CS 434
Julian Weisbord
04/15/18

Part 1

To run this program:
cd /scratch/cs434spring2018/bin/bash
2-1: python 2.7:
source env_2.7/bin/activate

Add the housing and usps dataset to assignment1 directory
cd assignment1

execute: "python linear_regression.py"

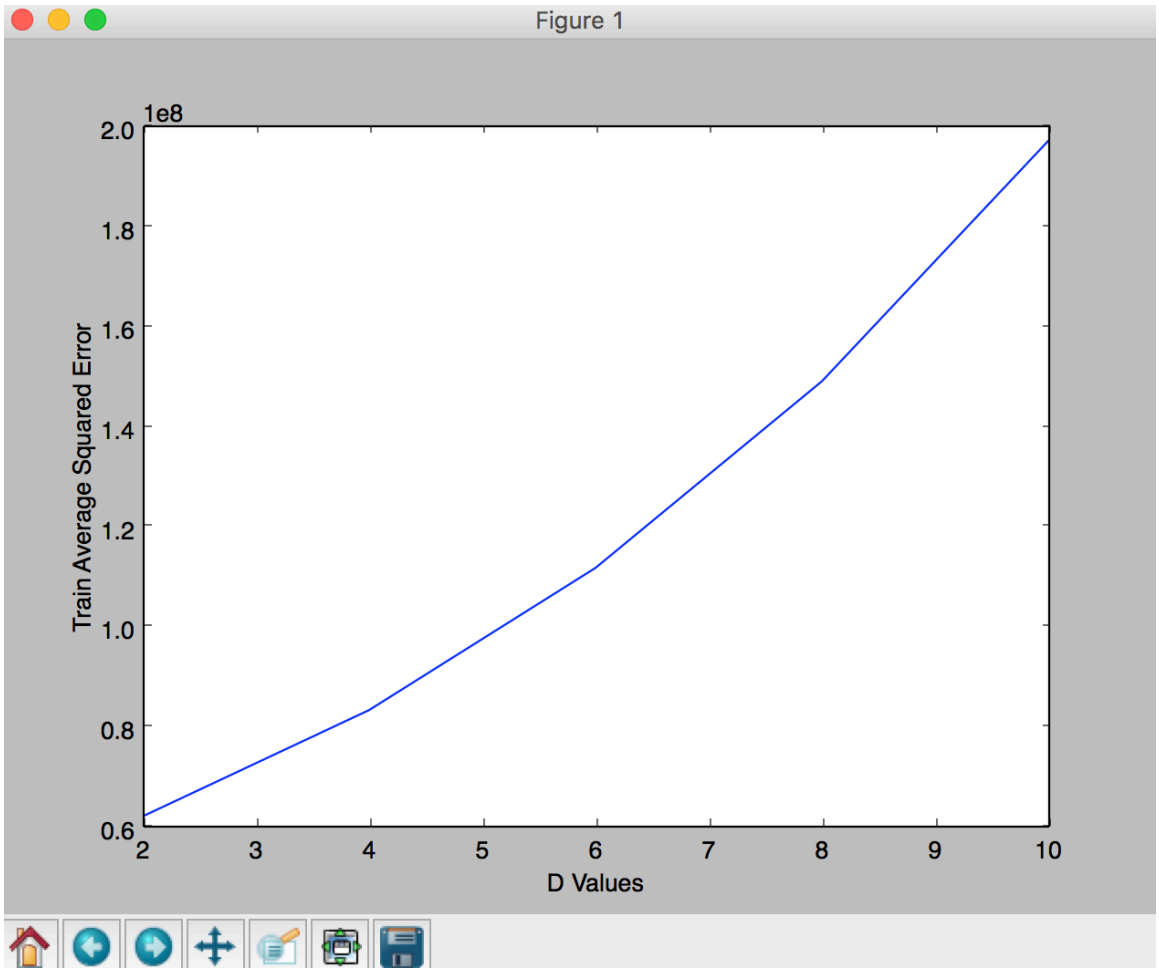
1. Weight Vector:

```
('w_1: ', array([ 3.95843212e+01, -1.01137046e-01,  4.58935299e-02, -2.73038670e-03,
        3.07201340e+00, -1.72254072e+01,  3.71125235e+00,  7.15862492e-03,
       -1.59900210e+00,  3.73623375e-01, -1.57564197e-02, -1.02417703e+00,
        9.69321451e-03, -5.85969273e-01]))
```

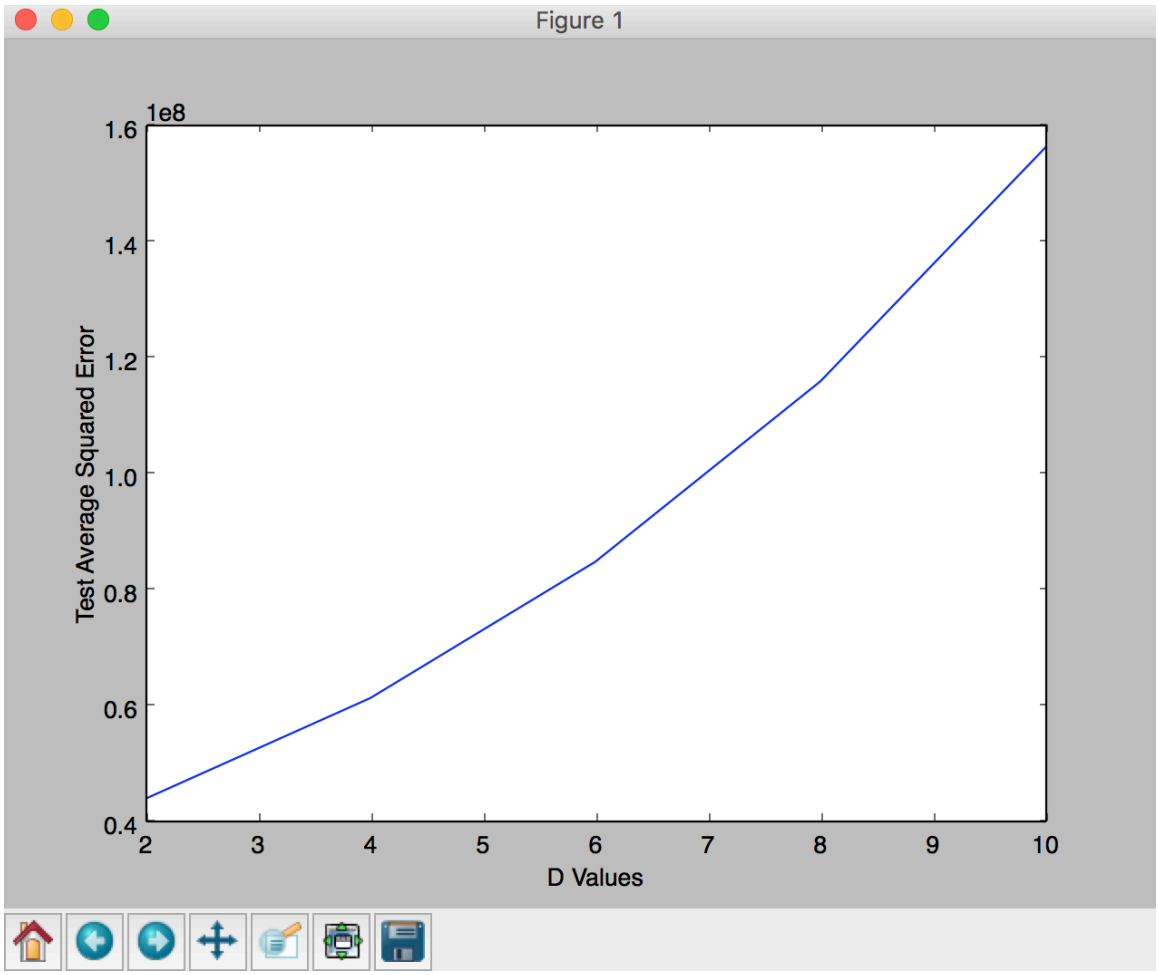
2.
Train ASE = 22.081273187013167
Test ASE = 22.638256296591674

3.
Getting rid of the dummy column of ones increases the train and test average sum error slightly. This likely means that the column of one's negatively correlates with the other features. Having the bias causes less weight to be given to the important features when predicting outcome. This causes an increase in error.
Train ASE w/out column of ones= 24.475882784643677
Test ASE w/out column of ones= 24.292238175661737

4.
Train ASE vs. D value plot:



Test ASE vs. D value plot:



The trend that I noticed for the training and test data is that they show a very similar graph that looks like a linear function. More features provide worse prediction performance at the testing stage. Randomly adding features that don't positively correlate, the more variables that we add which negatively correlate with performance, the larger the ASE value. This won't always be the case when randomly choosing additional features, but with our data, they negatively affect the performance.

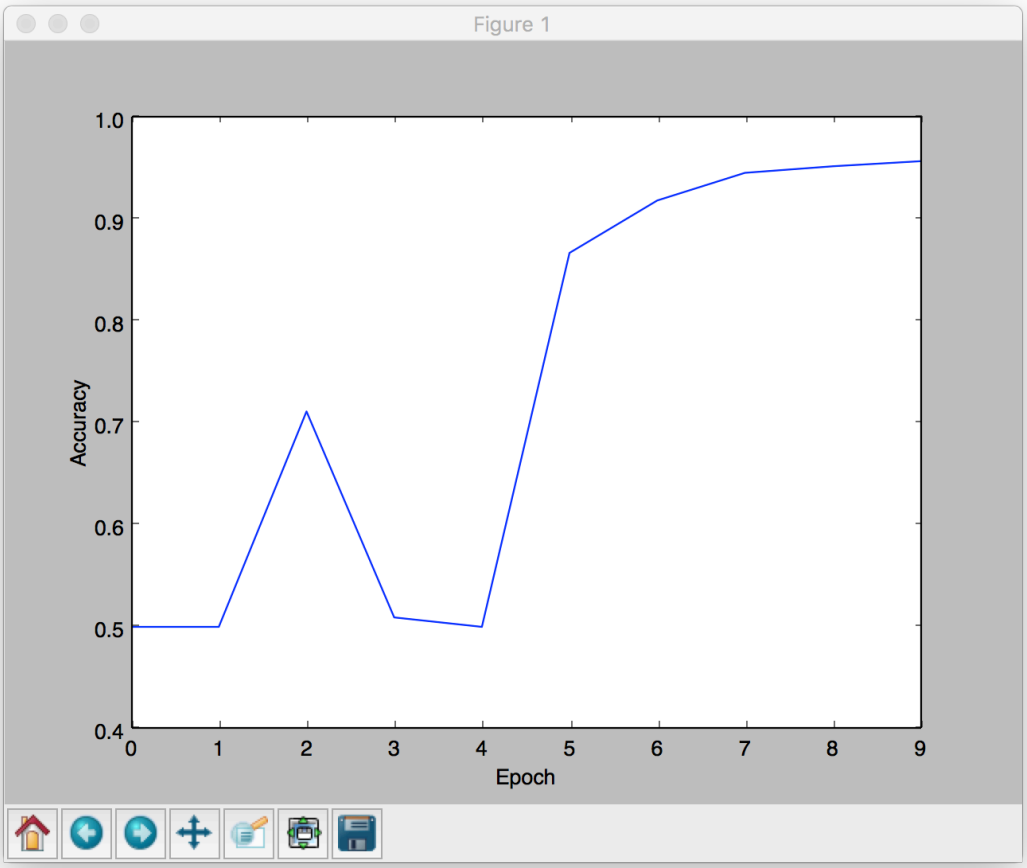
Part 2

execute: "python gradient_descent_logistic_regression.py"

1. Gradient Descent:
    With 10 epochs
    .0000001: Accuracy 47.857%
    .0001: Accuracy 88.64%

.01: Accuracy 95.64%
1: Accuracy 95.7%

The overall trend looks like this:



2. Gradient Descent pseudocode for batch learning with L2 Regularization:

With inputs X[1, … ,n] and labels Y[1, … , n]:

```
While(Iterations < Number of Epochs):
        Gradient = zeros(Number of Features)
        For i in example number:
                Y_hat = 1/(1 +e^(-w.T * X[i])
                Loss = y_hat – y
                gradient += loss * X[i]
```

w += -LEARNING_RATE * gradient + .5 * lambda * |w|^2

3.
After implementing L2 Regularization, I noticed that the test data started to perform better than the training data did as lambda got closer and closer to zero. For values of lambda >= 0.1, L2 Regularization of the training and testing data performed worse than plain logistic regression. I also found that there was almost no difference with regularization on the training data vs. no regularization on the training data, the real difference comes from comparing regularization on the training and testing data sets.

For a lambda value of 100:
After 10 epochs, logistic regression on the testing data reached a max accuracy of 50%
After 10 epochs, logistic regression on the training data reached a max accuracy of 50%

For a lambda value of .1:
After 10 epochs, logistic regression on the testing data reached a max accuracy of 50%
After 10 epochs, logistic regression on the training data reached a max accuracy of 50%

For a lambda value of .001:
After 10 epochs, logistic regression on the testing data reached a max accuracy of 50%
After 10 epochs, logistic regression on the training data reached a max accuracy of 68.8%

For a lambda value of .00000001:
After 10 epochs, logistic regression on the testing data reached a max accuracy of 95.21%
After 10 epochs, logistic regression on the training data reached a max accuracy of 95.625%