Information Engineering and Technology Faculty
German University in Cairo

NETW-1013: Machine Learning

Assignment Report

Submitted by: Fatma Abdelmageed 40-2005

Supervisor: Dr. Ing - Maggie Mashally

Submission Date: 25th April , 2021

Outline

Introduction

Machine learning (ML) is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. ML focuses on the development of computer programs that can access data and use it to learn for themselves. There are several types of ML, which are: Supervised, Unsupervised, and Reinforcement Learning. First, supervised learning. Which is also known as predictive learning. Its goal is to learn how to map from inputs x to outputs y. It consists of 2 types, which are: **Regression** “when y is a continuous value output” and **Classification** “when y is a discrete value output”. Second, unsupervised learning, which is also known as descriptive learning. Its goal is to find interesting patterns in the data. It consists of 2 types, which are **Clustering** “ grouping data into cohesive groups “ and **Non-Clustering** “finding structure in a chaotic environment”.

In this report, it focuses on linear regression to estimate the hypothesis function to predict continuous valued output while using ML diagnostics. In order to avoid the overfitting and underfitting problems, Model Selection technique is used.

Methodology

Model Selection

1. Goal:

Learn the parameter theta from training data to minimize training error $J(\theta)$ & compute test set error using computecostmulti

2. Steps:

1) Read the dataset :
`data=pd.read_csv("house_prices_data_training_data.csv")`

2) Drop the Nan cells in the dataset: `data.dropna(axis=0, how='any', thresh= None, subset= None, inplace=True)`

3) Get correlation between ll features and the price, and drop the features that have correlation < 0.5 :

```
correlation_table= data.corr()
cor_target = abs(correlation_table["price"])
relevant_features = cor_target[cor_target>0.5]
#drop features that have correlation <0.5
data =
data.drop(["id","date","bedrooms","sqft_lot","floors","waterfront","view","condition","sqft_basement","yr_built","yr_renovated","zipcode","lat","long","sqft_lot15"] , axis = 1)
```

4) Convert data frame to an array & divide the array into X and Y: X represents the features, Y represents the price

```
array = data.to_numpy()
X = array[:,1:]
Y = array[:,0]
```

5) Split the data into a Training Set (60%), a Cross Validation (CV) Set (20%) and a Test Set (20%)

```
# Dividing into 60% Trainning set and 40% the rest.
trainX, restX, trainY, restY = train_test_split(X, Y, test_size = 0.4, shuffle =
False)
# Dividing the remaining 40% into 20% Testing set and 20% validation set.
validateX , testX, validateY, testY = train_test_split(restX, restY, test_size =
0.5, shuffle = False)
```

6) Normalize the trainX, testX, validateX using the featureNormalize(X) function implemented in assignment 1

```
trainX_norm , mutrain, sigmatrain =
featureNormalize(trainX)
validateX_norm , muvalidate, sigmavalidate = featureNormalize(validateX)
testX_norm , mutest, sigmatest = featureNormalize(testX)
```

7) Add ones to normalized trainX, testX, validateX

```
trainY_size = trainY.size
trainX_norm = np.concatenate([np.ones((trainY_size,1)), trainX_norm],
axis=1)
testY_size= testY.size
testX_norm = np.concatenate([np.ones((testY_size,1)), testX_norm], axis=1)
validateY_size = validateY.size
validateX_norm = np.concatenate([np.ones((validateY_size,1)),
validateX_norm], axis=1)
```

8) Will need to use computeCostMulti(X, y, theta) function : (Previously implemented in assignment 1)
Compute cost for linear regression with multiple variables.Computes the cost of using theta as the parameter for linear regression to fit the data points in X and y.

9) Will need to use gradientDescentMulti(X, y, theta, alpha, num_iters) function: (Previously implemented in assignment 1)
Performs gradient descent to learn theta.
Updates theta by taking num_iters gradient steps with learning rate alpha.

10) Use different alphas = 0.001,0.01,0.0002

11) Number of iterations = 150
12) Compute gradient descent on the normalized trainX, testX, validateX

```
theta, J_history1 = gradientDescentMulti(trainX_norm, trainY, theta, alpha1, num_iters)
print('Computed theta:',theta)
    theta2, J_history2 = gradientDescentMulti(validateX_norm, validateY, theta, alpha2, num_iters)
print('Computed theta:',theta2)
theta3, J_history3= gradientDescentMulti(testX_norm, testY, theta, alpha3, num_iters)
print('Computed theta:',theta3)
```

13) Plot JHistory vs Number of iterations

```
pyplot.figure()
pyplot.plot(np.arange(len(J_history1)), J_history1, lw=2)
pyplot.xlabel('Number of iterations')
pyplot.ylabel('Cost J 1')

#alpha 2
pyplot.figure()
pyplot.plot(np.arange(len(J_history2)), J_history2, lw=2)
pyplot.xlabel('Number of iterations')
pyplot.ylabel('Cost J 2')

#alpha 3
pyplot.figure()
pyplot.plot(np.arange(len(J_history3)), J_history3, lw=2)
pyplot.xlabel('Number of iterations')
pyplot.ylabel('Cost J 3')
```

14) Computecostmulti on normalized testX, TestY, and thetas computed from the gradient descent to calculate the error

Train w validation

Estimate the generalization error using the test set

```
Jtest1 = computeCostMulti(testX_norm, testY, theta)
Jtest2 = computeCostMulti(testX_norm, testY, theta2)
Jtest3 = computeCostMulti(testX_norm, testY, theta3)
```

15) Make 4 new functions, which are:
    a) computeCostMulti2
    b) computeCostMulti3
    c) gradientDescentMulti2
    d) gradientDescentMulti3

NB: THE YELLOW HIGHLIGHTED LINES ARE UPDATED LINES IN THE FUNCTION

```
def computeCostMulti2(trainX, trainY, theta):

    # Initialize some useful values
    m = trainY.shape[0] # number of training examples
    # You need to return the following variable correctly
    J = 0
    # ====================== YOUR CODE HERE
==========================
    J=0
    J = np.dot((np.dot(np.square(trainX), theta) - trainY), (np.dot(np.square(trainX),
theta) - trainY)) / (2 * m) + ((1/(2*m))* np.sum(np.dot(theta, theta)))
    #
===========================================================
=====
    return J

def computeCostMulti3(trainX, trainY, theta):

    # Initialize some useful values
    m = trainY.shape[0] # number of training examples
    # You need to return the following variable correctly
    J = 0
    # ====================== YOUR CODE HERE
==========================
 J=0
 J = np.dot((np.dot(np.power(trainX, 3), theta) - trainY), (np.dot(np.power(trainX,
3), theta) - trainY)) / (2 * m) + ((1/(2*m))* np.sum(np.dot(theta, theta)))
 #
===========================================================
=====
    return J

def gradientDescentMulti2(trainX, trainY, theta, alpha, num_iters):

    # Initialize some useful values
    m = trainY.shape[0] # number of training examples
    # make a copy of theta, which will be updated by gradient descent
    theta = theta.copy()
    J_history = []
    for i in range(num_iters):
        # ====================== YOUR CODE HERE
=========================
        hypothesis=np.dot(np.square(trainX), theta)
```

```
        theta=theta*(1-(alpha)/m)-((alpha/m)*(np.dot(trainX.T, hypothesis-trainY)))
        #
=================================================================
====
        # save the cost J in every iteration
        J_history.append(computeCostMulti2(trainX, trainY, theta))
    return theta, J_history


def gradientDescentMulti3(trainX, trainY, theta, alpha, num_iters):

    # Initialize some useful values
    m = trainY.shape[0] # number of training examples
    # make a copy of theta, which will be updated by gradient descent
    theta = theta.copy()
    J_history = []
    for i in range(num_iters):
        # ====================== YOUR CODE HERE
===========================
        hypothesis=np.dot(np.power(trainX, 3), theta)
        theta=theta*(1-(alpha)/m)-((alpha/m)*(np.dot(trainX.T, hypothesis-trainY)))
        #
=================================================================
====
        # save the cost J in every iteration
        J_history.append(computeCostMulti3(trainX, trainY, theta))
    return theta, J_history
```

Regularization

In regularization, the computeCostMulti and gradientDescentMulti functions differ from that of the model selection by adding a “Penalty Term” (lamda) that increases with the complexity of the hypothesis to the optimization problem. “Regularization factor is added”

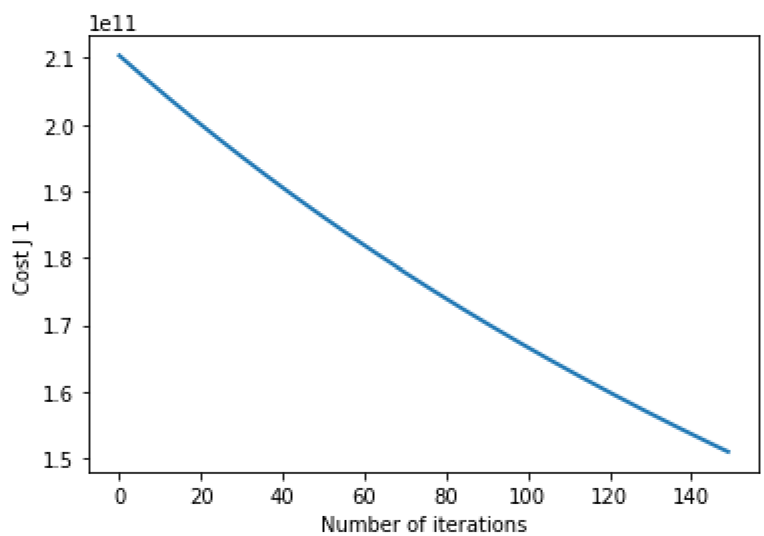
```
def computeCostMulti(X, y, theta, lambda):
    m = y.shape[0]
    J = 0
    thetaa = theta.copy()
    thetaa[0] = 0
    J= ((1/(2*m)) * np.dot(np.transpose(np.dot(X,theta)-y),np.dot(X,theta)-y)) +
((lambda_ / (2 * m)) * np.sum(np.square(thetaa)))
    return J
```

```
def gradientDescentMulti(trainX, trainY, theta, alpha, num_iters,lam):
    # Initialize some useful values
    m = trainY.shape[0] # number of training examples
    # make a copy of theta, which will be updated by gradient descent
    theta = theta.copy()
    J_history = []
    for i in range(num_iters):
        # ======================= YOUR CODE HERE ==========================
        hypothesis= np.dot(trainX,theta)
        theta = theta - ((alpha/m) * ( (np.dot(hypothesis-trainY,trainX))  +  (lam * theta) ))
        # ================================================================
        # save the cost J in every iteration
        J_history.append(computeCostMulti(trainX, trainY, theta, lam))
    return theta, J_history
```

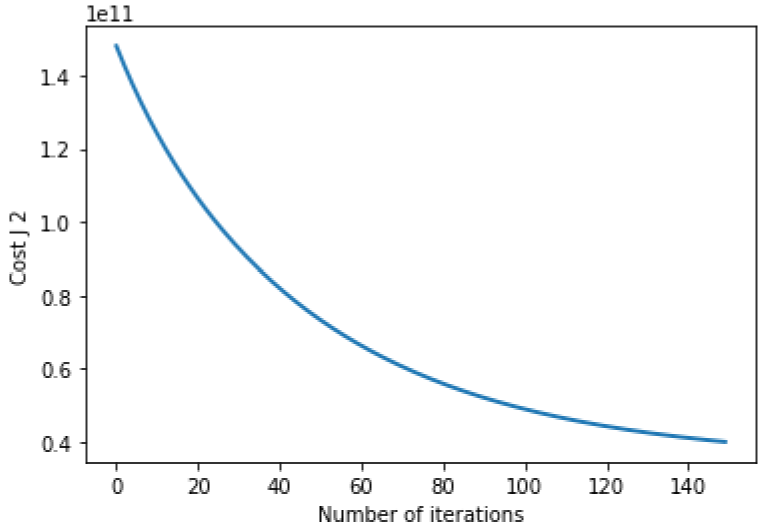
Results

A. Model Selection

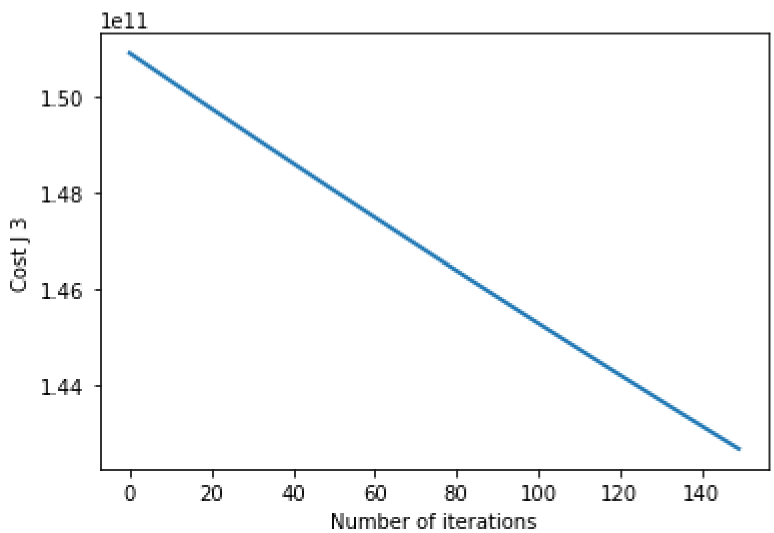
I. Number of iterations = 150
II. Using gradientDescentMulti(trainX_norm, trainY, theta, alpha, num_iters)
Used different alphas:
    A. alpha1= 0.001



B. alpha2= 0.01



C. alpha3= 0.0002



From the previous graphs, it's obvious that using alpha2 is appropriate as CostJ converges soon. So, theta2 will be used to evaluate the computeCostMulti for testX, trainX and validateX.

D. Cost error:

```
Jtest2 = computeCostMulti(testX_norm, testY, theta2)
Jtest2
```

```
34272149810.970287
```

```
Jtrain2 = computeCostMulti(trainX_norm, trainY, theta2)
Jtrain2
```

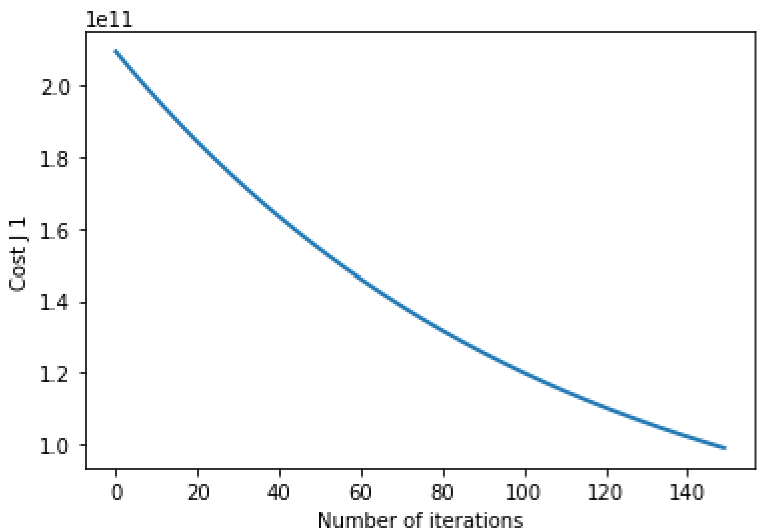
```
39928859090.027016
```

```
Jvalidate2 = computeCostMulti(validateX_norm, validateY, theta2)
Jvalidate2
```

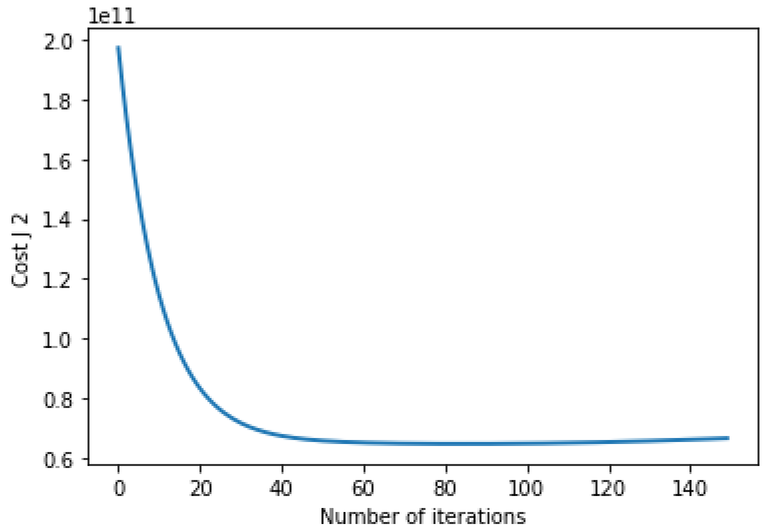
```
35229769855.84928
```

From the previous results, Jtrain>JValidate.

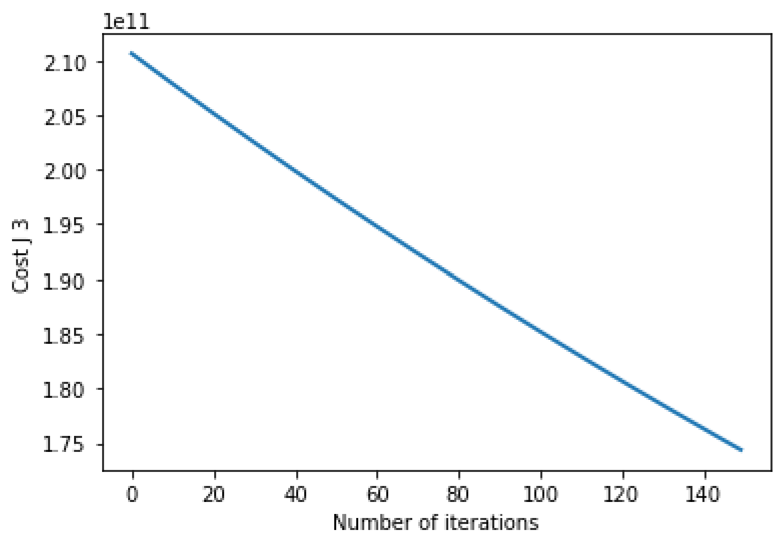
III. Using gradientDescentMulti2(trainX_norm, trainY, theta, alpha, num_iters)
Used different alphas:
   A. alpha1= 0.001



B. alpha2= 0.01



C. alpha3= 0.0002



From the previous graphs, it's obvious that using alpha2 is appropriate as CostJ converges soon. So, theta2 will be used to evaluate the computeCostMulti2 for testX, trainX and validateX.

D. Cost error:

```
: Jtest22 = computeCostMulti2(testX_norm, testY, theta_22)
Jtest22
```

```
: 65891270508.58069
```

```
: Jtrain22 = computeCostMulti2(trainX_norm, trainY, theta_22)
Jtrain22
```

```
: 66472517615.179245
```

```
alidate22 = computeCostMulti2(validateX_norm, validateY, theta_22
alidate22
```

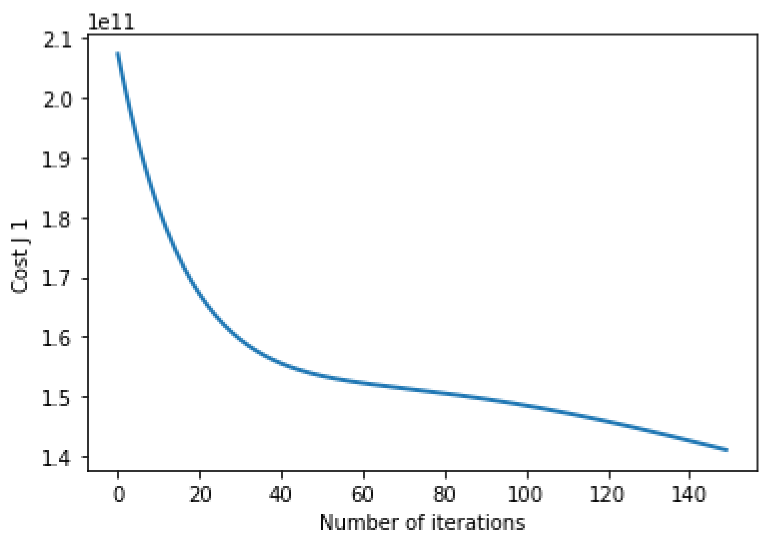
```
89698284417.26802
```

From the previous results, Jtrain << JValidate. Therefore, its suffering variance problem.

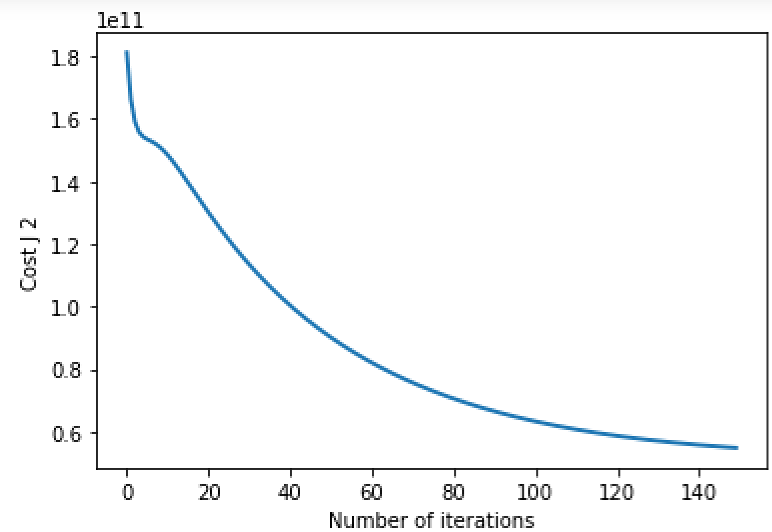
IV. Using gradientDescentMult3(trainX_norm, trainY, theta, alpha1, num_iters)

Used different alphas:

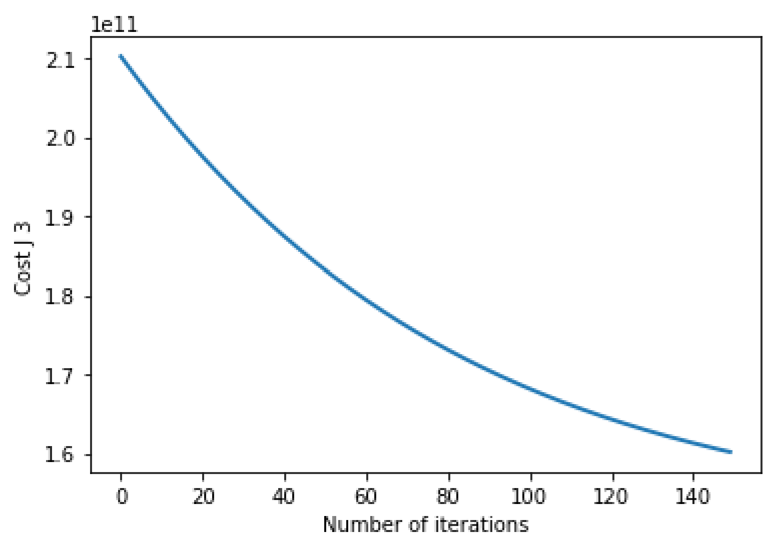
A. alpha1= 0.001



B. alpha2= 0.01



C. alpha3= 0.0002



From the previous graphs, it's obvious that using alpha2 is appropriate as CostJ converges soon. So, theta1 will be used to evaluate the computeCostMulti2 for testX, trainX and validateX.

D. Cost error:

```
Jtest31 = computeCostMulti3(testX_norm, testY, theta_31)
Jtest3
```

```
136638786146.45032
```

```
Jtrain31 = computeCostMulti3(trainX_norm, trainY, theta_31)
Jtrain31
```

```
141017885334.7105
```

```
: lidate31 = computeCostMulti3(validateX_norm, validateY, theta_31)
lidate31
```

```
: 256850585729.49384
```

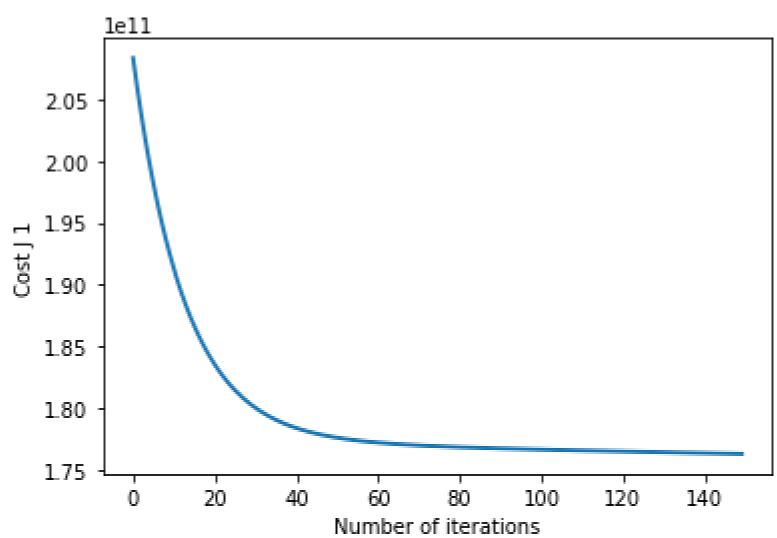
From the previous results, Jtrain << JValidate. Therefore, its suffering variance problem.

Regularization

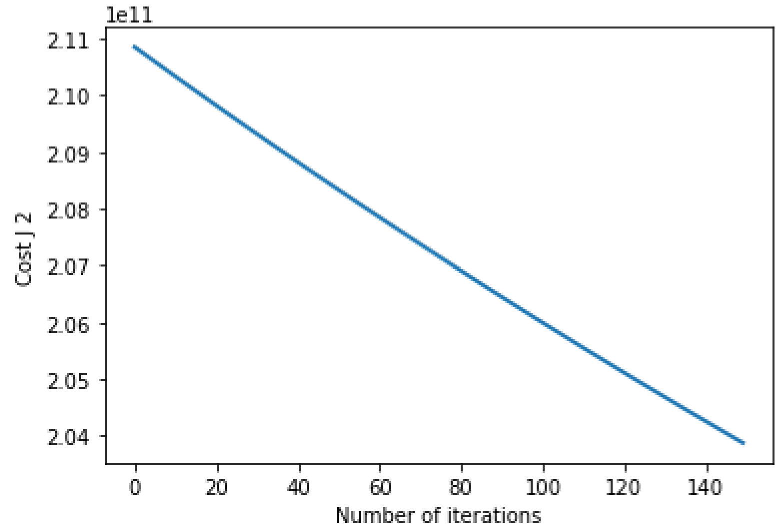
- Number of iterations = 150, lamda=0.6
- Using gradientDescentMulti(trainX_norm, trainY, theta, alpha, num_iters)

  Used different alphas:

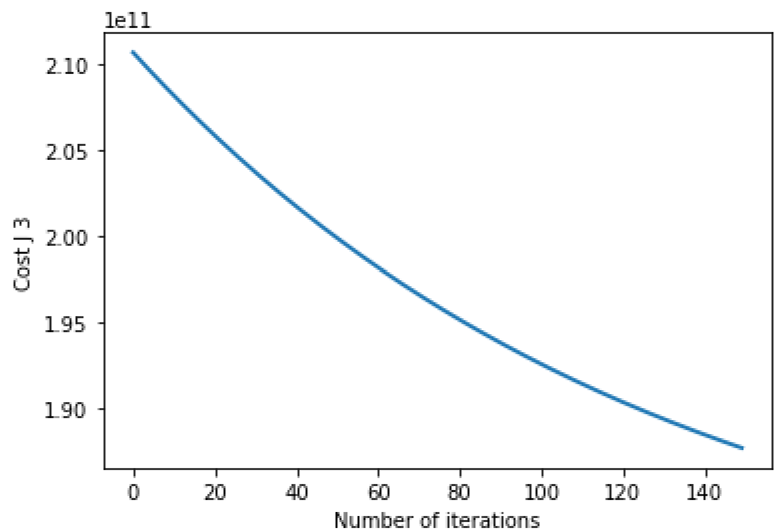
  1. alpha=0.01



2. alpha=0.0002



3. alpha=0.001



From the previous graphs, it's obvious that using alpha1 is appropriate as CostJ converges soon. So, theta1 will be used to evaluate the computeCostMulti for testX, trainX and validateX.

- Cost Error:

```
Jtest3 = computeCostMulti(testX_norm, testY, theta1,lam)
Jtest3
```

```
171934786794.8109
```

```
Jtrain3 = computeCostMulti(trainX_norm, trainY, theta1,lam)
Jtrain3
```

```
176310791977.15994
```

```
validate3 = computeCostMulti(validateX_norm, validateY, theta1,lam)
validate3
```

```
172966046800.98535
```

From the previous results, JValidate~ Jtrain ~ Jtests

Conclusion

In conclusion, while using the model selection technique, squaring and powering the H by 3 increased the cost error, so using the first computeCostMulti and gradientDescentMulti functions are better than computeCostMulti2, computeCostMulti3, gradientDescentMulti2 and gradientDescentMulti3. In addition, while using regularization technique, using alpha=0.01 is appropriate as CostJ converges soon.