Linear Regression(Multiple Variable) - From Scratch

Cost Function: Mean Squared Error(MSE)
Algorithm: Gradient Descent

In [1]:
```
# Importing dependencies and dataset
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from mpl_toolkits.mplot3d import Axes3D
```

In [2]:
```
boston = load_boston()
df = pd.DataFrame(data = boston['data'], columns = boston['feature_names'])
df['PRICE'] = boston['target']
df.head()
```

Out[2]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33

In [3]:
```
#Data statistics
df.describe()
```

Out[3]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126

In [4]:
```
corr = df.corr()
corr['PRICE'].sort_values(ascending = False)

#Data below shows that RM is most positively correlated to price and LSTAT
```

Out[4]:
```
PRICE      1.000000
RM         0.695360
ZN         0.360445
B          0.333461
DIS        0.249929
CHAS       0.175260
AGE       -0.376955
RAD       -0.381626
CRIM      -0.388305
NOX       -0.427321
TAX       -0.468536
INDUS     -0.483725
PTRATIO   -0.507787
LSTAT     -0.737663
Name: PRICE, dtype: float64
```

Feature Scaling

In [5]:
```
#Need to scale feature.  Standard Scaling(use with normally distributed dat
X = df.drop(['PRICE'], axis = 1)
X = preprocessing.scale(X)
X = np.c_[np.ones(df.shape[0]),X]
X
```

Out[5]:
```
array([[ 1.        , -0.41978194,  0.28482986, ..., -1.45900038,
         0.44105193, -1.0755623 ],
       [ 1.        , -0.41733926, -0.48772236, ..., -0.30309415,
         0.44105193, -0.49243937],
       [ 1.        , -0.41734159, -0.48772236, ..., -0.30309415,
         0.39642699, -1.2087274 ],
       ...,
       [ 1.        , -0.41344658, -0.48772236, ...,  1.17646583,
         0.44105193, -0.98304761],
       [ 1.        , -0.40776407, -0.48772236, ...,  1.17646583,
         0.4032249 , -0.86530163],
       [ 1.        , -0.41500016, -0.48772236, ...,  1.17646583,
         0.44105193, -0.66905833]])
```

In [6]:
```
y = df['PRICE']
y = preprocessing.scale(df['PRICE'])
```

In [7]:
```
#Splitting Data Set into Training and Testing Data Sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = .80,
```

CREATING COST FUNCTION AND GRADIENT DESCENT ALGORITHM

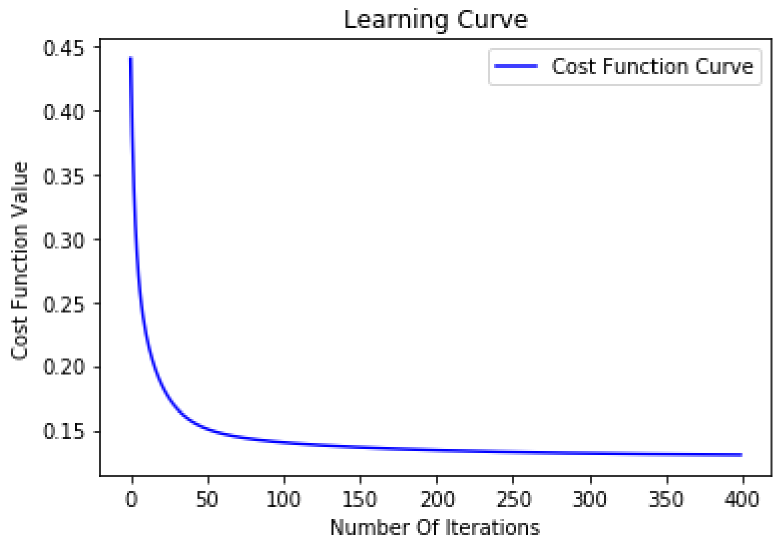
In [8]:
```
#Cost Function(Mean Squared Error)/Calculates MSE for at given point
def cost_function(X_train, y_train, thetas_array, n):
    cost = np.sum((np.dot(X_train,np.transpose(thetas_array)) - y_train)**2
    return cost
```

In [9]:
```
#Setting Up Gradient Descent Algorithm
def gradient_descent(X_train, y_train, alpha, thetas_array, n):
    thetas = thetas_array - (alpha * ((np.sum((np.dot(X_train,np.transpose(
    return thetas
```

TRAINING THE DATA SET

In [10]:
```
def training(X_train, y_train, alpha, iters):
    n = len(X_train)

    #initializing optimization parameter
    thetas_array = np.zeros(14)
    #new_thetas = []
    mse_values = []

    for i in range(iters):
        thetas_array = gradient_descent(X_train, y_train, alpha, thetas_arr
        #new_thetas.append(thetas_array)
        mse_values.append(cost_function(X_train, y_train, thetas_array, n))


    #Plot cost function error per iteration
    x = np.arange(0, len(mse_values), step=1)
    plt.plot(x, mse_values, "-b", label="Cost Function Curve")
    plt.title("Learning Curve")
    plt.xlabel("Number Of Iterations")
    plt.ylabel("Cost Function Value")
    plt.legend()
    plt.show()

    return thetas_array
```

In [11]:
```
training(np.array(X_train), np.array(y_train), .03, 400)
```



Out[11]:
```
array([ 0.0372569 , -0.09031931,  0.10662716,  0.04688386,  0.04670029,
       -0.27124515,  0.2855625 ,  0.08612669, -0.28095601,  0.16667364,
       -0.10112596, -0.27535978,  0.05474949, -0.43123812])
```

In [12]:
```
def testing(X_test, y_test, thetas_array):
    n = len(X_test)

    SSTO = []   # total sum of squares
    SSR = []    # regression sum of squares
    SSE = []    # error sum of squares
    y_mean = np.mean(y_train)

    #prediction = []

    for i in range(n):
        predict = np.dot(X_test[i],thetas_array)
        #prediction.append(predict)
        SSE.append((predict - y_test[i])**2)
        SSR.append((predict - y_mean)**2)
        SSTO.append((y_test[i] - y_mean)**2)

    print('\naverage error is : ', round(sum(SSE)/len(SSE),4))
    print('\nsum of squares of error (SSE) : ', round(sum(SSE),4))
    print('\nregression sum of squares (SSR) : ', round(sum(SSR),4))
    print('\ntotal sum of squares (SSTO) : ', round(sum(SSTO), 4))
    print('\nThe Coefficient Of Determination R-squared is : ', (round(sum(
```

In [13]:
```
testing(np.array(X_test), np.array(y_test), [ 0.03793393, -0.08794142,  0.0
       -0.25369856,  0.29775376,  0.07859893, -0.26907797,  0.15042058,
       -0.08865473, -0.2695052 ,  0.05265216, -0.42111509])
```

```
average error is :  0.2719

sum of squares of error (SSE) :  110.1104

regression sum of squares (SSR) :  293.4558

total sum of squares (SSTO) :  401.7879

The Coefficient Of Determination R-squared is :  73.0375 %
```

In [ ]: