Implementation 1

CS434, 4-15-2018

Team: Rex Henzie, Benjamin Richards, and Michael Giovannoni

Part 1: Linear Regression

1.1

```
 Learned weight vector=[[ 39.58432122]
                        [ -0.10113705]
                        [  0.04589353]
                        [ -0.00273039]
                        [  3.0720134 ]
                        [-17.22540718]
                        [  3.71125235]
                        [  0.00715862]
                        [ -1.5990021 ]
                        [  0.37362337]
                        [ -0.01575642]
                        [ -1.02417703]
                        [  0.00969321]
                        [ -0.58596927]]
```

1.2

```
Training ASE: 22.081273187
Test ASE: 11.5203964148
```

1.3

```
Learned weight vector=[[-0.09793424]
                        [ 0.04895868]
                        [-0.02539285]
                        [ 3.45087927]
                        [-0.35545893]
                        [ 5.81653272]
                        [-0.00331448]
                        [-1.02050134]
                        [ 0.22656321]
                        [-0.01224588]
                        [-0.38802988]
                        [ 0.0170215 ]
                        [-0.48501296]]
```

Training ASE without dummy variable: 24.4758827846
Test ASE without dummy variable: 11.9438538186

The ASE for both training and test went up slightly (which means it is less accurate). The dummy variable adds another dimension to the data, which helps the program to better predict the median value, according to what we were taught in class. The dummy variable helps the training data's ASE more, because it is a part of the training data, but it does still help the test data's ASE to a lesser extent.

1.4

Below is the graph for part 4 of problem 1.

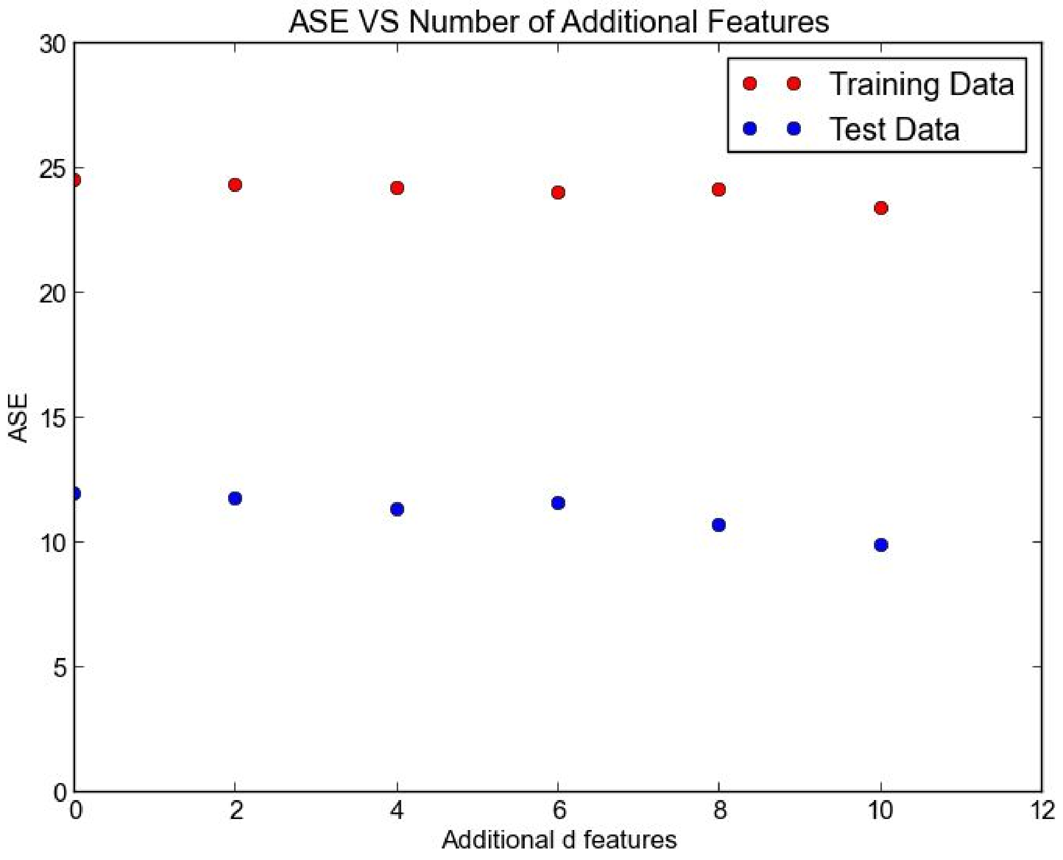

Figure 1: ASE vs. Number of Additional Features graph

In general, adding additional d features will cause the ASE of test and training will get better; however, at 6 additional features it was slightly worse than 4 additional features. This changes depending on the randomized d values (when tested multiple times, for instance, I noticed that 10 additional features wasn't always much better than 8, while other times it is a fair amount better). The reason for this is the randomization between the additional d features (if they are more similar to each other, it probably won't help much).

Part 2: Logistic Regression with Regression and Gradient Descent

2.1

After implementing the logistic regression algorithm we experimented with different learning rates and different numbers of iterations. In figure 2 below we can see the results of three different learning rates applied over 100 iterations. We notice that as the learning rate shrinks, the gradient magnitude (norm) shrinks at a slower and slower rate. Ultimately we decided to use a learning rates of 0.001, a learning rate that produced an accurate model and allowed a high number of iterations without producing an overflow error.

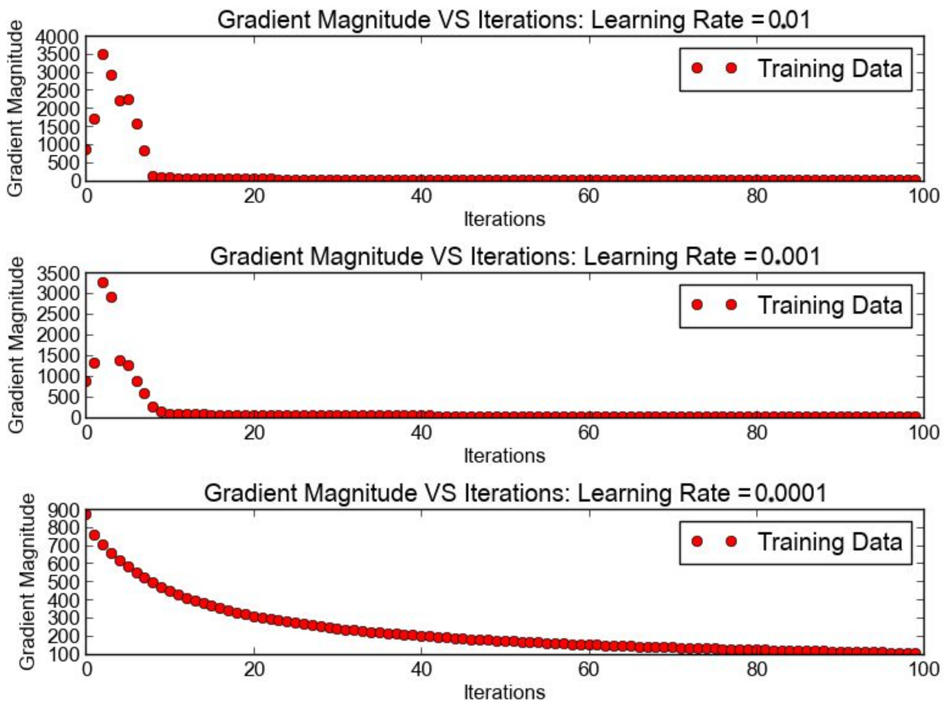

Figure 2: Gradient magnitude vs. # of iterations

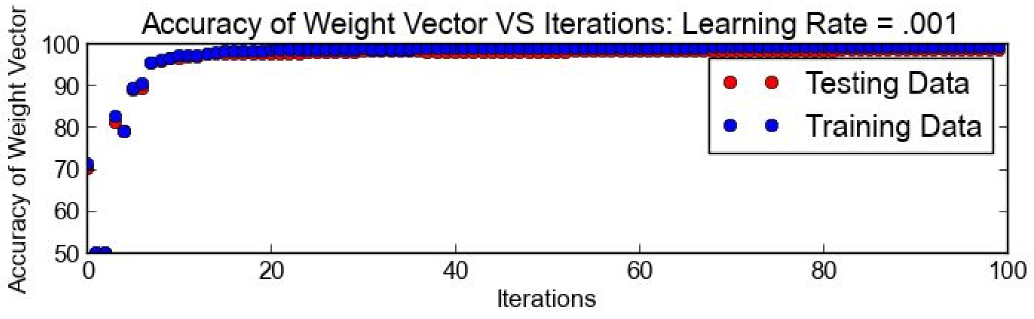

Figure 3: Accuracy vs. iterations

When experimenting with the number of iterations we found that model is always slightly less accurate when applied to the test data than when applied to the training set and never more accurate. As can clearly be seen in figure 3 the accuracy of the model increases as you increase the number of iterations. When run at a learning rate of 0.001 the model was found to be 98.5% accurate when applied to the test set after 100 and after running for 1000 iterations it was 98.625% accurate. Clearly the diminishing returns of the increased accuracy after every iteration makes using very high numbers of gradient descent iterations unnecessary. With this consideration we decided to use 100 iterations, a number that allowed the program to run quickly and produced fairly accurate results.

2.2

```
Given: training examples ($X_i$, $y_i$), i = 1, … n
Let w = (0, … , 0)
for j = 0, . . ., desired iterations
        del = (0, … , 0)
        for i = 1, . . . n
                yhat = sigmoid(X[i], w)
                del = del + (yhat - y[i]) * X[i]
        w = w - learning rate * (del + $\lambda$ * w)
```

2.3

Training rate = 0.001, 100 iterations

Lambda	Training accuracy	Test accuracy
100	62.5	62.625
10	99.1	98.125
1	99.4	98.5
.1	99.4	98.5
.01	99.4	98.5
.001	99.4	98.5
.0001	99.4	98.5
.000001	99.4	98.5

As we decrease lambda we see the accuracy rise, eventually leveling off. This suggests that with a sufficiently small lambda we can safeguard against overfitting while still maintaining the accuracy of the model.