Machine leaning Assigment 1

Name: Amgad Tarek

ID: 40-7261

Methodology

In this assignment after reading lecture 5 and 6 then we started coding using Jupyter notebook with the language being python in order to do the machine learning. The assignment was divided into 4 parts. Firstly we started by reading the data set file which is a CSV file. Then we divided our data into 3 parts 60% are for training, 20% for testing and 20% for validation. In the third part we defined two important methods which are gradientDescentMulti and computeCostMulti for multiple variables and features. Then lastly invoking those methods on each of the training, testing and validating data.

1. Reading CSV file

```
In [3]: data=pd.read_csv("house_prices_data_training_data.csv")
        columns =data.columns[3:]
        norm_data = (data[columns]-(data[columns]).mean())/(data[columns]).std()
        data[columns] = norm_data
```

In here we used the pd module which reads the csv file and saves it in a variable called data then we save the columns of the csv file in in an instant called columns by invoking the method columns on the variable data then we normalized the data so the values could be between -1 and 1.

2. Dividing Data

According to model selection in lecture 5 we have to divide our dataset since it's large into 60% used for training data to optimize the parameters in theta for each polynomial degree in J_train(theta). Then there is 20% used for cross validation to find the polynomial degree with the least error using the cross validation set using the J_cv(theta) equation in lecture 5. The last 20% are conserved for the testing data in order to estimate the generalization error using also the J_test(theta) equation in the lecture.

```
In [4]: train, validate , test = np.split(data.sample(frac=1),[int(.6*len(data)),int(.8*len(data))])
```

```
In [5]: train = train.to_numpy()
        test = test.to_numpy()
        validate = validate.to_numpy()
        train_x = train[:,3:]
        validate_x = validate[:,3:]
        test_x = test[:,3:]
        train_y = train[:,2]
        validate_y = validate[:,2]
        test_y =test[:,2]
        train_m =train_y.size
```

In this part of code we divided our data into train, validate and test randomly then we for each one we had X and Y. The X's are for the feature columns which are all columns except the first 3 which are price, id and date, the Y's are for the price feature which is the third column in our csv file. Then we saved all this in arrays using the numpy method so we can use them and use them inside our defined methods.

3. GradientDescentMulti and ComputeCostMulti

Firstly the gradientdescent, we used it to choose coffiecent of theta that will minimize our cost and the cost is the difference between input and output with the input being the X's which are the features and the output being the Y's which defines the price. Then we use the compute multicost to calculate this cost .

In [7]:
```
def gradientDescentMulti(X, y, theta, alpha, num_iters):
    m = y.shape[0]
    theta = theta.copy()
    J_history = []
    for i in range(num_iters):
        theta = theta - (alpha / m) * (np.dot(X, theta) - y).dot(X)
        J_history.append(computeCostMulti(X, y, theta))
    return theta, J_history
```

In [8]:
```
def computeCostMulti(X, y, theta):
    m = y.shape[0]
    J = 0
    h = np.dot(X, theta)
    J = (1/(2 * m)) * np.sum(np.square(np.dot(X, theta) - y))
    return J
```

This is the code used to define both methods. Also we call the compute method in the gradienrdescent because w need to know the cost and save it in array called J_history. The Gradient descent method takes x,y,theta,alpha and number of iterations and returns a new theta and the J_history array. The computemulti takes x,y,theta and returns J which is variable containing the cost.

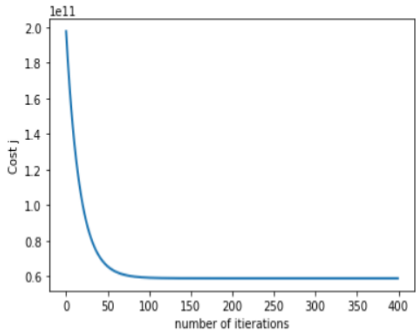
4. Invoking the methods

4.1 Train data set

We started by invoking the gradient descendt method on the training data set to get the theta and minimum cost function. We have 18 features that we need to invoke this method on every time the method takes the same alpha, number of iterations, Y, theta but different X value as the X value describes the feature.

```
In [9]: alpha = 0.03
        num_iters = 400
        theta = np.zeros(2)
        j_history=[]
        u, j_history = gradientDescentMulti(train_x[:,0:2], train_y, theta, alpha, num_iters)
        pyplot.plot(np.arange(len(j_history)), j_history, lw=2)
        pyplot.xlabel('number of itierations')
        pyplot.ylabel('Cost j')

Out[9]: Text(0, 0.5, 'Cost j')
```



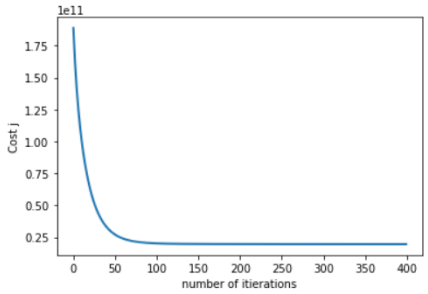

```
In [10]: j_history[-1]

Out[10]: 58983699166.75465
```

In this code we set the value of alpha by 0.03 and number of iterations by 400 then we firstly set the theta by zero and we invoke the gradient method on the first x column which is the first feature which is also the third column in our csv file and the method also takes the Y, theta, alpha, number of iterations. Then we plot we cost which is returned from this function against the number of iterations where cost is y_axis and number of iterations as x-axis. Then we invoke this method 18 times because we have 18 features in our dataset and in the code we just change the train_x parameter making it increment the value of the feature which represents the column we take and compute the gradient for. In everytime we invoke the method the graphs are nearly similar all that changed is the J_history array which has the cost values so we print the last value in this array which describes the minimum cost everytime we invoke the gradient method and in the last invoke this value will be at its absolute minimum.

```
In [35]: alpha = 0.03
         num_iters = 400
         theta = np.zeros(19)
         j_history=[]
         u, j_history = gradientDescentMulti(train_x[:,0:19], train_y, theta, alpha, num_iters)
         pyplot.plot(np.arange(len(j_history)), j_history, lw=2)
         pyplot.xlabel('number of itierations')
         pyplot.ylabel('Cost j')

Out[35]: Text(0, 0.5, 'Cost j')
```




```
In [36]: j_history[-1]

Out[36]: 19621705191.57939
```

This is the last iteration of the gradient method on the last feature in the training dataset and its observed that the last value in the J_history is at its minimum.

4.2 Validation data set

We defined the validation dataset before and we will use it just like we use the train dataset the only difference is we will just invoke the compute multicost on it no need to invoke the gradient because we use the validation to find the least error in the polynomial and to compare it with the cost we got in return from invoking gradient method on the training dataset.

```
In [38]: alpha = 0.03
         num_iters = 400
         theta = np.zeros(3)
         j_history=[]
         u, j_history = computeCostMulti(validate_x[:,0:3], validate_y, theta)
```

4.3 Test data set

After we are made sure that the training and validation datasets are working and running we will invoke the compute cost once on the test data set with the X being the best feature which got us the minimal cost as we learned from the training set and this feature is the last one, the Y is the price but the one defined for the test data set and the theta is the theta we got from the minimal and best degree.

In [1]:
```
alpha = 0.03
num_iters = 400
theta = np.zeros(19)
j_history=[]
u, j_history =computeCostMulti(test_x[:,0:19], test_y, theta)
```