

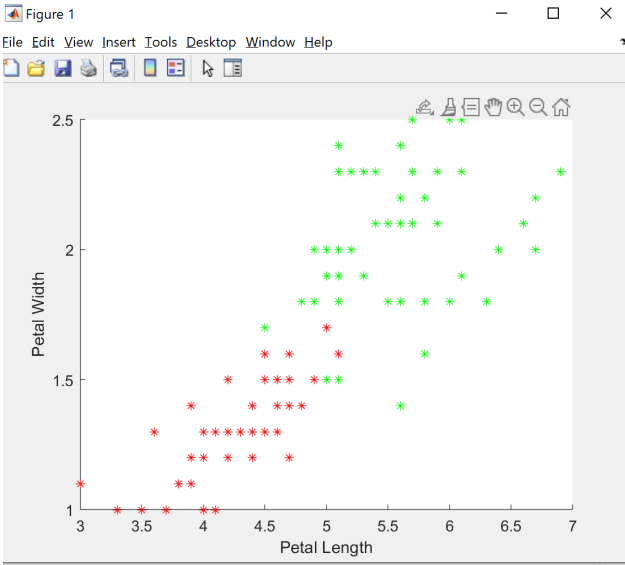
Linear Classifier Neural Network

This report outlines the implementation of a single layer neural network with one hidden layer for classifying two different flowers based on petal length and petal width. Explanations and intuitions behind every step of the process have been provided. While this implementation is for a simple neural network taking two parameters for a binary classification, similar ideas can be extended to n classes with m parameters with ease. Visualization was easy in our case to understand some basic concepts guiding the training of a neural network. The code for this can be found on my Github.

1) Data Visualization and Initialization

The first part initializes the dataset we are going to test our one-layer neural network on, defines the classifier function (Sigmoid Function), and visualizes the boundary overlaid on the sigmoid dataset.

Part a (Data Visualization)

<u>Dataset on which Neural Network will be trained</u>	
 <p>The figure is a scatter plot titled 'Figure 1' showing 'Petal Width' on the y-axis (ranging from 1 to 2.5) and 'Petal Length' on the x-axis (ranging from 3 to 7). Red asterisks represent one species, and green asterisks represent another. The red points are clustered in the lower-left region, while the green points are clustered in the upper-right region, showing a clear separation between the two groups.</p>	<p>Versicolor is indicated by red and virginica is indicated by green.</p>

Part b (Defining the classifier)

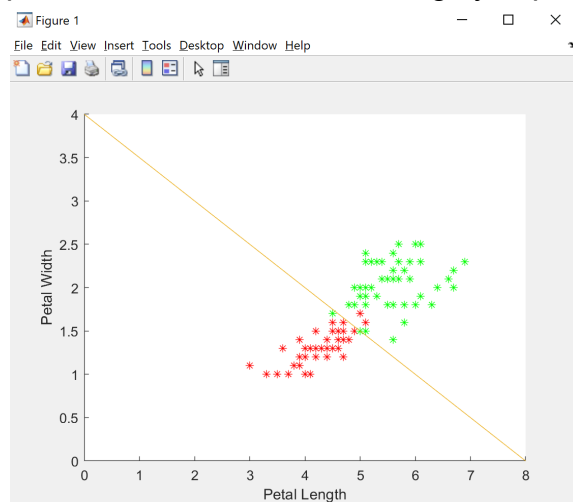
Simple function that classifies the data points as per a single layer neural network passed through a sigmoid. x_1 and x_2 represent the input point and w_1, w_2, w_0 are the parameters defining the boundary.

In abstract terms, the first layer of the neural network is doing the job of $y = x.w + w_0$ and the sigmoid acts as the hidden layer which spits out an output between 0 and 1 -- $z = \text{sigmoid}(y)$. If we just used a linear classifier, the model could only learn from a linear separable problem. Moreover, the function would not be differentiable in the entire which will cause problems during backpropagation. Adding the sigmoid in the hidden layer solves these problems by adding a non-linearity and is differentiable everywhere.

```
1 function classifier = classifier(x1,x2,w1,w2,w0)
2
3     y = (w1*x1 + w2*x2 + w0);
4
5     classifier = 1./(1+exp(-y));
6
7 end
8
```

Part c (Overlaying the decision boundary)

Weights chosen by hand : $w_1 = 0.5$; $w_2 = 1$; w_0 (bias term) = -4 and the boundary plotted on the dataset that roughly separates the 2 classes.



The function I used to get the above plot is on the next page. It's the same as part a with an additional input of the x and y intercepts for the straight line.

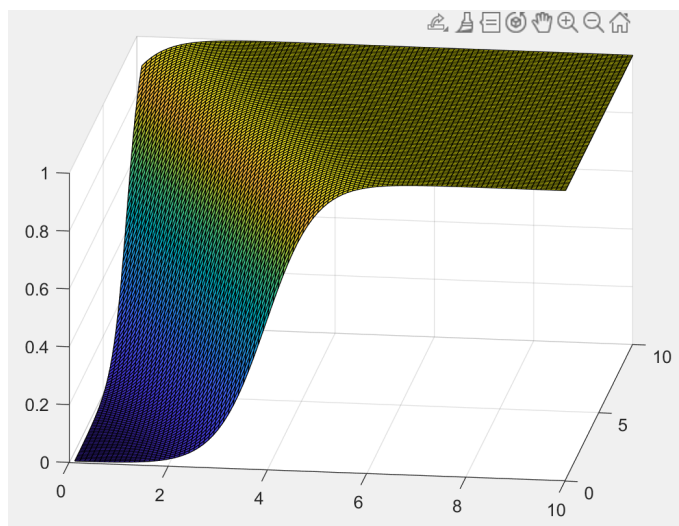
```

1- T = readtable('irisdata.csv');
2- p_length = [];
3- p_width = [];
4- species = [];
5- specie1 = {'versicolor'};
6- specie2 = {'virginica'};
7- for i = 1:150
8-     p_length = [p_length; T{i,3}];
9-     p_width = [p_width; T{i,4}];
10-    species = [species; T{i,5}];
11- end
12-
13- xlabel('Petal Length');
14- ylabel('Petal Width')
15-
16- hold on
17-
18- x1 = [];
19- for i = 1:150
20-     if isequal(specie1{1}, species{i})
21-         plot(p_length(i), p_width(i), '*r' )
22-
23-     end
24-     if isequal(specie2{1}, species{i})
25-         plot(p_length(i), p_width(i), '*g' )
26-     end
27- end
28- plot([4 0], [0 8])
29- hold off
30-

```

Part d (Visualizing the sigmoid on our dataset)

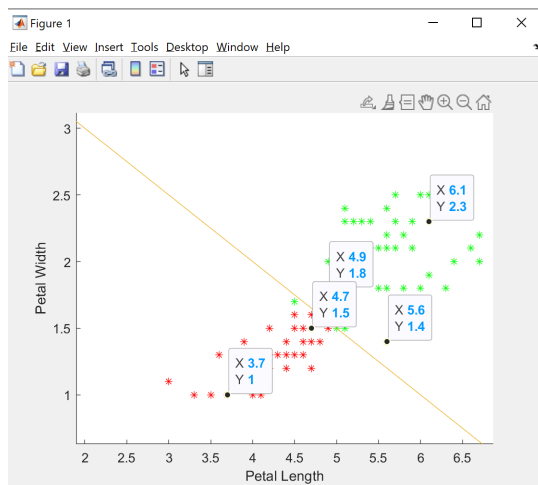
The graph below visualizes the output from the sigmoid on our training datasets with the weights as initialized in part c. The line derived by intersecting the plane $z=5$ with the curve projected onto the xy plane gives the resulting decision boundary for our dataset in 2-D.



The output is plotted over the domain $x = [0,10]$ and $y = [0,10]$

Part e(Outputs from the sigmoid at specific points)

The 5 points chosen are shown below along with the outputs from the classifier. For purposes of classification, an output of >0.5 should be considered class 3 and ≤ 0.5 should be considered class 2. As expected, for the points lying below the boundary, we get an output of ≤ 0.5 and >0.5 for the ones lying above the boundary.



Outputs

```
>> classifier(6.1,2.3)

ans =

    0.9370

>> classifier(4.9,1.8)

ans =

    0.6225

>> classifier(5.6,1.4)

ans =

    0.5987

>> classifier(4.7,1.5)

ans =

    0.4256

>> classifier(3.7,1)

ans =

    0.0911
```

Defining Loss functions and gradient descent

Part a (Loss Function - Mean squared error)

We use MSE as our loss function.

u_1 and u_2 are the input data vectors. w_1, w_2 , and w_0 are the parameters defining the boundary of the neural network that are to be optimized. The mean squared error is computed by using the ground truth and the output from the classifier for the respective points.

```
1 function mean_squared = mean_squared(u1,u2,w1,w2,w0,class1,class2)
2 T=readtable('irisdata.csv');
3 count = 0;
4 %Pass in the inputs through the classifier function to get the required
5 %output from the sigmoid function.
6 c = classifier(u1,u2,w1,w2,w0);
7 k = zeros(150,1);
8 %In iris data,versicolor and virginica starts from index 51
9 %Since we are not concerned with setosa, we will ignore those indices.
10 %The for-loop below classifies versicolor as 0 and virginica as 1
11 for i = 51:150
12     if isequal(T{i,5}{1},class1)
13         k(i)=0;
14     end
15     if isequal(T{i,5}{1},class2)
16         k(i)=1;
17     end
18 end
19
20 %Compute the mean squared error
21 for i = 51:150
22     count = count + (c(i)-k(i)).^2;
23 end
24 mean_squared = count/100;
25
26 end
```

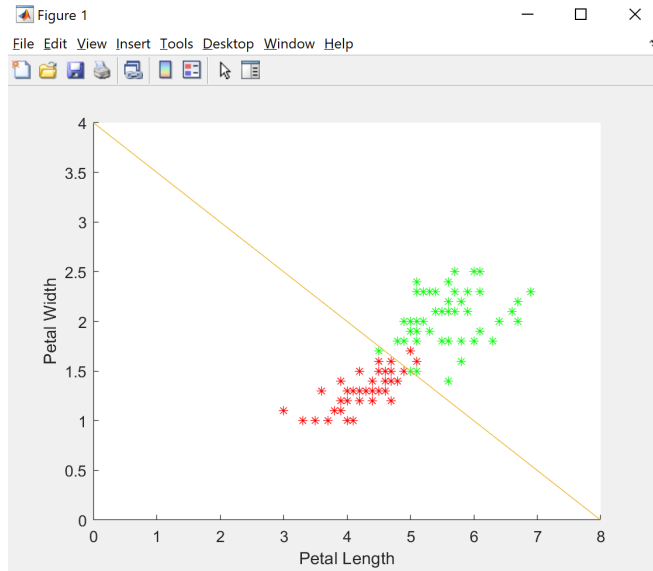
Part b (MSE for specific cases)

The settings I used to get a mean squared error of 0.128 were $w_1 = 0.5$; $w_2 = 1$; w_0 (bias term) = -4. This was the minimum mean squared error I could get by manually adjusting the weights by hand. Optimization will probably lead to better results.

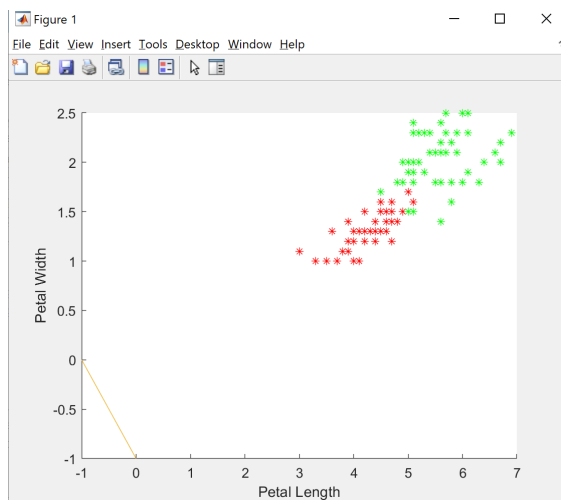
```
>> z=mean_squared(a,b,0.5,1,-4,'versicolor','virginica')
```

```
z =
```

```
0.1280
```



The settings I used to get a mean squared of 0.5 were $w_1 = 10$; $w_2 = 10$; $w_0(\text{bias term}) = 10$. In fact, for our particular set of data with 50% 1's and 50% 0's and the sigmoid giving a result between 0 and 1. For the correctly classified points, the squared error would be close to 0 whereas for the incorrect classifications, the difference would be close to 1. This should be expected given the distance of the decision boundary to the points. This would result in an approximate MSE of 0.5



```
>> z=mean_squared(a,b,10,10,10,'versicolor','virginica')

z =

    0.5000
```

Part c and d (Derivation of gradient)

Simple derivations for the gradients using chain rule and the fact $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. These gradients will be used in the gradient descent process to optimize the weights of the boundary.

Attached at the very end of the report.

Part e (Computing gradient)

Code that computes the gradient for the neural network based on the derivation above.

```
1 function gradient = gradient(w0,w1,w2,u1,u2)
2     T=readtable('irisdata.csv');
3     y = classifier(u1,u2,w1,w2,w0);
4     g_loss = zeros(3,1);
5
6     class1= 'versicolor';
7     class2= 'virginica';
8
9     for i = 51:150
10         if isequal(T{i,5}{1},class1)
11             k(i)=0;
12         end
13         if isequal(T{i,5}{1},class2)
14             k(i)=1;
15         end
16     end
17
18
19     for i = 51:150
20         g_loss(1) = g_loss(1) + (-2*(k(i)-y(i))*y(i)*(1-y(i)));
21         g_loss(2) = g_loss(2) + (-2*(k(i)-y(i))*y(i)*(1-y(i))*u1(i);
22         g_loss(3) = g_loss(3) + (-2*(k(i)-y(i))*y(i)*(1-y(i))*u2(i);
23     end
24     gradient = g_loss/100;
25 end
```

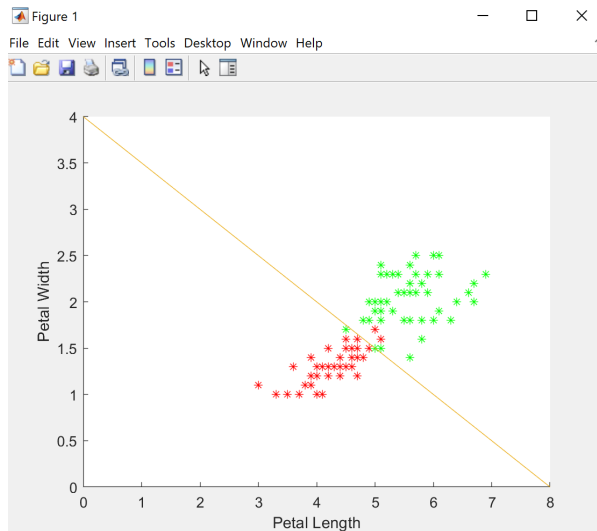
```
>> gradient(-4,0.5,1,T{:,3},T{:,4})
```

```
ans =
```

```
    0.0172
    0.0095
   -0.0137
```

Before applying the gradient

$w_0 = -4$; $w_1 = 0.5$; $w_2 = 1$



After applying the gradient(w_0, w_1, w_2 for $\text{eps}=0.5$ and $\text{iterations}=1$)

```
>> gradient_d(-4, 0.5, 1, T{:, 3}, T{:, 4}, 1)
```

```
ans =
```

```
-4.0086    0.4953    1.0069
```

Since the question asks for the boundary change for 1 iteration, the decision boundary(essentially w_0, w_1, w_2) gets updated by a small amount. **NOTE** : The **first part of question 3** will illustrate the optimization better when the gradient descent algorithm is run for more iterations.

There is a small reduction in the mean squared error for this small change.

Applying Gradient descent, interpreting results and Hyper-parameters

Part a (Defining gradient descent and applying it)

In this part we apply gradient descent for 200 iterations, after which the reduction in MSE stops changing. The MSE at this point is 0.0975

```
1 function gradient_d = gradient_d(w0,w1,w2,u1,u2,iterations)
2     eps = 0.5;
3     g = zeros(3,1);
4     for i = 1:iterations
5         g = gradient(w0,w1,w2,u1,u2);
6         w0 = w0 - eps*g(1);
7         w1 = w1 - eps*g(2);
8         w2 = w2 - eps*g(3);
9     end
10    gradient_d = [w0 w1 w2];
11 end
```

```
>> gradient_d(-4,0.5,1,T{:},3},T{:},4},200)
```

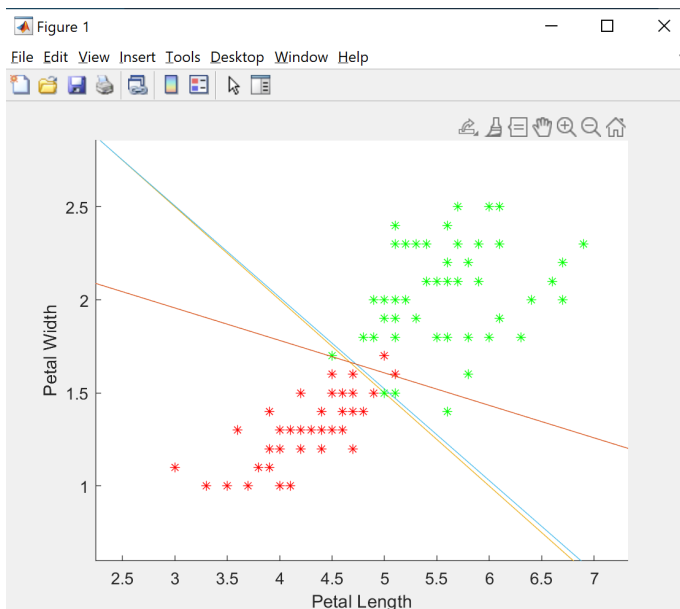
```
ans =
```

```
-5.2960    0.3725    2.1352
```

Weights before optimization : -4, 0.5,1

Weights after optimization : -5.296, 0.3725, 2.1352

The old decision boundary is the one in orange and the new one is the one in red (after 200 iterations of gradient descent). The one in blue shows the boundary after 1 iteration of gradient descent.



As expected, there is also a significant relative difference in the mean squared error after 200 iterations of gradient descent

```
>> mean_squared(T(:,3),T(:,4),0.5,1,-4,'versicolor','virginica')

ans =

    0.1280

>> mean_squared(T(:,3),T(:,4),0.3725,2.1352,-5.296,'versicolor','virginica')

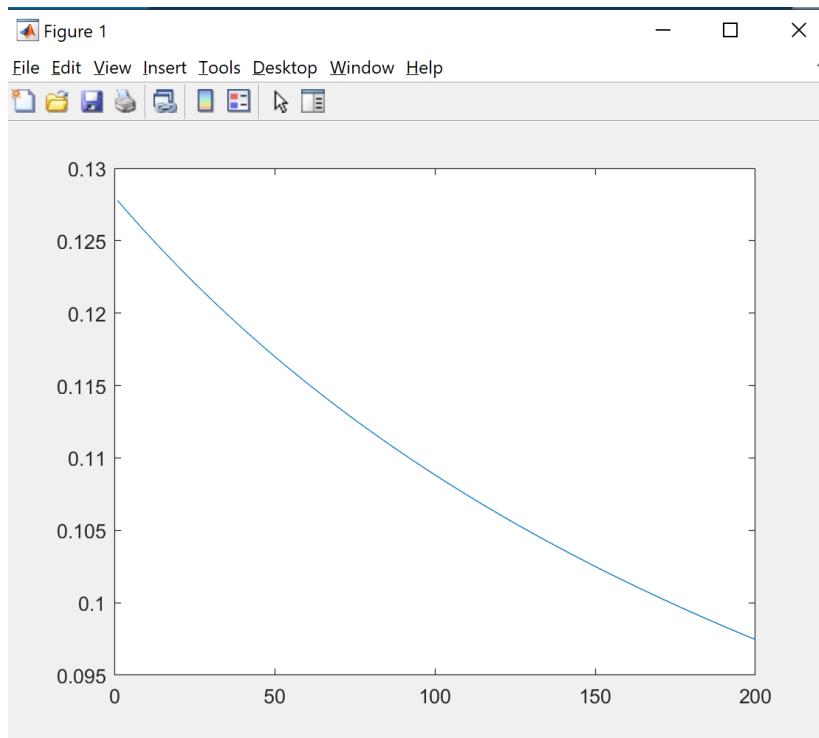
ans =

    0.0975
```

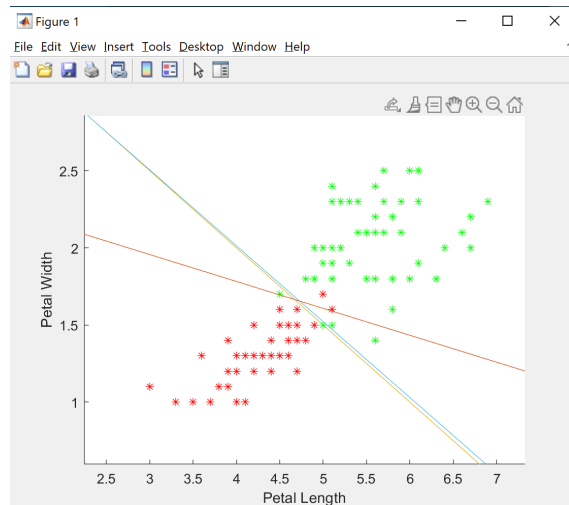
Part b (Trends in MSE)

Mean squared error (Objective function) vs # of iterations

As expected, the mean squared error goes down as we increase the number of iterations (Up to 200 iterations of gradient descent).



Red Boundary - Boundary after optimization.
Orange Boundary - Boundary before optimization



Part c (Trends in MSE)

Randomly generated weights

Note : I did use some restrictions on w_1, w_2 and w_0 while generating them to make sure we have a decent decision boundary to begin with. The reason for this is pretty simple - if our decision boundary produced misclassifications to a great degree then the gradient at that point would be really small since we will be at a relative maxima, meaning that gradient descent won't be able to get us out of here. Thus it's essential that our starting weights produce decent results to begin with (i.e. they're not at a relative maxima). We could use something like simulated annealing to avoid this issue too in the initial stages.

```
>> w1 = 0 + (5) .* rand(1,1)

w1 =

    1.3846

>> w2 = 0 + (5) .* rand(1,1)

w2 =

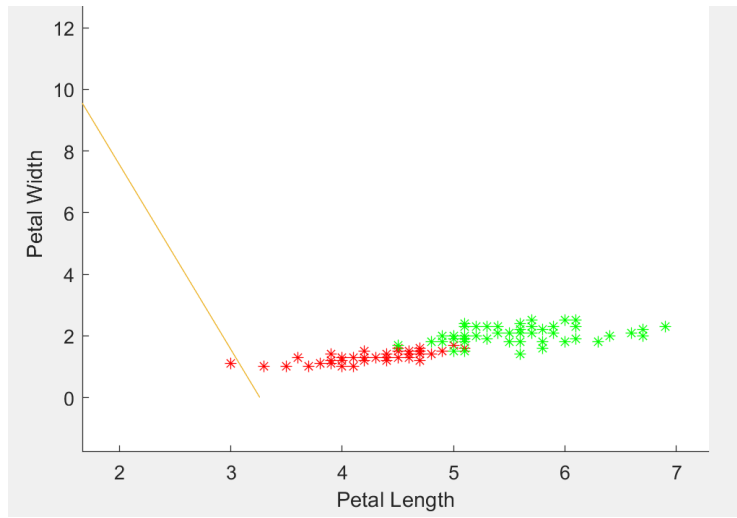
    0.2309

>> w0 = -5 + (5) .* rand(1,1)

w0 =

   -4.5143
```

Before Optimization($w_1x_1+w_2x_2+w_0=0$):



Mid-way through gradient descent: (After 50 iterations in this specific case)

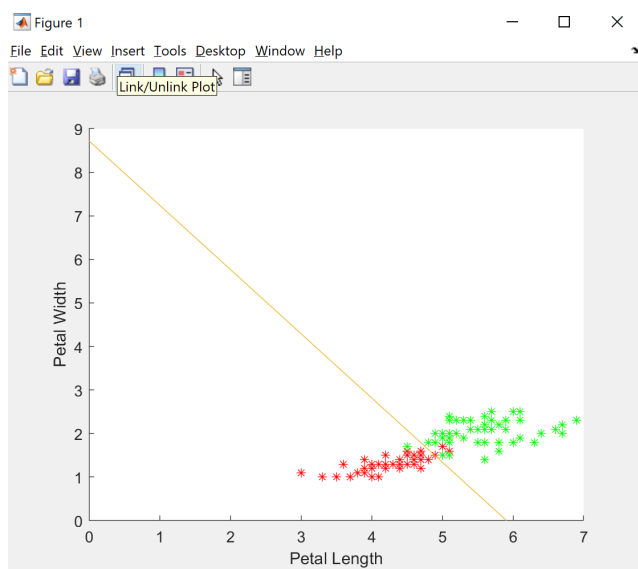
The second row indicates the weights midway through gradient descent

```
>> gradient_d(w0,w1,w2,T{:,3},T{:,4},100)
```

```
ans =
```

```
-5.3014    0.7682    0.9754  
-4.9764    0.8419    0.5706
```

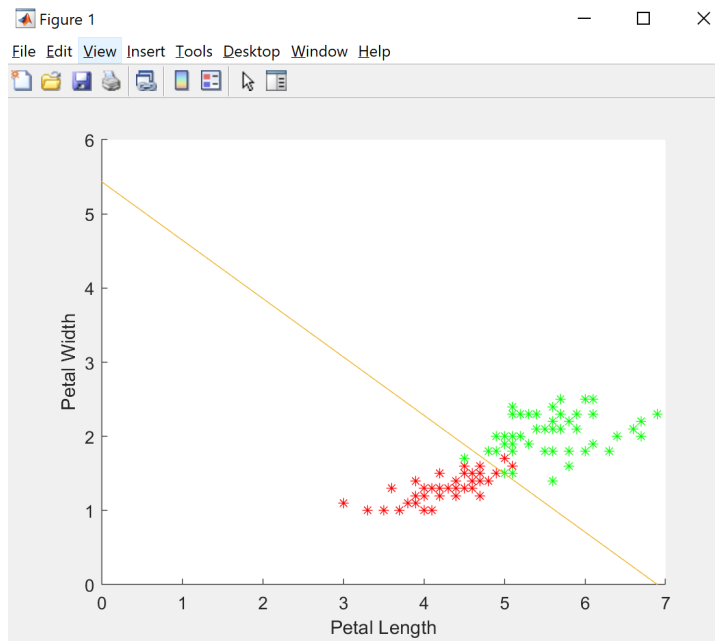
Using the second row as weights for w_0, w_1, w_2



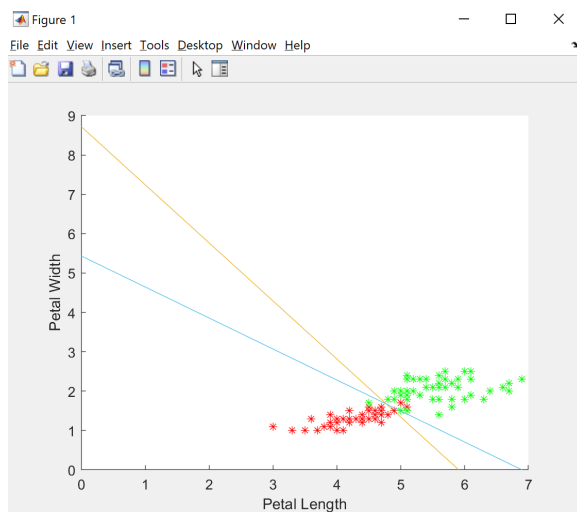
After gradient descent (After 100 iterations in this case)

The first row indicates the weights after full gradient descent

Boundary using $w_0=-5.3014$ $w_1=0.7682$ $w_2=0.9754$

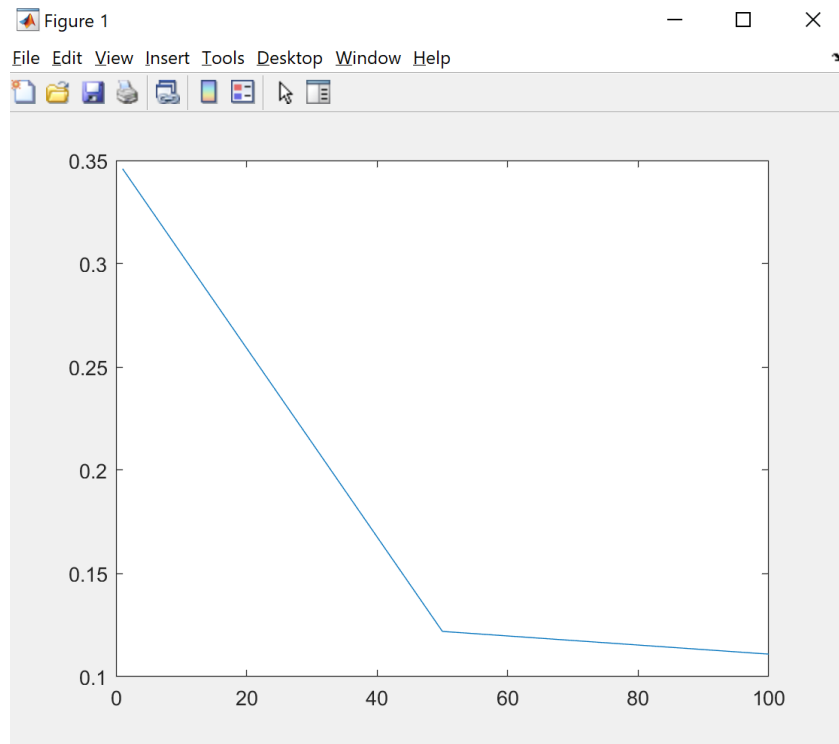


Combined boundaries after 50 and 100 iterations. The one in blue is the final decision boundary whereas the one in orange is the decision boundary after 50 iterations.



Mean squared errors before gradient descent, after gradient descent and midway through gradient descent.

Plot of the objective function(loss function) at 0,50 and 100 iterations. Loss is on the y-axis



Part d (Tuning Hyper-parameters)

There were mainly two factors that were taken into account while setting the step size:

- 1) If the step size was too small, the number of iterations needed to reach the relative minima would be high and this would need more computation and time meaning learning is slow in this case.
- 2) If the step size is too high, then we might take a large step and skip over the minima and keep oscillating on either side of the minima as we are using gradient descent.

Keeping these two factors and running a couple of experiments, I was able to choose a step size of 0.5 which turned out to be optimal, i.e, not taking too long to learn and converging to an optimal decision boundary within a given number of iterations as long as we start with decent weights. Whenever our initial guesses for weights were decent, this step size managed to converge to an optima in about 200 iterations.

Part e (Tuning Hyper-parameters)

- 1) Stopping criterion could have been chosen such that the difference in gradient between two successive iterations was really small, OR
- 2) Have a maximum number of iterations over which gradient descent will be performed.

I went with the second option in this case. Experimentally, I found out that my decision boundary converges to an optimal one in around 200 iterations if we start with a decent initial guess for the weights. My input function for gradient descent, in fact, can take in a variable which sets the number of iterations. So, depending on whether the initial boundary was good to begin with or not, we might want to adjust the number of iterations. However, in most cases, 200 iterations should be sufficient to converge to an optimal boundary after which, the rate of decrease of the loss function becomes smaller and smaller as illustrated in part b of this problem.

Another reason why I did not choose the first criterion is that the loop may terminate if we begin with a really bad guess for the weight(i.e a relative maxima), in which case the gradient will be small and so will the difference in gradients for two successive iterations thereby terminating the loop in the first iteration itself and we don't want this to happen.