EECS 391: Project 2

1.

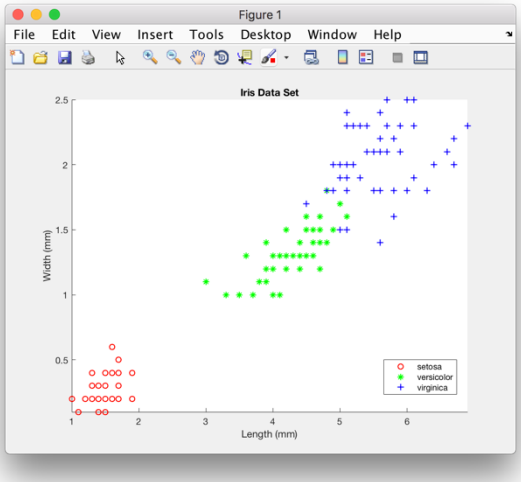
a. First, I used Matlab to auto-generate a function, *importfile.m*, that will import the *irisdata.csv*, and properly parse and input the data into the workspace. I then define two variables, X and Y that represent the pedal_length and pedal_width, respectively. I then partition the data into the various classes, or types of flower. There is most likely a more elegant solution to do this, but this is my first time using Matlab.

```
irisdata = importfile('irisdata.csv', 2, 151);

X = irisdata{:,3};
Y = irisdata{:,4};

Xsetosa = X(1:50);
Xversicolor = X(51:100);
Xvirginica = X(101:150);
Ysetosa = Y(1:50);
Yversicolor = Y(51:100);
Yvirginica = Y(101:150);
```

I then plot this data, using the same color scheme as seen in the lecture, for simplicity. I graph the Setosa class for reference, however we do not consider the Setosa class for the rest of the project.

```
%% 1(a) - plot the data
scatter(Xsetosa, Ysetosa, [], 'red', 'o')
hold on, scatter(Xversicolor, Yversicolor, [], 'green','*')
hold on, scatter(Xvirginica, Yvirginica, [], 'blue','+')

xlabel('Length (mm)'), ylabel('Width (mm)')
title('Iris Data Set')
xlim([min(X) max(X)]), ylim([min(Y) max(Y)])
```

This results in the distribution from the lecture slides:

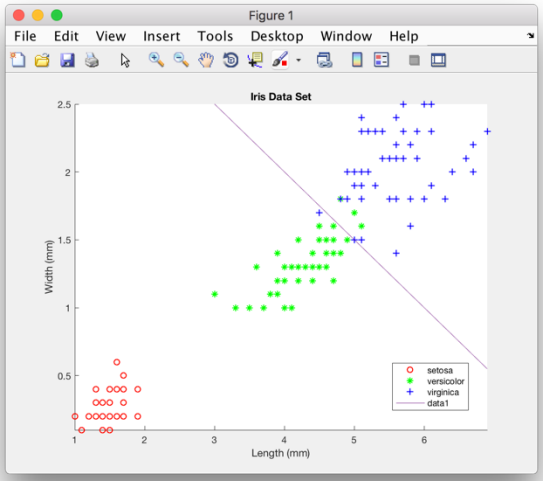


b. I then wrote an arbitrary line that roughly separates the two classes, and graphed the output.

```
%% 1(b) - plot a linear decision boundary by hand
% note that the decision boundary I am using is between the versicolor and
% virginica classes
```

```
boundary = -1/2*X + 4;
hold on, plot(X, boundary)
```

We now have the following graph:



c. An example of a simple threshold classifier would be using the arbitrary line as a boundary for classifying data between the two classes. In this model, all values < the decision boundary will be classified as "Versicolor". All values > the decision boundary will be classified as "Virginica". All values that are on the decision boundary line cannot specifically be classified as either, and are also considered misclassified. The output of a classifier for a specific classification is the number of points incorrectly classified. For this data set, we get missclassifiedVersicolor = 5 and missclassifiedVirginca = 5.

```
%% 1(c) - define a simple threshold classifier
% this counts the number of data points that would be miss classified for
% each classification.
missclassifiedVersicolor = 0;
for i = 1:50
   if Yversicolor(i) >= -1/2*Xversicolor(i) + 4
       missclassifiedVersicolor = missclassifiedVersicolor + 1;
   end
end
for i = 1:50
   if Yvirginica(i) <= -1/2*Xvirginica(i) + 4
       missclassifiedVersicolor = missclassifiedVersicolor + 1;
   end
end

missclassifiedVirginca = 0;
for i = 1:50
   if Yvirginica(i) <= -1/2*Xvirginica(i) + 4
       missclassifiedVirginca = missclassifiedVirginca + 1;
   end
end
for i = 1:50
   if Yversicolor(i) >= -1/2*Xversicolor(i) + 4
       missclassifiedVirginca = missclassifiedVirginca + 1;
   end
end
```

d. For this part, I plotted two circles, using a plotcircle function that I found on mathworks.com:

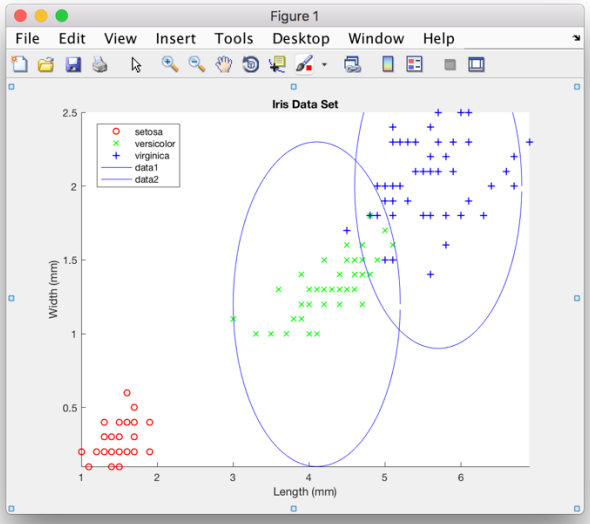
```
function plotcircle(x1,y1,r,c)
% x1,y1: center
% r = radius
```

```
theta = 0:0.05:2*pi;
hold on;
plot(x1+r*cos(theta),y1+r*sin(theta),c);
end
```

Using the generated graph, I estimated two circles, with centers at (4.1, 1.2) and (5.7, 2) respectively. I plotted both with radius = 1.1.

```
plotcircle(4.1, 1.2, 1.1, 'b')
plotcircle(5.7, 2, 1.1, 'b')
```

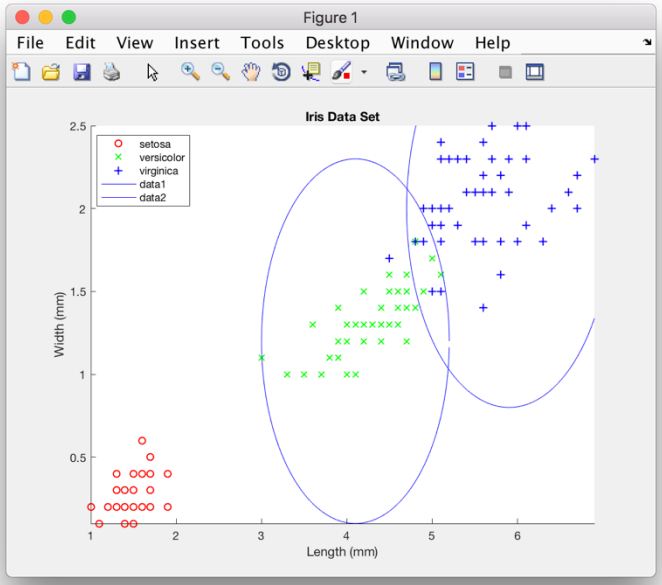
At this point, I got a graph that contains the two circles. In order to get most of the dataset in both circles, a small amount of overlap exists. To output the performance of this classifier, I created a function that counts the number of misclassified points. I performed the classification in the same way as part b.



To check the performance of this classifier, I counted all the values outside of the boundary that should be and all the values inside the boundary that shouldn't be.

```
missclassifiedVersicolor = 0;
for i = 1:50
   if (4.1-Xversicolor(i))^2 + (1.2-Yversicolor(i))^2 > 1.1^2
        missclassifiedVersicolor = missclassifiedVersicolor + 1;
   end
end
for i = 1:50
   if (4.1-Xvirginica(i))^2 + (1.2-Yvirginica(i))^2 <= 1.1^2
        missclassifiedVersicolor = missclassifiedVersicolor + 1;
   end
end

missclassifiedVersicolor

missclassifiedVirginica = 0;
for i = 1:50
   if (5.7-Xvirginica(i))^2 + (2.0-Yvirginica(i))^2 > 1.1^2
        missclassifiedVirginica = missclassifiedVirginica + 1;
   end
end
for i = 1:50
   if (5.7-Xversicolor(i))^2 + (2.0-Yversicolor(i))^2 <= 1.1^2
        missclassifiedVirginica = missclassifiedVirginica + 1;
   end
end

missclassifiedVirginica
```

When doing this, I got values of missclassifiedVersicolor = 8 and missclassifiedVirginica = 9. To get a third center, I moved the Virginica decision boundary farther right, and increased the radius to allow it to get a larger portion of the Virginica data. This provided better results, with values missclassifiedVersicolor = 8 and missclassifiedVirginica = 6.



2.

a. To calculate the mean-squared error for the iris data given a decision boundary, I created a function that takes in the parameters of the decision boundary, and the class data vectors. We will be using the following formula to determine the error (Lecture 20, slide 11):

$$E = \frac{1}{2}\sum_{n=1}^{N}(\mathbf{w}^T\mathbf{x}_n - c_n)^2$$

In addition, we will take the sigmoid of the weights to keep the weight between 0 and 1:

$$E = \frac{1}{N}\sum_{n=1}^{N}(\sigma(W^T X_N) - c_n)^2$$

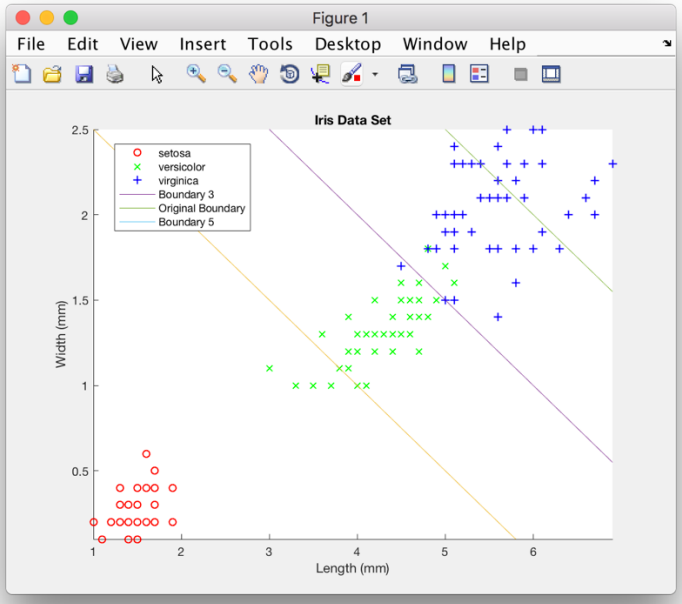
Since the data is symmetric with respect to the decision boundary, we need the sources of error from both classes to cancel each other out. To do this I subtract 1 from the sigmoid of the weight of versicolor, since it is below the decision boundary.

```
function error = mse(m, b, y, length_virginica, width_virginica,
length_versicolor, width_versicolor)
%UNTITLED Summary of this function goes here
```

```
%   Detailed explanation goes here

error = 0;

for i = 1:50
    weight = b + (m * length_virginica(i)) + (y * width_virginica(i));
    error = error + (sigmoid(weight))^2;
end

for i = 1:50
    weight = b + (m * length_versicolor(i)) + (y * width_versicolor(i));
    error = error + (sigmoid(weight) - 1)^2;
end

error = 1/2 * (error / 100);
end

function sigvalue = sigmoid(x)
sigvalue = 1/(1 + exp(-x));
end
```

I then ran this function with the values m = -1/2, b = 4, y = -1. The reason I used y = -1 is because -1 is the coefficient in front of the y term when we rearrange the formula of a line to 0 = mx + b – y. With this initial data, we get error = 0.0640.

b. I then made two different decision boundaries, y = -1/2x + 5 and y = -1/2x + 3. Using these two boundaries, ran my MSE function to determine the error of both. I got error3 = 0.1011 and error5 = 0.0869. Since these errors are both larger than the error of the decision boundary that looks visually more accurate, the MSE function I wrote appears to be accurate. I have included a graph of the other decision boundaries below.



c. To find the gradient, I took the derivative of the MSE with respect to w, as seen on page 16 of lecture 20. As a result, we get the following equation:

$$\frac{\partial MSE}{w} = \frac{1}{N}\sum_{n=1}^{N} 2(\sigma(W^T X_N) - c_n)\sigma`(W^T X_N)$$

From slide 32 of lecture 20, we know the derivative of the sigmoid function:

$$\begin{aligned}\frac{d\sigma(x)}{dx} &= \frac{d}{dx}\frac{1}{1+\exp(-x)} \\ &= \sigma(x)(1-\sigma(x))\end{aligned}$$

From slide 16 of lecture 20, we know that

$$W^T X_N = [w_o, w_1, w_2]\begin{bmatrix}1\\x\\y\end{bmatrix}$$

d. In part c, we found the vector form of the gradient. To determine the vector, we multiply X_N by the gradient found in part c.

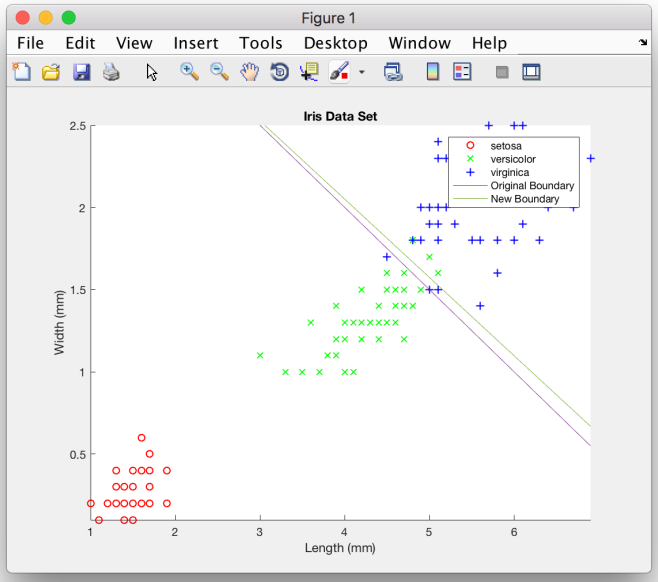
$$\frac{\partial MSE}{w} = \begin{bmatrix}\frac{1}{N}\sum_{n=1}^{N} 2(\sigma(W^T X_N) - c_n)\sigma`(W^T X_N)(1) \\ \frac{1}{N}\sum_{n=1}^{N} 2(\sigma(W^T X_N) - c_n)\sigma`(W^T X_N)(X_N) \\ \frac{1}{N}\sum_{n=1}^{N} 2(\sigma(W^T X_N) - c_n)\sigma`(W^T X_N)(Y_N)\end{bmatrix}$$

e. To compute the gradient found in part d, I made a function similar to the one in part a, with a few modifications.

```
function error = gradient(m, b, y, length_virginica, width_virginica,
length_versicolor, width_versicolor)
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here

error0 = 0;
error1 = 0;
error2 = 0;

% 1
for i = 1:50
    weight = b + (m * length_virginica(i)) + (y * width_virginica(i));
    error0 = error0 + (sigmoid(weight)) * d_sigmoid(weight) * 1;
end

for i = 1:50
    weight = b + (m * length_versicolor(i)) + (y * width_versicolor(i));
    error0 = error0 + (sigmoid(weight) - 1) * d_sigmoid(weight) * 1;
end

% Xn
for i = 1:50
    weight = b + (m * length_virginica(i)) + (y * width_virginica(i));
    error1 = error1 + (sigmoid(weight)) * d_sigmoid(weight) *
length_virginica(i);
end

for i = 1:50
    weight = b + (m * length_versicolor(i)) + (y * width_versicolor(i));
    error1 = error1 + (sigmoid(weight) - 1) * d_sigmoid(weight) *
length_versicolor(i);
end

% Yn
for i = 1:50
    weight = b + (m * length_virginica(i)) + (y * width_virginica(i));
```

```
    error2 = error2 + (sigmoid(weight)) * d_sigmoid(weight) *
width_virginica(i);
end

for i = 1:50
    weight = b + (m * length_versicolor(i)) + (y * width_versicolor(i));
    error2 = error2 + (sigmoid(weight) - 1) * d_sigmoid(weight) *
width_versicolor(i);
end


error = 2/100 .* [error0, error1, error2];
end

function sigvalue = sigmoid(x)
sigvalue = 1/(1 + exp(-x));
end

function sigvalue = d_sigmoid(x)
sigvalue = sigmoid(x) * (1 - sigmoid(x));
end
```

This function simply executes each line in part d, and outputs the answer as an array of values. These values represent the change in w_o, w_1, and w_2 (which I called m, b, and y originally). Upon executing this function on my original decision boundary, I get the following output: [-0.0172 -0.0095 0.0137]. When adding these values to my decision boundary values, I get: -0.4828 4.0095 - 1.0137. I then plotted this new line to compare it to the old line.



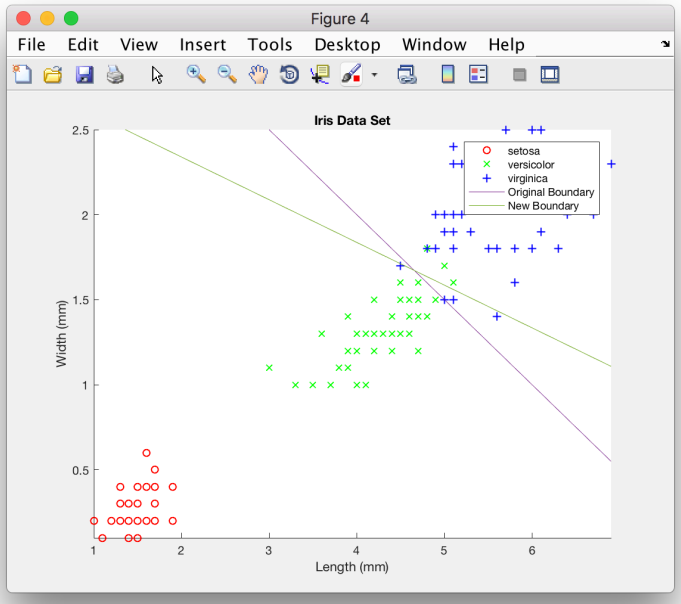
We can see the change from the new line to old line.

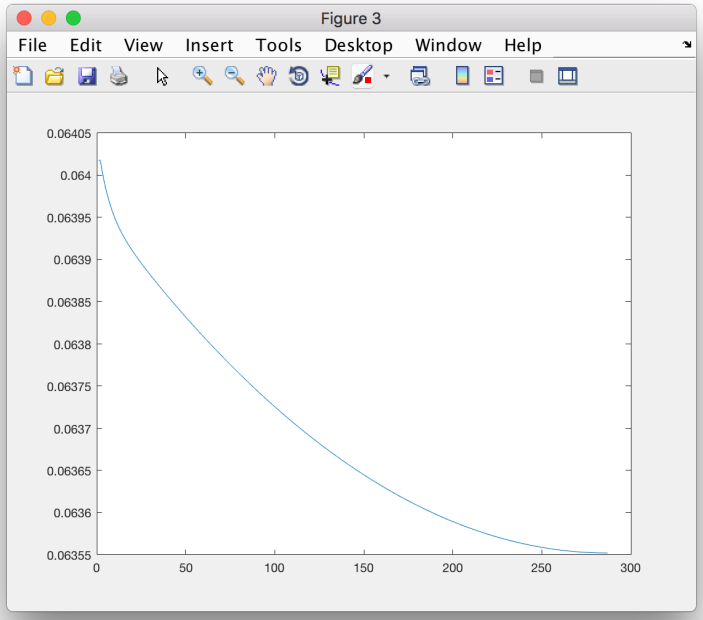
3.

a. To implement the gradient descent to optimize the decision boundary, the first realization I had to make was that it the gradient function always outputs a vector of length 1. Using this vector will cause use to "overshoot" the local minimum that the gradient is pointing to, so I used a step size of 0.05. In addition, since the gradient points to a local maximum, I minimize the amount of iterations to prevent it from starting to "climb".

```
function delta = optimize_gradient(m, b, y, length_virginica, width_virginica,
length_versicolor, width_versicolor)
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here

delta = [m b y];

optimized_mse = zeros(1, 200);

optimized_mse(1) = mse(delta(1), delta(2), delta(3), length_virginica,
width_virginica, length_versicolor, width_versicolor)
i = 1;


while (optimized_mse(i) >= mse(delta(1), delta(2), delta(3), length_virginica,
width_virginica, length_versicolor, width_versicolor))

i = i + 1;

gradient_out = gradient(delta(1), delta(2), delta(3), length_virginica,
width_virginica, length_versicolor, width_versicolor);

optimized_mse(i) = mse(delta(1), delta(2), delta(3), length_virginica,
width_virginica, length_versicolor, width_versicolor);

gradient_out = 0.05 .* gradient_out;

delta = delta - gradient_out;

end

figure, plot(optimized_mse);
fprintf('test');

end
```

b. I executed the function I made, and plotted the new line, and the change in MSE vs iterations. As we can see, the MSE lowers over time, and reaches a local minimum. At this point, the new decision boundary is optimal.





c. I created 3 functions that define parameters for a random decision boundary that would be easily plotted in the chart.

```
m = -1/2 + rand(1,1)*0.1
b = 5 - rand(1,1)*3
y = -1 + rand(1,1)*0.1
```

I then plotted the randomly generated line, the new line generated via the gradient, and the change in MSE over time. Note that I slightly modified my gradient optimization function to return value of the middle decision boundary value. I ran two example iterations and have them included below.

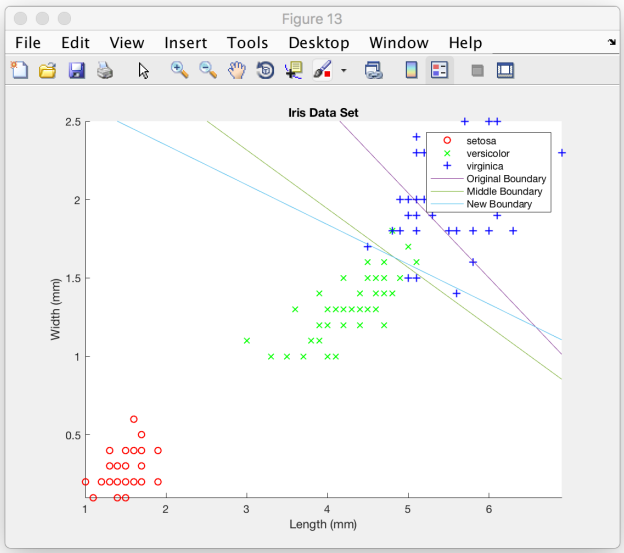


Figure 13
File Edit View Insert Tools Desktop Window Help
Iris Data Set
setosa
versicolor
virginica
Original Boundary
Middle Boundary
New Boundary
Width (mm)
Length (mm)
2.5
2
1.5
1
0.5
1
2
3
4
5
6

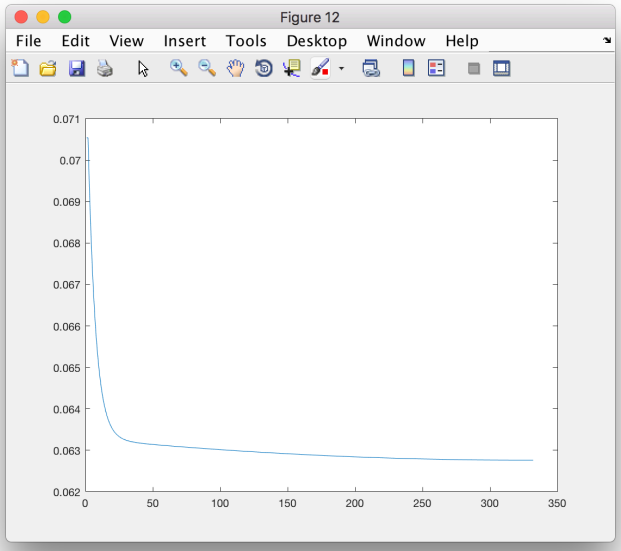
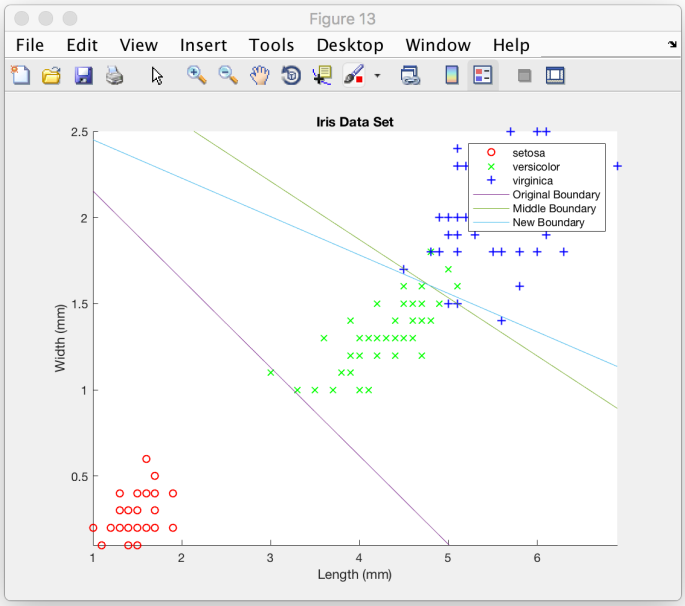
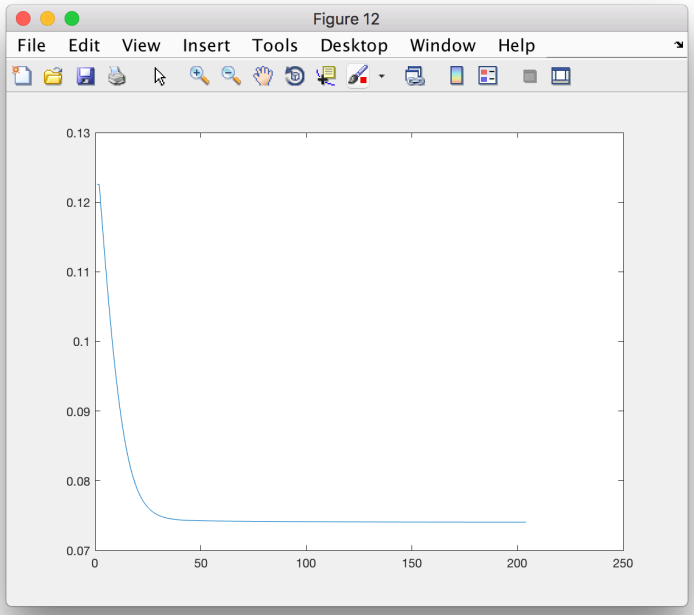


Figure 12
File Edit View Insert Tools Desktop Window Help
0.071
0.07
0.069
0.068
0.067
0.066
0.065
0.064
0.063
0.062
0
50
100
150
200
250
300
350





d. For the gradient step size, I originally chose a value of 0.01. In order to reach the local minimum in part b, I had to run 1400 iterations for it to converge. In order to speed up the process, I chose a value of 0.05. This allowed me to get close to converging in only 270 iterations. If I chose a larger value, like 0.1 the MSE converges too quickly, and we do not get a fine enough optimization.

e. My stopping criterion stems from the idea that we are trying to find a local minimum. If the new MSE calculated is larger than the previous one (it starts climbing the hill) then we know a local minimum has been reached. This is the stopping criterion.