_ == "\_\_main\_\_":

    model = LeastAngleRegression()

*# data input*

    data = pd.read\_csv("./data.csv", header=None)

    X = data.loc[:,0:1].values

    y = data.loc[:,2].values

*# data preprocessing (MinMax scaling)*

    mscaler = MinMaxScaler()

    mscaler.fit(X[:,0])

    X[:,0] = mscaler.transform(X[:,0])

    mscaler.fit(X[:,1])

    X[:,1] = mscaler.transform(X[:,1])

*# Training the model by choosing alpha and max\_iter values.*

*# gradient descent algorithm can be set as either ‘batch’ or ‘stochastic’*

*# in this function call.*

    alpha = 0.1

    eta = 0.01

    max\_iter = 150

    algo = 'batch'

    arr = model.train(X,y,alpha,eta,max\_iter,algo)

    print("weights: ",model.W)

    print("Total Cost: ",model.cost)