Mohamed Ahmed Gouda 40-936

Report

After importing all the needed libraries in the code, the CSV file is read to check how many rows and columns it has and two of its columns are deleted which are the date and id columns as they deem useless!

```
In [27]: data = pd.read_csv('C:\\Users\\Mohamed Gouda\\Desktop\\house_prices_data_training_data.csv')

In [28]: data.dropna(inplace=True)
         data.shape

Out[28]: (17999, 21)

In [29]: del data['id']
         del data['date']
         data.head(10)
```

Out[29]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade	sqft_above	sqft_basement	yr_built	yr_renovated	zipco
0	221900.0	3.0	1.00	1180.0	5650.0	1.0	0.0	0.0	3.0	7.0	1180.0	0.0	1955.0	0.0	9817
1	538000.0	3.0	2.25	2570.0	7242.0	2.0	0.0	0.0	3.0	7.0	2170.0	400.0	1951.0	1991.0	9812
2	180000.0	2.0	1.00	770.0	10000.0	1.0	0.0	0.0	3.0	6.0	770.0	0.0	1933.0	0.0	9802
3	604000.0	4.0	3.00	1960.0	5000.0	1.0	0.0	0.0	5.0	7.0	1050.0	910.0	1965.0	0.0	9813
4	510000.0	3.0	2.00	1680.0	8080.0	1.0	0.0	0.0	3.0	8.0	1680.0	0.0	1987.0	0.0	9807
5	1230000.0	4.0	4.50	5420.0	101930.0	1.0	0.0	0.0	3.0	11.0	3890.0	1530.0	2001.0	0.0	9805
6	257500.0	3.0	2.25	1715.0	6819.0	2.0	0.0	0.0	3.0	7.0	1715.0	0.0	1995.0	0.0	9800
7	291850.0	3.0	1.50	1060.0	9711.0	1.0	0.0	0.0	3.0	7.0	1060.0	0.0	1963.0	0.0	9819
8	229500.0	3.0	1.00	1780.0	7470.0	1.0	0.0	0.0	3.0	7.0	1050.0	730.0	1960.0	0.0	9814
9	323000.0	3.0	2.50	1890.0	6560.0	2.0	0.0	0.0	3.0	7.0	1890.0	0.0	2003.0	0.0	9803

Data threshold is then put to 0.3 in the correlation produced by the heatmap so any columns with variables of correlation less than 0.3 is dropped which are columns 4,5,6,8,12,13,14,16 and 18. The correlation is between price and the other features affecting it!

```
In [30]: dfcorrnew=data.corr()
         dfcorrnew.drop(dfcorrnew.index[[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18]] , inplace=True)
         dfcorrnew.head()
         dnew=dfcorrnew.iloc[0].to_numpy()
         #print(dnew)
         result = np.where(dnew < .3)
         print(result)

         (array([ 4,  5,  6,  8, 12, 13, 14, 16, 18], dtype=int64),)

In [31]: data.drop(data.columns[result], axis = 1, inplace = True)
         data.head()
```

Out[31]:

	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

Now Y will take the role of price and X will take the role of features that affect the price. Data is then split into training dataset which is equal to 0.6, validation dataset which is equal to 0.2 and testing dataset which is equal to 0.2 as well. Then each of them is divided into an input set X and Output set Y. The datasets are to be normalized then trained. 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 and for each feature 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 [33]:
```
def  featureNormalize(X):
    X_norm = X.copy()
    mu = np.zeros(X.shape[1])
    sigma = np.zeros(X.shape[1])
    # ========================== YOUR CODE HERE =====================
    mu = np.mean(X, axis = 0)
    sigma = np.std(X, axis = 0)
    X_norm = (X - mu) / sigma
    # ================================================================
    return X_norm, mu, sigma
```

In [37]:
```
data_norm, mu, sigma = featureNormalize(data)
print('Computed mean:', mu)
print('Computed standard deviation:', sigma)
#print(data_norm)
data_norm = np.concatenate([np.ones((m, 1)), data_norm], axis=1)
data_norm=pd.DataFrame(data_norm)
data_norm.head()
```

In [38]:
```
def train_validate_test_split(df, train_percent=.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
train , validate , test =train_validate_test_split(data_norm, train_percent=.6, validate_percent=0.2, seed=None)
print(validate.shape)
print(train.shape)
print(test.shape)
```

```
(3599, 11)
(10799, 11)
(3601, 11)
```

In the compute cost segment the compute of cost is done for linear regression with multiple variables, theta is computed to as a parameter to fit the data points in Y and X and j is set to compute the cost of certain choice of theta where they represent parameters of linear regression!

In [16]:
```
def computeCostMulti(X, y, theta):
    m = y.shape[0]
    J = 0
    # ====================== YOUR CODE HERE ==========================
    h = np.dot(X, theta)
    J = (1/(2 * m)) * np.sum(np.square(np.dot(X, theta) - y))
    # ================================================================
    return J
```

As for the gradient descent, this function takes five inputs and produces two. It takes in X,Y and theta which were previously explained along with number of iterations to run gradient descent and alpha which is the learning rate of the gradient descent. Then this function returns theta which is a learned linear regression parameter and J history which is a list of all cost functions after each iteration.

In [17]:
```
def gradientDescentMulti(X, y, theta, alpha, num_iters):
    m = y.shape[0]
    theta = theta.copy()
    J_history = []
    for i in range(num_iters):
        # ====================== YOUR CODE HERE ==========================
        theta = theta - (alpha / m) * (np.dot(X, theta) - y).dot(X)
        # ================================================================
        J_history.append(computeCostMulti(X, y, theta))

    return theta, J_history
```

When training the data, thetas are to have zeros in their columns as a starting point, then the gradient descent function is called to get the J trained along with the trained thetas. The J trained resembles a list of J history for a single feature that is iterated 400 times where alpha is at 0.1! At the end the least theta is chosen and is then represented.

Theta 1

In [189]:
```
alpha = 0.01
num_iters = 400
theta_one = np.zeros(1)
theta, J_history = gradientDescentMulti(X_train.iloc[:,0:1], y_train, theta_one, alpha, num_iters)
pyplot.plot(np.arange(len(J_history)), J_history, lw=2)
pyplot.xlabel('Number of iterations')
pyplot.ylabel('Cost J')
thetaone_trained=theta
print('theta computed from gradient descent: {:s}'.format(str(theta)))
# ====================== YOUR CODE HERE ===========================
normalize_test_data = None
normalize_test_data = None
price = 0
# ================================================================
print(J_history)
print('J =',J_history[-1])
print('thetaone_trained',thetaone_trained)
print('Predicted price of a 1650 sq-ft, 3 br house (using gradient descent): ${:.0f}'.format(price))
```

```
theta computed from gradient descent: 2    0.296056
```

Theta 2

In [190]:
```
alpha = 0.01
num_iters = 400
theta_two = np.zeros(2)
theta, J_history = gradientDescentMulti(X_train.iloc[:,0:2], y_train, theta_two, alpha, num_iters)
pyplot.plot(np.arange(len(J_history)), J_history, lw=2)
pyplot.xlabel('Number of iterations')
pyplot.ylabel('Cost J')
thetatwo_trained=theta
print('theta computed from gradient descent: {:s}'.format(str(theta)))
# ====================== YOUR CODE HERE ===========================
normalize_test_data = None
normalize_test_data = None
price = 0
# ================================================================
print(J_history)
print('J =',J_history[-1])
print(thetatwo_trained)
print('Predicted price of a 1650 sq-ft, 3 br house (using gradient descent): ${:.0f}'.format(price))
```

```
theta computed from gradient descent: 2    0.093942
```

Theta 3

In [191]:
```
alpha = 0.01
num_iters = 400
theta_three = np.zeros(3)
theta, J_history = gradientDescentMulti(X_train.iloc[:,0:3], y_train, theta_three, alpha, num_iters)
pyplot.plot(np.arange(len(J_history)), J_history, lw=2)
pyplot.xlabel('Number of iterations')
pyplot.ylabel('Cost J')
thetathree_trained=theta
print('theta computed from gradient descent: {:s}'.format(str(theta)))
# ====================== YOUR CODE HERE ===========================
normalize_test_data = None
normalize_test_data = None
price = 0
# ================================================================
print(J_history)
print('J =',J_history[-1])
print('Predicted price of a 1650 sq-ft, 3 br house (using gradient descent): ${:.0f}'.format(price))
```

Theta 4

In [192]:
```
alpha = 0.01
num_iters = 400
theta_four = np.zeros(4)
theta, J_history = gradientDescentMulti(X_train.iloc[:,0:4], y_train, theta_four, alpha, num_iters)
pyplot.plot(np.arange(len(J_history)), J_history, lw=2)
pyplot.xlabel('Number of iterations')
pyplot.ylabel('Cost J')
thetatfour_trained=theta
print('theta computed from gradient descent: {:s}'.format(str(theta)))
# ====================== YOUR CODE HERE ==========================
normalize_test_data = None
normalize_test_data = None
price = 0
# ================================================================
print(J_history)
print('J =',J_history[-1])
print('Predicted price of a 1650 sq-ft, 3 br house (using gradient descent): ${:.0f}'.format(price))
```

```
theta computed from gradient descent: 2   -0.046601
```

Theta 5

In [193]:
```
alpha = 0.01
num_iters = 400
theta_five = np.zeros(5)
theta, J_history = gradientDescentMulti(X_train.iloc[:,0:5], y_train, theta_five, alpha, num_iters)
pyplot.plot(np.arange(len(J_history)), J_history, lw=2)
pyplot.xlabel('Number of iterations')
pyplot.ylabel('Cost J')
thetatfive_trained=theta
print('theta computed from gradient descent: {:s}'.format(str(theta)))
# ====================== YOUR CODE HERE ==========================
normalize_test_data = None
normalize_test_data = None
price = 0
# ================================================================
print(J_history)
print('J =',J_history[-1])
print('Predicted price of a 1650 sq-ft, 3 br house (using gradient descent): ${:.0f}'.format(price))
```

```
theta computed from gradient descent: 2   -0.029895
```

Theta 6

In [194]:
```
alpha = 0.01
num_iters = 400
theta_six = np.zeros(6)
theta, J_history = gradientDescentMulti(X_train.iloc[:,0:6], y_train, theta_six, alpha, num_iters)
pyplot.plot(np.arange(len(J_history)), J_history, lw=2)
pyplot.xlabel('Number of iterations')
pyplot.ylabel('Cost J')
thetatsix_trained=theta
print('theta computed from gradient descent: {:s}'.format(str(theta)))
# ====================== YOUR CODE HERE ==========================
normalize_test_data = None
normalize_test_data = None
price = 0
# ================================================================
print(J_history)
print('J =',J_history[-1])
print('Predicted price of a 1650 sq-ft, 3 br house (using gradient descent): ${:.0f}'.format(price))
```

```
theta computed from gradient descent: 2   -0.030005
```

Theta 7

In [195]:
```
alpha = 0.01
num_iters = 400
theta_seven = np.zeros(7)
theta, J_history = gradientDescentMulti(X_train.iloc[:,0:7], y_train, theta_seven, alpha, num_iters)
pyplot.plot(np.arange(len(J_history)), J_history, lw=2)
pyplot.xlabel('Number of iterations')
pyplot.ylabel('Cost J')
thetatseven_trained=theta
print('theta computed from gradient descent: {:s}'.format(str(theta)))
# ====================== YOUR CODE HERE ==========================
normalize_test_data = None
normalize_test_data = None
price = 0
# ================================================================
print(J_history)
print('J =',J_history[-1])
print('Predicted price of a 1650 sq-ft, 3 br house (using gradient descent): ${:.0f}'.format(price))
```

```
theta computed from gradient descent: 2   -0.054645
```

Theta 8

In [196]:
```
alpha = 0.01
num_iters = 400
theta_eight = np.zeros(8)
theta, J_history = gradientDescentMulti(X_train.iloc[:,0:8], y_train, theta_eight, alpha, num_iters)
pyplot.plot(np.arange(len(J_history)), J_history, lw=2)
pyplot.xlabel('Number of iterations')
pyplot.ylabel('Cost J')
thetateight_trained=theta
print('theta computed from gradient descent: {:s}'.format(str(theta)))
# ====================== YOUR CODE HERE ==========================
normalize_test_data = None
normalize_test_data = None
price = 0
# ================================================================
print(J_history)
print('J =',J_history[-1])
print('Predicted price of a 1650 sq-ft, 3 br house (using gradient descent): ${:.0f}'.format(price))
```

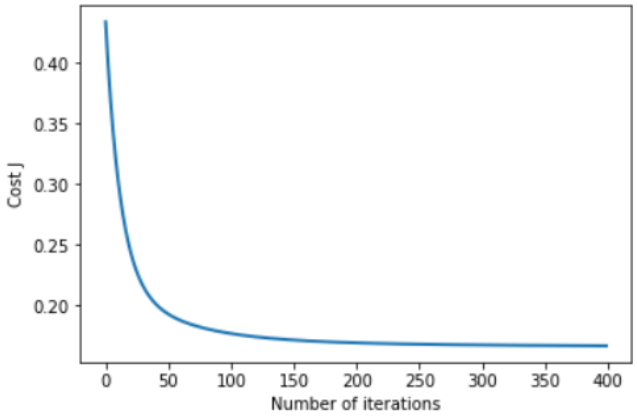
```
theta computed from gradient descent: 2   -0.047353
```

Theta 9

In [198]:
```
alpha = 0.01
num_iters = 400
theta_nine = np.zeros(9)
theta, J_history = gradientDescentMulti(X_train.iloc[:,0:9], y_train, theta_nine, alpha, num_iters)
pyplot.plot(np.arange(len(J_history)), J_history, lw=2)
pyplot.xlabel('Number of iterations')
pyplot.ylabel('Cost J')
thetatnine_trained=theta
print('theta computed from gradient descent: {:s}'.format(str(theta)))
# ====================== YOUR CODE HERE ==========================
normalize_test_data = None
normalize_test_data = None
price = 0
# ================================================================
print(J_history)
print('J =',J_history[-1])
print('Predicted price of a 1650 sq-ft, 3 br house (using gradient descent): ${:.0f}'.format(price))
```

```
theta computed from gradient descent: 2    -0.044472
```

Gradient Descent



To validate the data each feature is called in the compute cost function where the X,Y and trained thetas from the previous figures are imported and a J cost of validating data is returned and since they all had the same training, the least J is the best in terms of error deduction and J8 falls under that hypothesis! After training and validation, J general is now calculated by using theta 8 into the compute cost function.

In [204]:
```
j1=computeCostMulti(X_val.iloc[:,0:1], y_val, thetaone_trained)
j2=computeCostMulti(X_val.iloc[:,0:2], y_val, thetatwo_trained)
j3=computeCostMulti(X_val.iloc[:,0:3], y_val, thetathree_trained)
j4=computeCostMulti(X_val.iloc[:,0:4], y_val, thetatfour_trained)
j5=computeCostMulti(X_val.iloc[:,0:5], y_val, thetatfive_trained)
j6=computeCostMulti(X_val.iloc[:,0:6], y_val, thetatsix_trained)
j7=computeCostMulti(X_val.iloc[:,0:7], y_val, thetatseven_trained)
j8=computeCostMulti(X_val.iloc[:,0:8], y_val, thetateight_trained)
j9=computeCostMulti(X_val.iloc[:,0:9], y_val, thetatnine_trained)
print ('j1=',j1)
print ('j2=',j2)
print ('j3=',j3)
print ('j4=',j4)
print ('j5=',j5)
print ('j6=',j6)
print ('j7=',j7)
print ('j8=',j8)
print ('j9=',j9)
```

```
j1= 0.5075982761975284
j2= 0.39679672737513844
j3= 0.2814188086043088
j4= 0.26211231621121117
j5= 0.24415756894579493
j6= 0.24579711213365488
j7= 0.24124551227982943
j8= 0.20888392808636239
j9= 0.210294906204049
```

In [205]:
```
j_general=computeCostMulti(X_test.iloc[:,0:8], y_test, thetateight_trained)
print ('j_general',j_general)
```

```
j_general 0.2091026437820602
```