Mohamed Hany Elkhalifa 40-0331 T8

ASSIGNMENT 1 REPORT

Import the useful libraries to be used in the code. Read the csv file and visualize the first five rows of the dataset using the head command and get the dimensions of the dataset using the shape command in order to know how many rows and columns.

In [1]:
```
import os
import pandas as pd
import numpy as np
import seaborn as sns
from matplotlib import pyplot
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline
from sklearn.preprocessing import StandardScaler
from collections import Counter
import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
data = pd.read_csv(r"D:\Desktop\Assignments of Machine Learning\house_prices_data_training_data.csv", delimiter=',')
#Return the 1st five rows
data.head()
```

Out[2]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	sqft_above	sqft_basement	yr_bu
0	7.129301e+09	20141013T000000	221900.0	3.0	1.00	1180.0	5650.0	1.0	0.0	0.0	...	7.0	1180.0	0.0	195
1	6.414100e+09	20141209T000000	538000.0	3.0	2.25	2570.0	7242.0	2.0	0.0	0.0	...	7.0	2170.0	400.0	195
2	5.631500e+09	20150225T000000	180000.0	2.0	1.00	770.0	10000.0	1.0	0.0	0.0	...	6.0	770.0	0.0	193
3	2.487201e+09	20141209T000000	604000.0	4.0	3.00	1960.0	5000.0	1.0	0.0	0.0	...	7.0	1050.0	910.0	196
4	1.954401e+09	20150218T000000	510000.0	3.0	2.00	1680.0	8080.0	1.0	0.0	0.0	...	8.0	1680.0	0.0	198

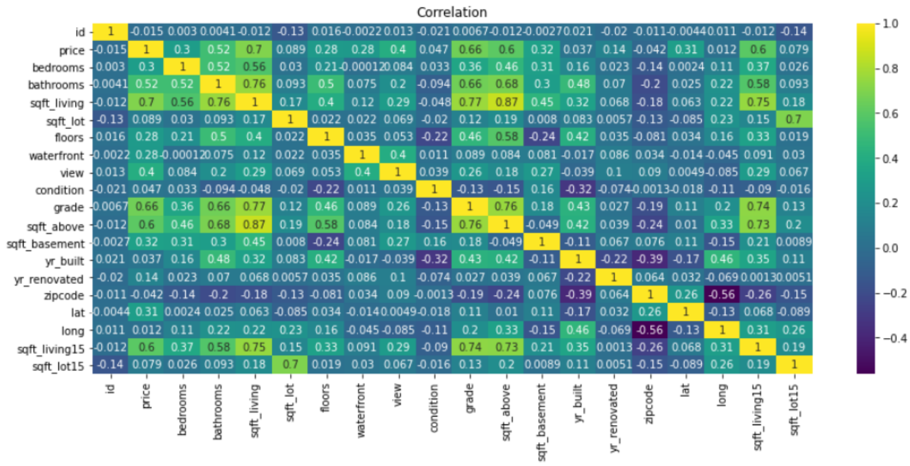
5 rows × 21 columns

In [3]:
```
data.shape
```

Out[3]:
```
(21607, 21)
```

Remember and use from the data engineering course, the heat map. Heat map helps to find correlation between the target variable (price) and the features affecting the target variable (other variables). The positive correlation ranges from]0,1[, the higher the value gets the more positively correlated the variables are. (Zero and one are not included).

```
In [4]: plt.figure(figsize=(15,6))
        plt.title('Correlation')
        sns.heatmap(data.corr(),annot=True,cmap='viridis',linecolor='white')
        plt.show()
```



Drop the empty rows which contain no values.

After Putting a threshold level of 0.3 in the correlation produced by the heatmap, drop the columns of variables that their correlation values are less than 0.3. Adjust the dataset to be the data after removing the unnecessary variables in order to decrease the complexity.

data=data.drop(['id','date','sqft_lot','floors','waterfront','condition','yr_built','yr_renovated','zipcode','long','sqft_lot15'], axis=1)

Let X be the features in the adjusted dataset that affect the target variable. Let Y be the target variable (Price) column and get the dimensions of X and the dimension of y. then, using the head command review the first five rows of the adjusted data set.

```
In [7]: X= data.iloc[:, 1:].values
        y= data.iloc[:, 0].values

In [8]: X.shape
Out[8]: (17999, 9)

In [9]: y.shape
Out[9]: (17999,)

In [10]: data.head()
Out[10]:
```

	price	bedrooms	bathrooms	sqft_living	view	grade	sqft_above	sqft_basement	lat	sqft_living15
0	221900.0	3.0	1.00	1180.0	0.0	7.0	1180.0	0.0	47.5112	1340.0
1	538000.0	3.0	2.25	2570.0	0.0	7.0	2170.0	400.0	47.7210	1690.0
2	180000.0	2.0	1.00	770.0	0.0	6.0	770.0	0.0	47.7379	2720.0
3	604000.0	4.0	3.00	1960.0	0.0	7.0	1050.0	910.0	47.5208	1360.0
4	510000.0	3.0	2.00	1680.0	0.0	8.0	1680.0	0.0	47.6168	1800.0

Plot each variable of X against Price(Y) using pyplot. (9 figures) shwn in the code.

In the model selection, split the data into a Training Set(60%), a Cross Validation (CV) Set (20%) and a Test Set (20%)

Check their dimensions (datatrain datavalidate datatest)

```
In [20]: def train_validate_test_split(df, train_percent=0.6, validate_percent=0.2, seed=None):
    np.random.seed(seed)
    perm = np.random.permutation(df.index)
    m = len(df.index)
    train_end = int(train_percent * m)
    validate_end = int(validate_percent * m) + train_end
    train = df.iloc[perm[:train_end]]
    validate = df.iloc[perm[train_end:validate_end]]
    test = df.iloc[perm[validate_end:]]
    return train,validate,test
datatrain, datavalidate, datatest = train_validate_test_split(data, train_percent=0.6, validate_percent=0.2, seed=None)
print(datavalidate.shape)
print(datatrain.shape)
print(datatest.shape)

(3599, 10)
(10799, 10)
(3601, 10)
```

FEATURE NORMALIZATION

In feature normalization, normalize the features in X. return a normalized version of X where the mean value of each feature is 0 and the standard deviation is 1.

For each feature dimension, compute the mean of the feature and subtract it from the dataset, storing the mean value in mu. Next, compute the standard deviation of each feature and divide each feature by its standard deviation, storing the standard deviation in sigma.

```
In [21]: def  featureNormalize(X):
    X_norm = X.copy()
    mu = np.zeros(X.shape[1])
    sigma = np.zeros(X.shape[1])

    mu = np.mean(X, axis = 0)
    sigma = np.std(X, axis = 0)
    X_norm = (X - mu) / sigma

    return X_norm, mu, sigma
```

Perform the call of the featurenormalize function sepearetly on the datatrain, datavalidate and datatest and print the computed mean and standard deviation for each of the three splitted data. After calling the feature normalize function, add the intercept term (concatenate) to the datatrain_norm, datavalidate_norm, datatest_norm.

In [22]:
```
#featureNormalize(datatrain) showing all details

# call featureNormalize on the loaded data
datatrain_norm, datatrain_mu, datatrain_sigma = featureNormalize(datatrain)

print('Train Computed mean:', datatrain_mu)
print('Train Computed standard deviation:', datatrain_sigma)
```

```
Train Computed mean: price            533607.771923
bedrooms              3.368090
bathrooms             2.064427
sqft_living        2055.418465
view                  0.246319
grade                 7.591444
sqft_above         1752.597926
sqft_basement       302.820539
lat                  47.558484
sqft_living15      1970.957218
dtype: float64
Train Computed standard deviation: price            368601.750966
bedrooms              0.941961
bathrooms             0.756542
sqft_living         905.020999
view                  0.782529
grade                 1.168364
sqft_above          805.142997
sqft_basement       449.859964
lat                   0.140621
sqft_living15       672.418993
dtype: float64
```

```
In [23]: #featureNormalize(datavalidate)

         # call featureNormalize on the loaded data
         datavalidate_norm, datavalidate_mu, datavalidate_sigma = featureNormalize(datavalidate)

         print('Validate Computed mean:', datavalidate_mu)
         print('Validate Computed standard deviation:', datavalidate_sigma)

         Validate Computed mean: price          531880.418450
         bedrooms              3.349264
         bathrooms             2.053904
         sqft_living        2037.035010
         view                  0.244512
         grade                 7.587941
         sqft_above         1738.729091
         sqft_basement       298.305918
         lat                  47.561992
         sqft_living15      1970.780217
         dtype: float64
         Validate Computed standard deviation: price          366190.324779
         bedrooms              0.918758
         bathrooms             0.756153
         sqft_living         891.152571
         view                  0.792930
         grade                 1.173468
         sqft_above          804.775899
         sqft_basement       447.360445
         lat                   0.137749
         sqft_living15       675.561045
         dtype: float64
```

```
In [24]: #featureNormalize(datatest)

         # call featureNormalize on the loaded data
         datatest_norm, datatest_mu, datatest_sigma = featureNormalize(datatest)
         print('Test Computed mean:', datatest_mu)
         print('Test Computed standard deviation:', datatest_sigma)

         Test Computed mean: price          532667.447098
         bedrooms              3.361289
         bathrooms             2.060816
         sqft_living        2055.816995
         view                  0.228825
         grade                 7.599556
         sqft_above         1752.185782
         sqft_basement       303.631214
         lat                  47.563611
         sqft_living15      1982.527909
         dtype: float64
         Test Computed standard deviation: price          349640.935140
         bedrooms              0.924979
         bathrooms             0.767159
         sqft_living         907.130936
         view                  0.752935
         grade                 1.168712
         sqft_above          806.723813
         sqft_basement       450.185245
         lat                   0.138476
         sqft_living15       672.937605
         dtype: float64
```

```
In [25]: # Add intercept term to X
         datatrain = pd.DataFrame(np.concatenate([np.ones((len(datatrain.index), 1)), datatrain_norm], axis=1))
         datavalidate = pd.DataFrame(np.concatenate([np.ones((len(datavalidate.index), 1)), datavalidate_norm], axis=1))
         datatest = pd.DataFrame(np.concatenate([np.ones((len(datatest.index), 1)), datatest_norm], axis=1))
```

COMPUTE COST

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. Set J to the cost, to compute the cost of a particular choice of theta where the thetas represent linear regression parameters.

```
In [32]: 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
```

GRADIENT DESCENT

the gradientdescentmulti function takes 5 parameters and returns two parameters,

the five parameters are

X ------> adjusted dataset features array

y-------> target variable (price column)

Theta --> linear regression parameters

alpha---> learning rate for gradient descent

num_iters --->number of iterations to run gradient descent

the returned parameters are

theta--> The learned linear regression parameters

J_history --> A python list for the values of the cost function after each iteration.

```
In [33]: def gradientDescentMulti(X, y, theta, alpha, num_iters):
             m = y.shape[0]
             theta = theta.copy()
             J_history = []
             for i in range(num_iters):
                 #hypothesis=0
                 #theta=0
                 theta = theta - (alpha / m) * (np.dot(X, theta) - y).dot(X)
                 J_history.append(computeCostMulti(X, y, theta))
             return theta, J_history
```

In training data, initialize thetas to have zeros in the columns of each feature. For example the first feature has zero in all its column where its column has an index equal to 2 and so on.

call the gradientdescent using the datatrain to get the trained thetas and jtrain. Where jtrain is a list of j_history for the one feature iterated 400 times and taking into consideration alpha=0.1. where the last minimum theta between all thetas is taken as the minimum theta thus its degree represents it. (Degree 8)

```
In [34]: alpha =0.1
         num_iters=400
         #Training data
         Temptrain=datatrain
         price=datatrain.iloc[:,1]
         Temptrain=Temptrain.drop(columns=1)
         theta1 = np.zeros(2)
         theta2 = np.zeros(3)
         theta3 = np.zeros(4)
         theta4 = np.zeros(5)
         theta5 = np.zeros(6)
         theta6 = np.zeros(7)
         theta7 = np.zeros(8)
         theta8 = np.zeros(9)
         #print(Temptrain.iloc[:,0:2])
         #print(price)
         theta1, jtrain1 =gradientDescentMulti(Temptrain.iloc[:,0:2],price, theta1, alpha, num_iters)
         theta2, jtrain2 =gradientDescentMulti(Temptrain.iloc[:,0:3],price, theta2, alpha, num_iters)
         theta3, jtrain3 =gradientDescentMulti(Temptrain.iloc[:,0:4],price, theta3, alpha, num_iters)
         theta4, jtrain4 =gradientDescentMulti(Temptrain.iloc[:,0:5],price, theta4, alpha, num_iters)
         theta5, jtrain5 =gradientDescentMulti(Temptrain.iloc[:,0:6],price, theta5, alpha, num_iters)
         theta6, jtrain6 =gradientDescentMulti(Temptrain.iloc[:,0:7],price, theta6, alpha, num_iters)
         theta7, jtrain7 =gradientDescentMulti(Temptrain.iloc[:,0:8],price, theta7, alpha, num_iters)
         theta8, jtrain8 =gradientDescentMulti(Temptrain.iloc[:,0:9],price, theta8, alpha, num_iters)

In [ ]: jtrain1

In [35]: jtrain1[399:400],jtrain2[399:400],jtrain3[399:400],jtrain4[399:400],jtrain5[399:400],jtrain6[399:400],jtrain7[399:400],jtrain8[3

Out[35]: ([0.4562535567027253],
          [0.3607480442601414],
          [0.2514131741514999],
          [0.23109183593202062],
          [0.2156958083107458],
          [0.21808987558090578],
          [0.21270748038015672],
          [0.1818200807665678])
```


In validating data, what we care about is the compute cost by each feature. Call computecostmulti function passing to it the X (feature) , y(target variable i.e: price), and the thetas that came from training the date (not the initial zero thetas). In return, this function gives me the J cost of validating data/error where the least j is the best as deducing the error. (Its degree is equal to 8) same degree of training data. Thus I have a degree 8.

```
In [37]: #validating data
         Tempvalidate=datavalidate
         price=datavalidate.iloc[:,1]
         Tempvalidate=Tempvalidate.drop(columns=1)
         jval1 = np.zeros(2)
         jval2 = np.zeros(3)
         jval3 = np.zeros(4)
         jval4 = np.zeros(5)
         jval5 = np.zeros(6)
         jval6 = np.zeros(7)
         jval7 = np.zeros(8)
         jval8 = np.zeros(9)
         #print(Tempvalidate.iloc[:,0:2])
         #print(price)
         print(theta8)
         jval1 =computeCostMulti(Tempvalidate.iloc[:,0:2],price, theta1)
         jval2 =computeCostMulti(Tempvalidate.iloc[:,0:3],price, theta2)
         jval3 =computeCostMulti(Tempvalidate.iloc[:,0:4],price, theta3)
         jval4 =computeCostMulti(Tempvalidate.iloc[:,0:5],price, theta4)
         jval5 =computeCostMulti(Tempvalidate.iloc[:,0:6],price, theta5)
         jval6 =computeCostMulti(Tempvalidate.iloc[:,0:7],price, theta6)
         jval7 =computeCostMulti(Tempvalidate.iloc[:,0:8],price, theta7)
         jval8 =computeCostMulti(Tempvalidate.iloc[:,0:9],price, theta8)
```

After training and validating data, now Test data by calculating jtest by calling computecostmulti function now only once by using theta8 representing degree 8 which show the least error. Calculate price which is equal to the dot product of jtest and theta 8.

```
In [38]: print(jval1, jval2, jval3, jval4,jval5,jval6,jval7,jval8)

0.44984461782413154 0.3685970050001038 0.2617134112477748 0.24195469825819826 0.22448553356745796 0.226266737618277 0.222479319
4169082 0.19098984354087986

In [39]: #Testing data
Temptest=datatest
price=datatest.iloc[:,1]
Temptest=Temptest.drop(columns=1)
jtest = np.zeros(9)
#print(Temptest.iloc[:,0:2])
#print(price)
jtest =computeCostMulti(Temptest.iloc[:,0:9],price, theta8)

In [40]: print(jtest)

0.17861567803770628

In [41]: price = np.dot(jtest, theta8)

In [42]: print(price)

[-1.17265964e-14 -1.07808827e-02  3.12759643e-03  4.23511455e-02
  3.62695244e-02  4.10939344e-02  3.53839874e-02  2.18723762e-02
  4.46810054e-02]
```

Acti