ECE194N HW 1

Regression Report

February 6, 2019

Student: Erik Rosten
Perm Number: 7143571
Email: erosten@ucsb.edu

Department of Electrical and Computer Engineering, UCSB

Code

My code for the first coding exercise is below

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from mpl_toolkits.mplot3d import Axes3D
4
5
6
7
8  # square footage, number of bedrooms, selling price
9  def get_data_from_file():
10   return np.loadtxt("hw1_housing_prices.txt",dtype='float32',delimiter=',')
11
12 def get_train_test_sets(data):
13   train_X = data[0:(data.shape[0] - 10),0:2]
14   test_X = data[(data.shape[0] - 10):,0:2]
15   train_y = data[0:(data.shape[0] - 10),2].reshape(-1,1)
16   test_y = data[(data.shape[0] - 10):,2].reshape(-1,1)
17   return train_X, test_X, train_y, test_y
18
19 def visualize_data(data):
20   numInstances = data.shape[0]
21   fig = plt.figure()
22   ax = fig.add_subplot(111, projection='3d')
23   ax.scatter(data[0:numInstances - 10,0], data[0:numInstances - 10,1],
         data[0:numInstances - 10,2], c='r', marker='o',label='Training Points')
24   ax.set_xlabel('Square Footage')
25   ax.set_ylabel('Number of Bedrooms')
26   ax.set_zlabel('House Price')
27   ax.scatter(data[numInstances - 10:,0], data[numInstances - 10:,1], data[numInstances
         - 10:,2], c='b', marker='o', label='Testing Points')
28   ax.legend()
29   plt.show()
30
31 # use MSE as the loss function
32 def compute_loss(train_X, train_y, weights):
33   numInstances = train_X.shape[0]
34   coeff = 1 / (2 * numInstances)
35   loss = np.sum((np.matmul(train_X,weights) - train_y)**2)
36   return coeff * loss
37
38 def gradientDescent(train_X, train_y, weights, alpha, num_iters):
39   numInstances = train_X.shape[0]
40   losses = np.zeros((num_iters,))
41   for i in range(num_iters):
42     grad = np.matmul(np.transpose(train_X), np.matmul(train_X,weights) - train_y)
43     weights = weights - (alpha / numInstances) * grad
44     loss = compute_loss(train_X, train_y, weights)
45     losses[i] = loss
46     print('Iteration: {}, Loss: {}'.format(i,loss))
47   return weights, losses
48
```

```
49 def normalize_features(train_X):
50    mu = np.mean(train_X, axis = 0)
51    sigma = np.std(train_X, axis = 0)
52    return (train_X - mu) / sigma
53 
54 def plotLoss(losses):
55    fig = plt.figure()
56    ax = plt.axes()
57    ax.scatter(range(losses.shape[0]),losses)
58    ax.set_xlabel('Iterations')
59    ax.set_ylabel('Loss')
60    plt.show()
61 
62 def print_results(test_X, weights, test_y):
63    predicted_y = np.matmul(test_X, weights)
64    print('Weights are {}'.format(weights))
65    for i in range(test_X.shape[0]):
66       print('Predicted Value: {}, Actual Value: {}'.format(predicted_y[i], test_y[i]))
67 
68    totalMSE = compute_loss(test_X,test_y,weights)
69    print('Total Mean Squared Error: {}'.format(totalMSE))
70 
71 def append_ones(X):
72    ones = np.ones((X.shape[0],1))
73    return np.concatenate((ones,X), axis = 1)
74 
75 def run_regression():
76    # set up and visualize data
77    data = get_data_from_file()
78    train_X, test_X, train_y, test_y = get_train_test_sets(data)
79    visualize_data(data)
80    # train the weights
81    weights = np.zeros((3,1))
82    learning_rate = 0.01
83    num_iter = 1000
84    train_X = normalize_features(train_X)
85    train_X = append_ones(train_X)
86    weights, losses = gradientDescent(train_X, train_y, weights, learning_rate, num_iter)
87    # view loss and test set results
88    plotLoss(losses)
89    test_X = normalize_features(test_X)
90    test_X = append_ones(test_X)
91    print_results(test_X, weights, test_y)
92 
93 
94 
95 
96 if __name__ == '__main__':
97     run_regression()
```

The code is run through the *run_regression*() function, directly above. The data is parsed using the function *get_data_from_file*() on lines 9-10. The data is parsed into training/test sets and training/test outputs in the *get_train_test_sets*(*data*) from lines 12-17. These functions are called

on lines 78 and 79 of the *run_regression*() function.

Part a: Data Visualization

The function visualizing the data (lines 19-29) is called on line 79 of the code, and results in the plots below

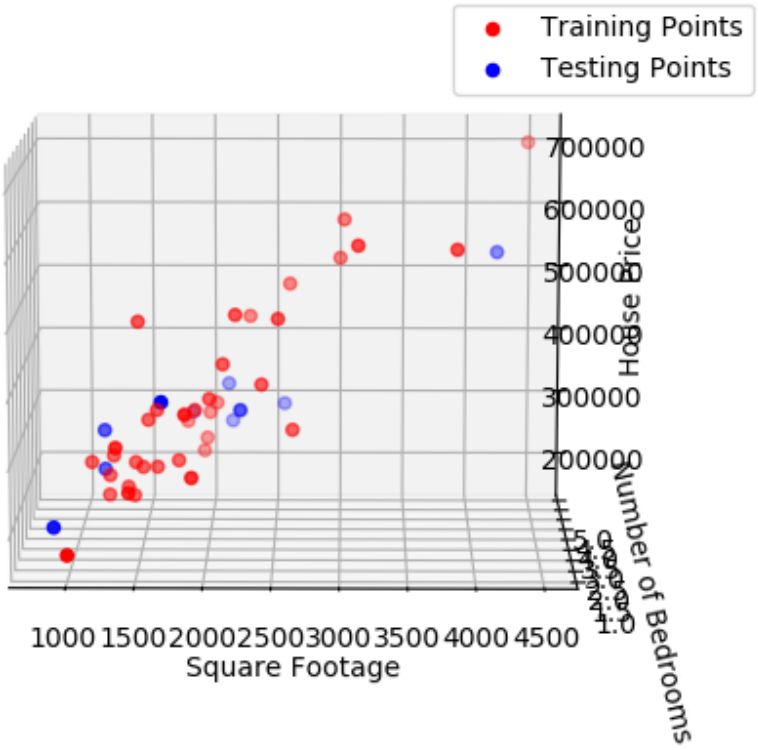

Figure 1: Housing price data visualization emphasizing square footage

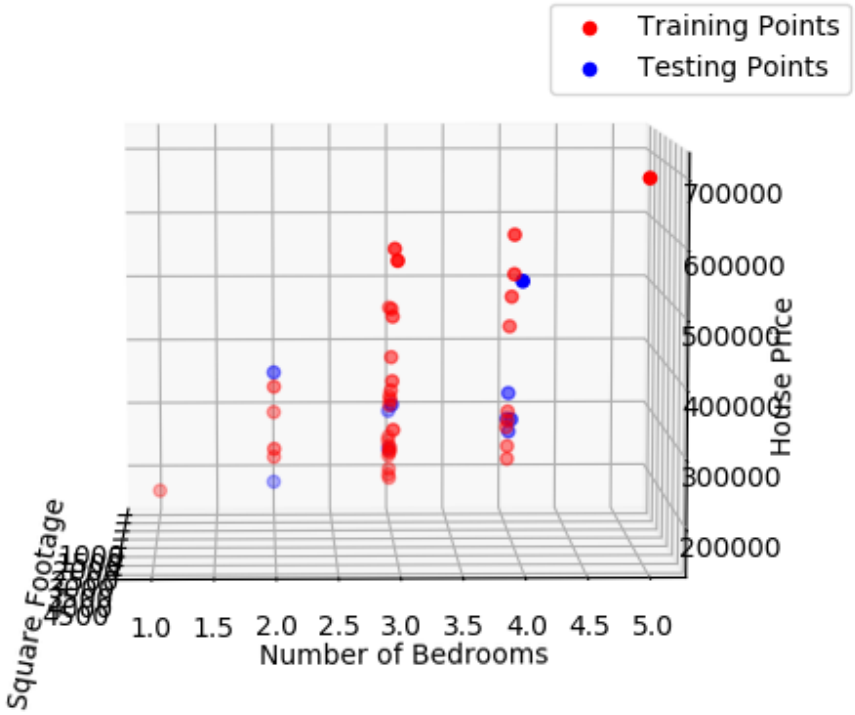

Figure 2: Housing price data visualization emphasizing number of bedrooms

Note that there does seem to be strong correlation between the features of number of bedrooms and square footage vs housing price.

Part b: Loss Function and Update Rule Derivations

I chose a mean squared error variant from Professor Ng's online machine learning course as a loss function, given below

$$Loss = L = \frac{1}{2m} \sum_{i=0}^{m-1} w^T x^{(i)} - y^{(i)}$$

where m denotes the number of training examples, and i represents the ith training example
This means that the update rule is given by

$$w = w - \frac{\alpha}{m} \cdot \frac{\partial L}{\partial w}$$

where

$$\frac{\partial L}{\partial w} = \sum_{i=0}^{m-1} x^{(i)} \left(w^T x^{(i)} - y^{(i)} \right)$$

and α is the learning rate.

Part c: Implementation Results

The loss function and update rule are implemented on lines 32-36 and 38-47 respectively. We also add bias terms using the function on lines 71-73 and called on lines 85 and 90. Note that on line 82, we set $\alpha = 0.01$. Commenting out the feature normalization function calls on lines 84 and 89 results in overflow, and infinite loss past iteration 30. Capping the iterations at 30 results in the loss function diverging. The results are below.

```
1  Iteration: 0, Loss: 1.3755880959712928e+20
2  Iteration: 1, Loss: 2.8623826716269936e+29
3  Iteration: 2, Loss: 5.956168552892905e+38
4  Iteration: 3, Loss: 1.2393850822994832e+48
5  Iteration: 4, Loss: 2.5789656027790302e+57
6  Iteration: 5, Loss: 5.366422167981406e+66
7  Iteration: 6, Loss: 1.1166681267082328e+76
8  Iteration: 7, Loss: 2.3236109015909124e+85
9  Iteration: 8, Loss: 4.835069160528522e+94
10 Iteration: 9, Loss: 1.006101915389009e+104
11 Iteration: 10, Loss: 2.0935399898990976e+113
12 Iteration: 11, Loss: 4.35632774599387e+122
13 Iteration: 12, Loss: 9.064833498322952e+131
14 Iteration: 13, Loss: 1.8862494087568018e+141
15 Iteration: 14, Loss: 3.92498861969569e+150
16 Iteration: 15, Loss: 8.167284555913646e+159
17 Iteration: 16, Loss: 1.699483577673081e+169
18 Iteration: 17, Loss: 3.536358273067903e+178
19 Iteration: 18, Loss: 7.358605872860908e+187
20 Iteration: 19, Loss: 1.531210251079199e+197
21 Iteration: 20, Loss: 3.186207922423325e+206
22 Iteration: 21, Loss: 6.629998014810883e+215
23 Iteration: 22, Loss: 1.3795984049579558e+225
24 Iteration: 23, Loss: 2.8707274945041238e+234
25 Iteration: 24, Loss: 5.973532818018213e+243
26 Iteration: 25, Loss: 1.2429983130148817e+253
27 Iteration: 26, Loss: 2.5864841681248646e+262
28 Iteration: 27, Loss: 5.382067121019877e+271
29 Iteration: 28, Loss: 1.1199235955951457e+281
30 Iteration: 29, Loss: 2.3303850207893536e+290
31 Weights are [[-4.42847822e+138]
32  [-1.01079658e+142]
33  [-1.45533065e+139]]
34 Predicted Value: [-2.15502458e+145], Actual Value: [345000.]
35 Predicted Value: [-4.26051386e+145], Actual Value: [549000.]
36 Predicted Value: [-2.18534847e+145], Actual Value: [287000.]
37 Predicted Value: [-1.68196887e+145], Actual Value: [368500.]
38 Predicted Value: [-2.26216756e+145], Actual Value: [329900.]
39 Predicted Value: [-2.59472109e+145], Actual Value: [314000.]
```

```
Predicted Value: [-1.21296071e+145], Actual Value: [299000.]
Predicted Value: [-8.61202041e+144], Actual Value: [179900.]
Predicted Value: [-1.87200153e+145], Actual Value: [299900.]
Predicted Value: [-1.2159931e+145], Actual Value: [239500.]
```

One problem might be that the learning rate is too high. Since the features are not normalized, the weights are updated with huge values, and this might cause the gradient descent to be bouncing more and more away from the local minimum, as it tries to take too large of steps. By drastically lowering the learning rate to $\alpha = 0.00000001$ and $num\_iter = 2500000$ we obtain the results below.

```
Weights are [[133.15333576]
 [169.7988393 ]
 [269.66280931]]
Predicted Value: [363222.92995087], Actual Value: [345000.]
Predicted Value: [716913.91220327], Actual Value: [549000.]
Predicted Value: [368316.89512974], Actual Value: [287000.]
Predicted Value: [283217.74754199], Actual Value: [368500.]
Predicted Value: [380951.94410688], Actual Value: [329900.]
Predicted Value: [437085.42504439], Actual Value: [314000.]
Predicted Value: [204700.74891821], Actual Value: [299000.]
Predicted Value: [145341.09003409], Actual Value: [179900.]
Predicted Value: [315679.25494815], Actual Value: [299900.]
Predicted Value: [205210.1454361], Actual Value: [239500.]
```

Part d: Data Normalization Results

Using data normalization, implemented on lines 49-52 and called on lines 84 and 85, the results are much better. By setting $\alpha = 0.000001$ and $num\_iter = 2500000$, we arrive at the results below. We set alpha to be low and increase the number of iterations to converge on a more precise global minimum.

```
Weights are [[317243.74742327]
 [ 94808.58668629]
 [ 16611.05753307]]
Predicted Value: [345177.50501465], Actual Value: [345000.]
Predicted Value: [565215.86835649], Actual Value: [549000.]
Predicted Value: [348346.56427655], Actual Value: [287000.]
Predicted Value: [253203.61779896], Actual Value: [368500.]
Predicted Value: [335106.57110824], Actual Value: [329900.]
Predicted Value: [391128.86713761], Actual Value: [314000.]
Predicted Value: [225457.1082145], Actual Value: [299000.]
Predicted Value: [167427.73976407], Actual Value: [179900.]
Predicted Value: [315599.61574481], Actual Value: [299900.]
Predicted Value: [225774.01809641], Actual Value: [239500.]
```

where we have shown the weights and prediction results for comparison above.

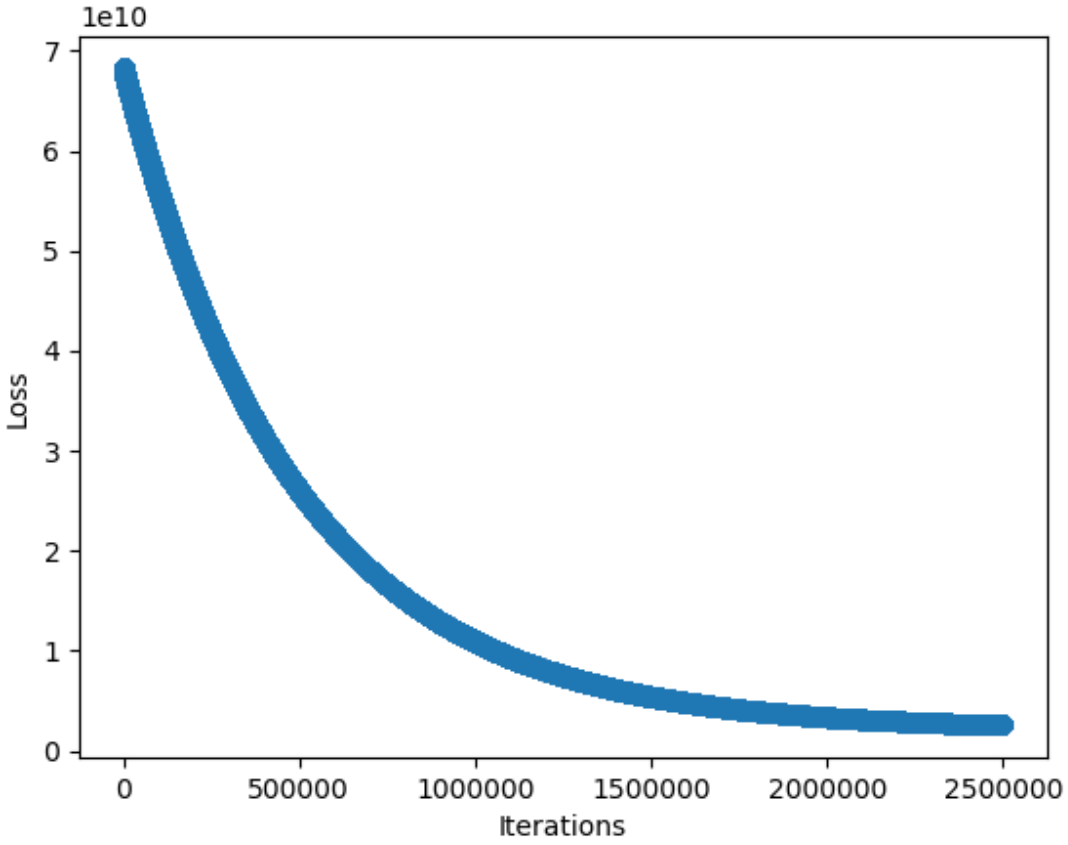

Figure 3: Loss vs 2500000 Iterations

Part e: Discussion of Results With and Without Normalization

Using mean normalization gives much better results using an eyeball comparison of parts c and d. Without normalizing our features, there are significant differences in the final weights, where without feature normalization the weights are around 3 orders of magnitude smaller. Another characteristic of not normalizing features is the vast reduction of the learning rate in our case. Since the weights were being updated with such a large value, we had to drastically decrease our learning rate to get convergence. One last observation is that with feature normalization, we force every feature to contribute equally to the weights and result in a more accurate solution.

Part f: Learning Rate Adjustments

Increasing the learning rate by 10000 to $\alpha = 0.0001$ with the same number of iterations gives interesting results. We arrive at the closest possible local minimum quite early, but the learning rate is not high enough such that it moves far away. However, the learning rate is high enough that it cannot converge any closer. The results are below.

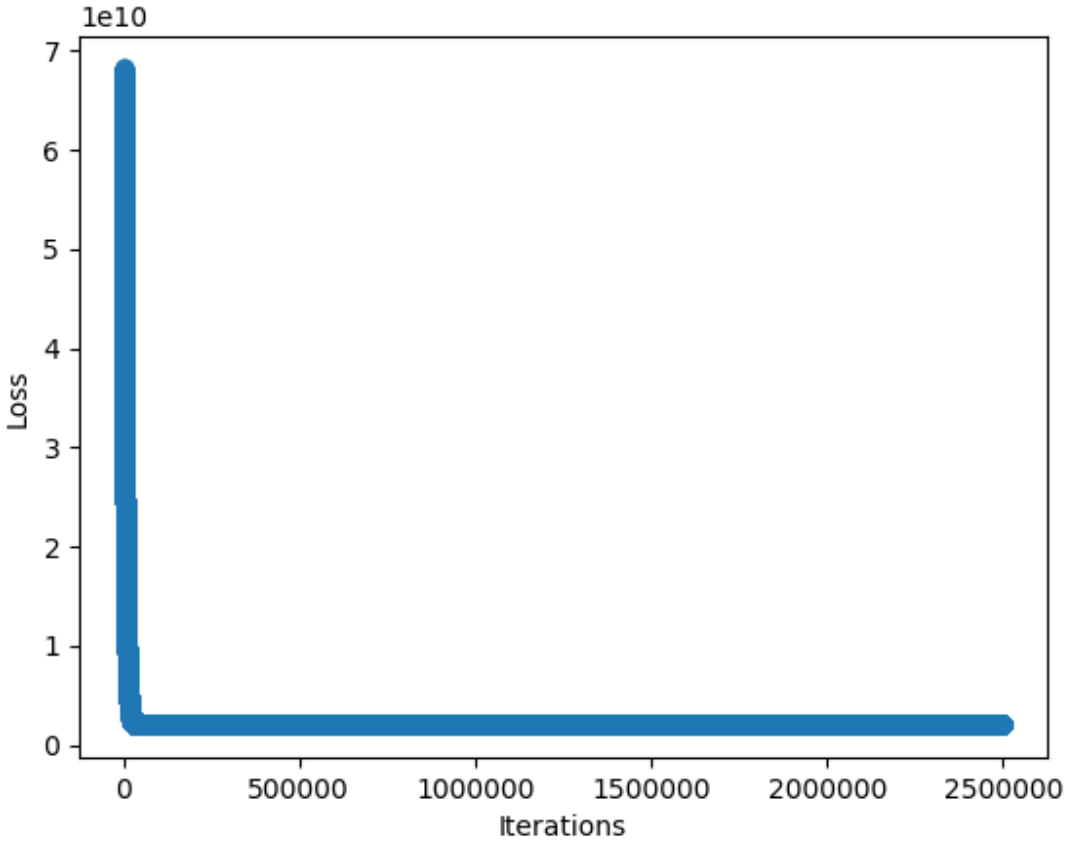

Figure 4: Loss using MSE vs 2500000 Iterations using $\alpha = 0.0001$

```
Weights are [[345613.38748577]
 [114207.79069084]
 [  -372.58976509]]
Predicted Value: [360994.80418318], Actual Value: [345000.]
Predicted Value: [626056.1896846], Actual Value: [549000.]
Predicted Value: [364812.29866885], Actual Value: [287000.]
Predicted Value: [302395.99004392], Actual Value: [368500.]
Predicted Value: [374960.33971219], Actual Value: [329900.]
Predicted Value: [416348.47762905], Actual Value: [314000.]
Predicted Value: [242875.01553193], Actual Value: [299000.]
Predicted Value: [199069.13044895], Actual Value: [179900.]
Predicted Value: [325364.85224661], Actual Value: [299900.]
Predicted Value: [243256.76974561], Actual Value: [239500.]
```

It can be seen that this is not the true local minimum, as the results in part d are better. This, along with the divergence seen in part c show the impact of the learning rate on our solution.