Yahia Ashraf Khalil 40-3878 T 10

Machine Learning Report 1

Insights about the data set

The data set has a single output (y) and 18 features. I had to decide what approach I will take when minimizing the cost. First of all, decided to divide the data into 10800 vectors (to obtain the theta), 4600 vectors (to decide on deletion/higher order) and 4600 vectors to verify (testing with data that was not used to tune). I plotted all of the features individually with the price of the house to gain insight of which features might be taken to a higher order and which features might be omitted. I also viewed the data set in excel and realized that I should remove the first two columns that represent the ID and the date of posting.

Code:

```
# Load data
data = np.loadtxt(os.path.join('Data', 'house_prices_data_training_data.csv'), delimiter=',')
X = data[:, 1:20]
m = X.shape
X = np.concatenate([np.ones((m[0], 1)), X], axis=1)
y = data[:, 0]

X1 = X[:10800,:]
Y1 = y[:10800]

X2 = X[10800:14400,:]
Y2 = y[10800:14400]

X3 = X[14400:18000,:]
Y3 = y[14400:18000]
```

```
for i in range (1, 19):
    fig = pyplot.figure()
    pyplot.plot(X[:,i],y , 'ro', ms=10, mec='k')
    pyplot.ylabel('Price')
    pyplot.xlabel( i)
```

Approach

I decided to decrease the cost by calculating the cost with the normal equation. I used the normal equation because the main issue with the normal equation approach is the time constraint. However, when I tried to obtain the theta for the primary data set, it was instantaneous. Also, using the gradient descent has a lot of issues to consider such as if the actual point where the theta converges is the global minimum.

I used the raw data set to obtain the preliminary error. The first trial showed an error of 1.88e10.

Adding a higher order

Now, given the goal is to decrease this cost/error, I took a look at the features that showed a parabolic relation and I added the square of all of these data. This did not work and the error increased around 100%. This approach was wishful and did not work due to the large nature of the data set. This might be because most of the data that I choose was not actually parabolic but had a few outliers that made the data seem parabolic.

I decided to change the approach by obtaining the cost if I square a feature while keeping the rest of the features the same. Then when I find which features decrease the cost, I will choose the feature that showed the most decrease. After choosing that feature, I will append it to the data and repeat this whole process again but with the new data matrix (with the added squared feature).

Code:

```
theta = normalEqn(X1, Y1);
print('Theta computed from the normal equations: {:s}'.format(str(theta)));
OriginalCost = computeCostMulti(X2,Y2,theta)
print(computeCostMulti(X2,Y2,theta)*(10**-10))

print("################################################################################

for i in range (1,19):
    k1 = [i]
    X4 = np.insert(X, i, X[:,i]**2, axis=1)

    X5= X4[:10800,:]
    X6 = X4[10800:14400,:]
    theta = normalEqn(X5, Y1);
    Cost = computeCostMulti(X6,Y2,theta)
    if(OriginalCost>Cost):
        print(i)
        print(Cost)
        print("")
print("################################################################################
```

This technique obtained a good result where the error decreased around 6% to reach 1.782e10. This was done when a higher order of the ninth feature (grade) was added to vector. Then I used the testing part of the vector to view the error. The tested error was even lower and reached 1.66e10. While this counter intuitive, I think the reason is that because this vector of set is more localized compared to the second set. Also, I tried using the cube of the feature to see if it will improve the cost. However, it increased it by 2e8 to become 1.8e10.

Code:

```
X4 = np.insert(X, 19, X[:,9]**2, axis=1) #adding the square of the feature
#X4 = np.insert(X4, 20, X[:,9]**3, axis=1) #adding the cube of the feature
X5= X4[:10800,:]
X6 = X4[10800:14400,:]
X7 = X4[14400:18000,:]
theta = normalEqn(X5, Y1);
OriginalCost = computeCostMulti(X6,Y2,theta)
#print(OriginalCost)
print(computeCostMulti(X7,Y3,theta)) #testing with vectors that were not used to tune)
```

I repeated this step again to find the next feature to square. The error decreased to 1.776e10 after adding the square of feature 13 (year of renovation). When the feature was cubed and added the error increased to 2.96e10. Testing the new thetas with the third set of vectors showed a 1.65e10 error.

Code:

```
X7 = X4

for i in range (1,19):
    k1 = [i]
    X7 = np.insert(X4, i, X4[:,i]**2, axis=1)

    X5= X7[:10800,:]
    X6 = X7[10800:14400,:]
    theta = normalEqn(X5, Y1);
    Cost = computeCostMulti(X6,Y2,theta)
    if(OriginalCost>Cost):
        print(i)
        print(Cost)
```

```
X7 = np.insert(X4, 20, X4[:,13]**2, axis=1) #adding the square of the feature
#X7 = np.insert(X7, 20, X4[:,13]**3, axis=1) #adding the square of the feature
X5= X7[:10800,:]
X6 = X7[10800:14400,:]
X8 = X7[14400:18000,:]
theta = normalEqn(X5, Y1);
OriginalCost = computeCostMulti(X6,Y2,theta)
#print(OriginalCost)
print(computeCostMulti(X8,Y3,theta)) #testing with vectors that were not used to tune)
```

I repeated the step again but I found that the error now decreased only to 1.774e10 (0.002e10 decrease) so I decided to stop adding higher polynomials after this point. the feature that was squared was the sixteenth feature(long). The error when tested with the third set was 1.65e10.

Code:

```
for i in range (1,19):
    k1 = [i]
    X8 = np.insert(X7, i, X7[:,i]**2, axis=1)

    X5= X8[:10800,:]
    X6 = X8[10800:14400,:]
    theta = normalEqn(X5, Y1);
    Cost = computeCostMulti(X6,Y2,theta)
    if(OriginalCost>Cost):
        print(i)
        print(Cost)
print("#########################################################################################
X8 = np.insert(X7, 21, X7[:,16]**2, axis=1)#adding the square of the feature
#X8 = np.insert(X8, 22, X7[:,16]**3, axis=1) #adding the cube of the feature
X5= X8[:10800,:]
X6 = X8[10800:14400,:]
X9 = X8[14400:18000,:]
theta = normalEqn(X5, Y1);
OriginalCost = computeCostMulti(X6,Y2,theta)
#print(OriginalCost)
print(computeCostMulti(X9,Y3,theta)) #testing with vectors that were not used to tune)
```

Deleting features

In the case of deciding which features to delete, there is a question that needs to be answered. How much does a feature actually cost me when it is added to thetas? Occam's Razor Law states that "Two explanations that account for all the facts, the simpler one is more likely to be correct". However, I found that all of the explanations I approached were fairly simple to python. The computation time was very low. I could delete features that will decrease the error when removed only. I could also delete features that will only increase the error by a slight margin so it is better to simplify it.
When I tried to figure out which features could be deleted to improve the error, I found that there were no features increased the error.
In order to remove a feature, I decided to remove features that only increase the error by 0.1%. The feature that was deleted was feature 1.

Code:

```
A =OriginalCost*0.001

for s in range (1,20):
    X8 = np.delete(X7, [s], axis=1)
    X5= X8[:10800,:]
    X6 = X8[10800:14400,:]
    theta = normalEqn(X5, Y1);
    Cost = computeCostMulti(X6,Y2,theta)

    if(OriginalCost+A>Cost):
        print(s)
        print(Cost)
```

```
X8 = np.delete(X7, [1], axis=1)
#X8 = X; #to compare results of the current work compared to the original
X5= X8[:10800,:]
X6 = X8[10800:14400,:]
X9 = X8[14400:18000,:]
theta = normalEqn(X5, Y1);
OriginalCost = computeCostMulti(X6,Y2,theta)
print(computeCostMulti(X9,Y3,theta)) #testing with vectors that were not used to tune)
print(theta)
```

Results

Below are the thetas obtained. The last 3 features are the squared values. The error from the third set was 1.653e10. The same test done to the original thetas showed 1.789e10. The improvement was around 9%. The results converged to 1.65e10 and any further improvements were minimal.

```
[ 1.30319085e+02  3.31031425e+04  8.98828353e+01  1.49460949e-01
  7.97346912e+03  6.79192682e+05  4.72565031e+04  3.28211241e+04
 -3.83213554e+05  5.34121882e+01  3.64549686e+01 -2.30457907e+03
 -3.81254492e+03 -4.79353659e+02  6.02460772e+05 -1.96464594e+05
  2.21269497e+01 -4.25055207e-01  3.02397072e+04  1.92638612e+00]
```

Verification

To verify the data, I used K-fold sampling with a K of 10. Which means I chose 10 subsets of the data to test and the rest to train. I did this by moving the first part of the data to the end and using it normally. By doing this in a for loop, I traversed the whole data set.

The result was that the mean error was 1.857e10 when the data was modified as did previously. However, when the data was taken raw with no modifications, the error was 2e10. This shows a clear improvement.
Also, due to the fact that removing feature 1 increased the error slightly, I tried not removing the feature and doing the same test. The result was an error of 18.49e10 which is a clear improvement compared to the main error. I commented the deletion line to not delete the first feature.

Code:

```
X0 = X8
#X0 = X #to compare with the original data
Y0 = y
K = 1800
sum = 0
for i in range (0,10):
    X0 =np.concatenate([X0[K:,:],X0[:K,:]], axis=0)
    Y0 =np.concatenate([Y0[K:],Y0[:K]], axis=0)
    Y01 = Y0[:16200]
    Y02 = Y0[16200:]
    X01= X0[:16200,:]
    X02 = X0[16200:,:]
    theta = normalEqn(X01, Y01)
    cost = computeCostMulti(X02,Y02,theta)
    sum = sum + cost

print(sum/10)
```

The thetas now are (after not removing the first feature):

```
[ 1.66361791e+02 -2.50938114e+04  4.17718649e+04  1.00454878e+02
  1.44591335e-01  5.53784705e+03  6.68579499e+05  4.56290473e+04
  3.32831264e+04 -3.57349915e+05  5.94635682e+01  4.09776301e+01
 -2.37763247e+03 -3.91937822e+03 -4.77997506e+02  5.97122634e+05
 -1.98223991e+05  2.18854819e+01 -4.79487841e-01  2.83729554e+04
  1.97890351e+00]
```

Regularization

The matrix form for regularization is:

$$\theta = \left(X^T X + \lambda \cdot L\right)^{-1} X^T y$$

The code now accounts for the regularization. I used the lambda to decrease the error and find features that could be removed. However, no thetas decreased with increasing lambda except when lambda was taken to extreme values.

Also, I normalized the values of X with featureNormalise to have lambda take the same effect on all of the features. I calculated the cost with changing Lambdas. I used lambda to decrease the error (to avoid over fitting). The value that showed the best result was around lambda = 0.03. The cost decreased with lambda to 18.47e10 (when tested with k-fold sampling).

Code:

```python
def normalEqnnorm(X, y,lambdaa):

    # ==================== YOUR CODE HERE ============================
    theta = np.linalg.pinv((X.transpose()@X) + lambdaa*(np.identity(X.shape[1])) )@X.transpose()@y
    # ================================================================
    return theta
```

```python
X0 = np.concatenate([np.ones((X8.shape[0], 1)), featureNormalize(X8[:,1:])[0]], axis=1)
Y0 = y
K = 1800
sum = 0
testlambda = [0 , 0.01, 0.02, 0.03, 0.1]
for l in testlambda:
    sum = 0
    for i in range (0,10):
        X0 =np.concatenate([X0[K:,:],X0[:K,:]], axis=0)
        Y0 =np.concatenate([Y0[K:],Y0[:K]], axis=0)
        Y01 = Y0[:16200]
        Y02 = Y0[16200:]
        X01= X0[:16200,:]
        X02 = X0[16200:,:]
        theta = normalEqnnorm(X01, Y01,l)
        cost = computeCostMulti(X02,Y02,theta)
        sum = sum + cost

    print(sum/10)
theta = normalEqnnorm(X01, Y01,0.03)
print(theta)
```

The best theta to describe the dataset is (with the last 3 terms the squared terms shown before):

```
[  533378.94132571   -17835.39314294    31696.13524298    64597.21815972
     6295.9002945      7033.46607482    53640.71295346    36884.8035979
    22781.61611433  -373204.2374423     56740.8262037     28066.17691962
   -69340.08527783 -1568274.22626954   -32862.9698597     83711.01713127
   -27232.54418538    20625.92011578   -11573.10823137   495734.47549701
  1580390.00655676]
```