# CSDS 391 P2 Writeup

Name: Rohan Singh
Case id: rxs1182@case.edu

## Code Design:

In this section I will briefly explain the way in which I designed my code to make it more readable, easy to debug and functional. I achieved this using the following:

1) **Helper functions:** I tried to make my code as loosely coupled as possible, so that it could be easier to debug it and find errors. I achieved this primarily by using helper functions for getting information like:
    - Objective function values
    - Learning functions
    - Other functions like: calculating euclidean distance, sigmoid functions etc.
2) **Processing of data:** Since the data that I received from from reading the csv files to the output of the code were in different forms so I did the following to avoid any type mismatch problems:
    - Converting the species names to binary codes.
    - Converting panda dataframes to numpy arrays, so that I could have consistent function argument types.
3) **Plotting helper functions:** Instead of plotting the output in a long function, I delegated this task to plotting helper functions for the different types of plots. This made my code easier to debug. Most of my plots use plotly and some use matplotlib, depending on what suited the task better.
4) **Making separate modules for different code:** As mentioned in the README, I divided my code into separate Python Modules based on the exercise number. It was also easier for me to track my progress that way.
5) **Running Code in Jupyter:** I wrote down most of my code in Jupyter Notebooks, as it is much easier to see the value of variables there, and then I would rewrite my functions into Python Modules.
6) **Main Method Demonstrations:** I wrote down all of the demonstrations inside the main methods of those python modules.

## Exercise 1. Clustering

(a) The Objective function that I used for the 4-dimensional KMeans was:

$$Val = \Sigma_n \Sigma_k r_{n,k} ||x_n - \mu_k||^2$$

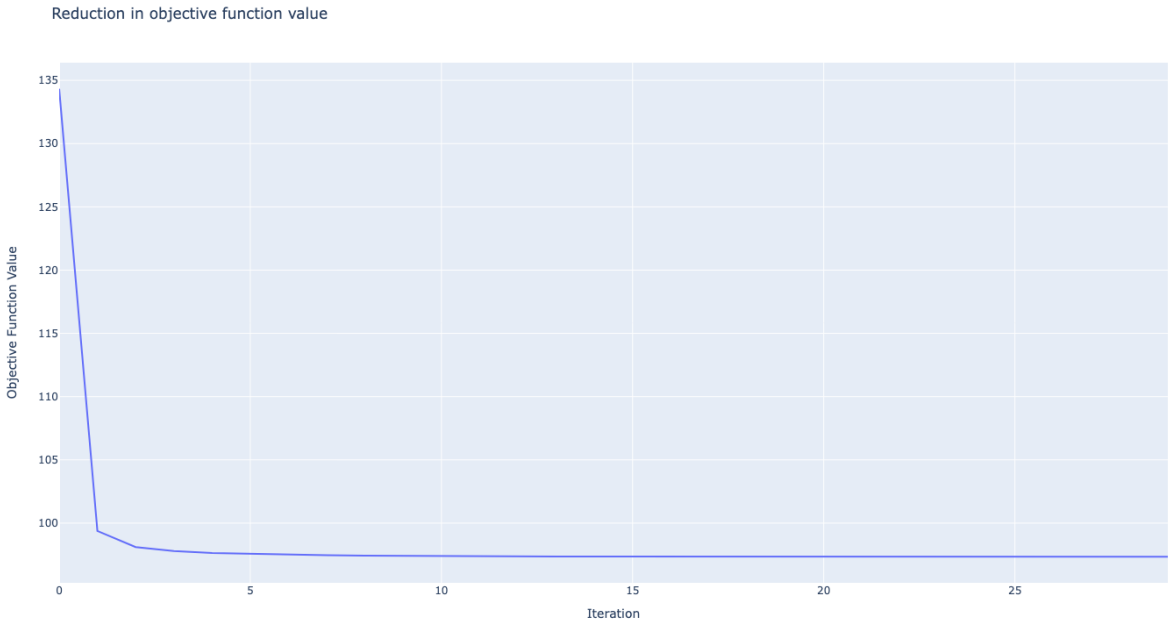
For getting the euclidean distance I used the following function

```
#Helper function to get the euclidean distance
def euc_dist(self,point, centroid):
    dist_sum = 0
    dist_sum += np.power((point[0]-centroid[0]),2)
    dist_sum += np.power((point[1]-centroid[1]),2)
    dist_sum += np.power((point[2]-centroid[2]),2)
    dist_sum += np.power((point[3]-centroid[3]),2)
    return np.sqrt(dist_sum)
```

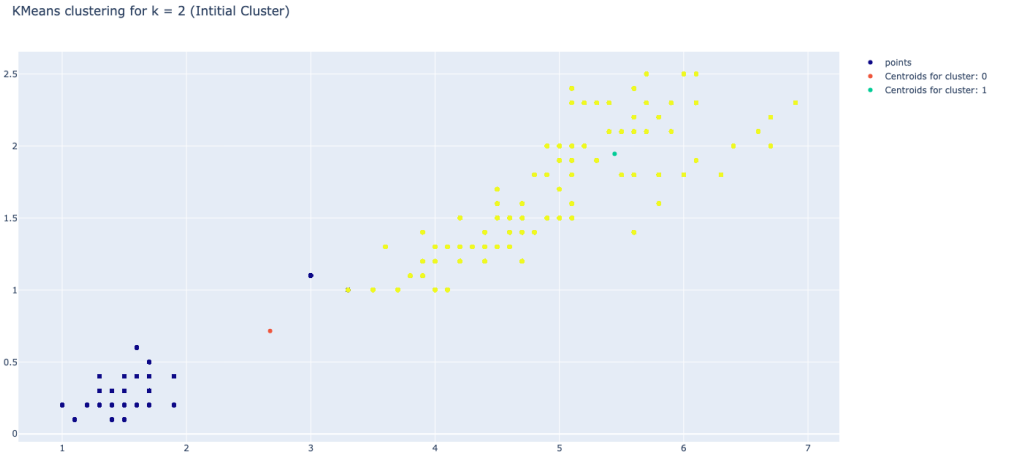
The Learning rule was to take the mean of each cluster and make that the coordinates of the new centroid.

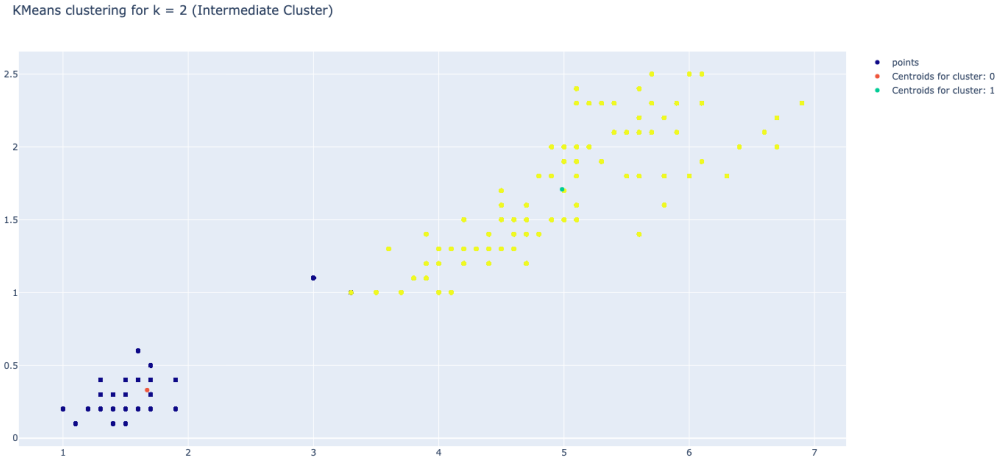
```
#Helper function for the update rule for KMeans
def updateCentroids(self,centroids, classes):
    #updating the coordinates of the centroids for each class
    new_centroids = []

    #This is done by taking the mean of each cluster
    for i in range(0,self.k,1):
        tempclass = classes[i]
        s_len_sum = 0
        s_wid_sum = 0
        p_len_sum = 0
        p_wid_sum = 0
        for j in range(0,len(tempclass),1):
            point = tempclass[j]
            s_len_sum += point[0]
            s_wid_sum += point[1]
            p_len_sum += point[2]
            p_wid_sum += point[3]
        cl = []
        cl.append(s_len_sum/len(tempclass))
        cl.append(s_wid_sum/len(tempclass))
        cl.append(p_len_sum/len(tempclass))
        cl.append(p_wid_sum/len(tempclass))
        new_centroids.append(cl)

    return new_centroids
```

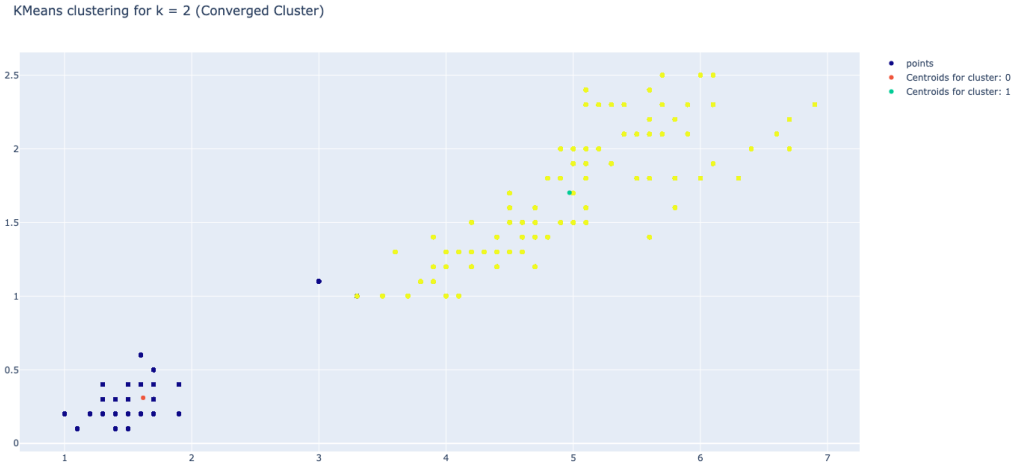
(b) This was the result that I got from the learning rule by plotting the objective function over multiple iterations:


Reduction in objective function value
Objective Function Value
135
130
125
120
115
110
105
100
0
5
10
15
20
25
Iteration

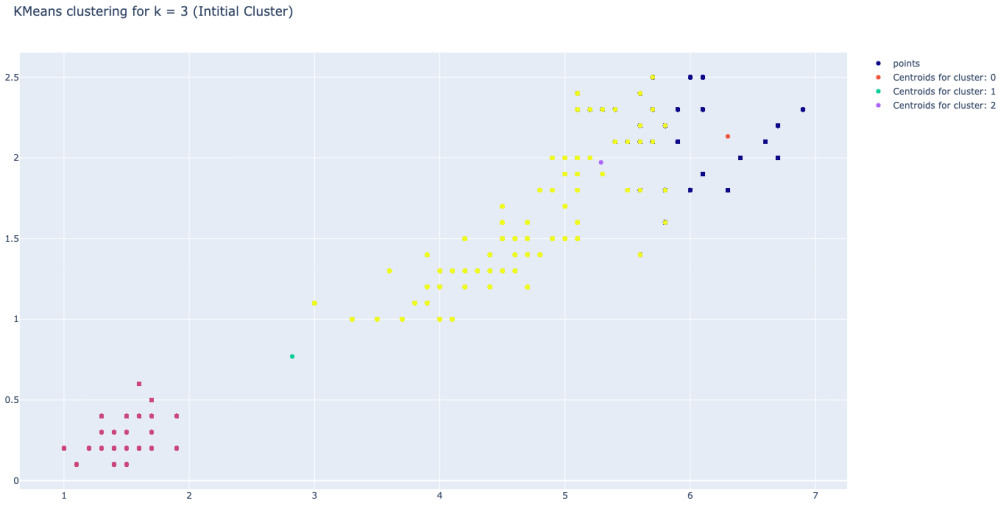
(c) Result of KMeans for K=2:

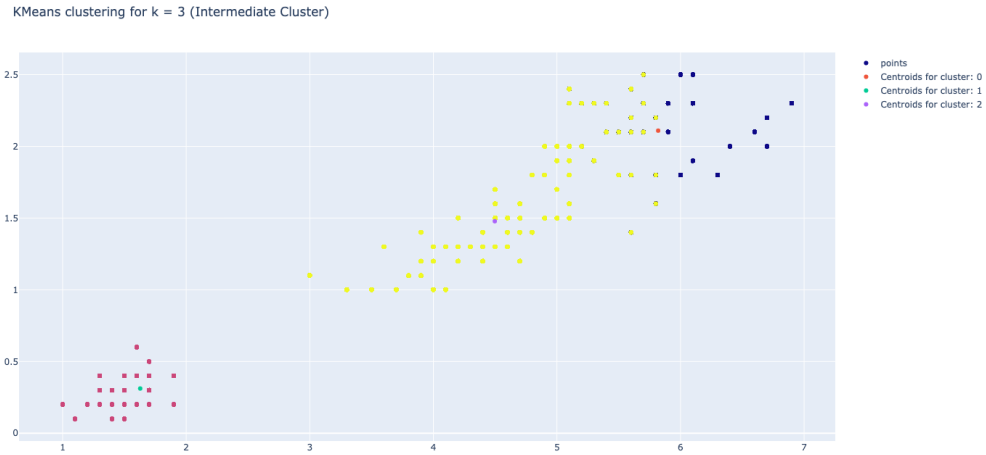


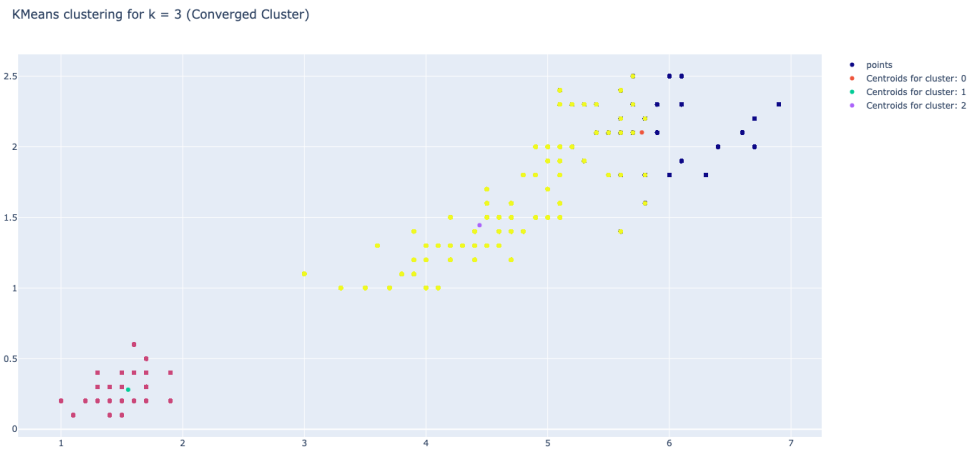



Results for K = 3:



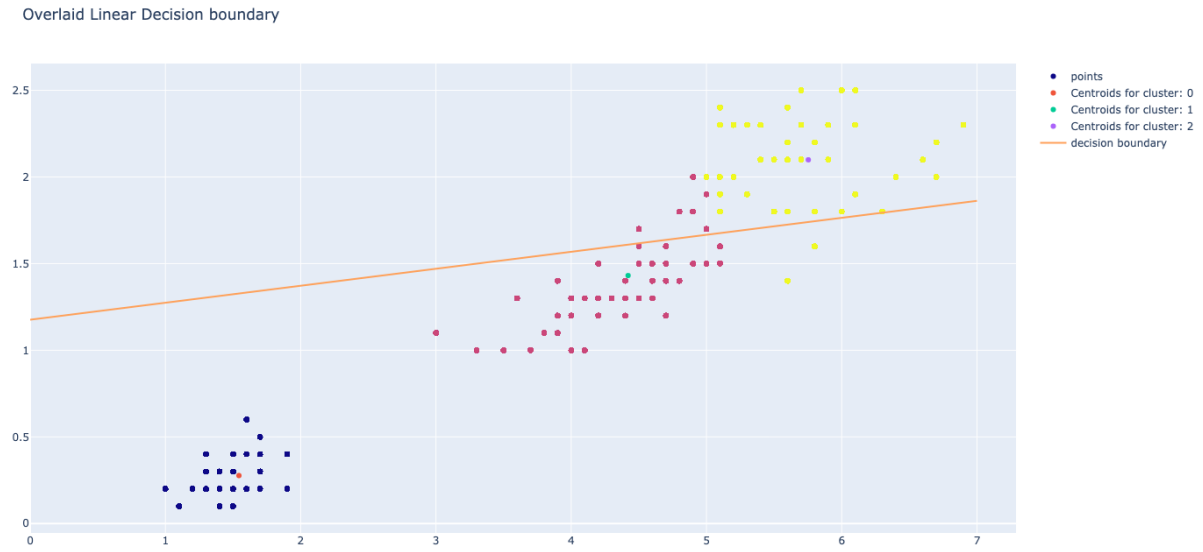



(d) Plotting the decision boundary on the clusters using optimized parameters:
My approach to this was to obtain the optimized linear decision boundary weights and bias terms and then overlaying that onto my KMeans clustering plot. This showed me the error in the plot. Here is the function and plot:

```
#Function to plot the decision boundaries
def plot_decision_boundaries(self,b,w):
    #Getting the inctercept and the slope
    c = -(b/w[1])
    m = -(w[0]/w[1])

    #Making the arrays for the x and y axes
    xd = np.array([0,7])
    yd = m*xd + c

    #Getting the predicted values
    init_centroids, intermediate_centroids, converged_centroids, df_initial, df_intermediate, df_converged = self.predict()

    #Adding it all to a plotly grpah object
    fig_1 = go.Figure()
    fig_1.add_trace(go.Scatter(x=df_converged["Petal Length"], y=df_converged["Petal Width"],
        mode='markers',
        name='points',
        marker = {'color':df_converged["Class"]}
    ))
    for i in range(0,len(init_centroids),1):
        _name = "Centroids for cluster: " + str(i)
        fig_1.add_trace(go.Scatter(x=np.array(converged_centroids[i][0]),y=np.array(converged_centroids[i][1]),
            mode="markers",
            name=_name
        ))
    fig_1.add_trace(go.Scatter(x=xd, y=yd,
        mode='lines',
        name='decision boundary'
    ))
    fig_1.update_layout(title="Overlaid Linear Decision boundary")
    fig_1.show()
```

Plot:



## Exercise 2. Linear Decision Boundaries:

(a) This was the plot of the IRIS classes and the overlaid decision boundary:


(b) To calculate the function value, I used these helper functions:

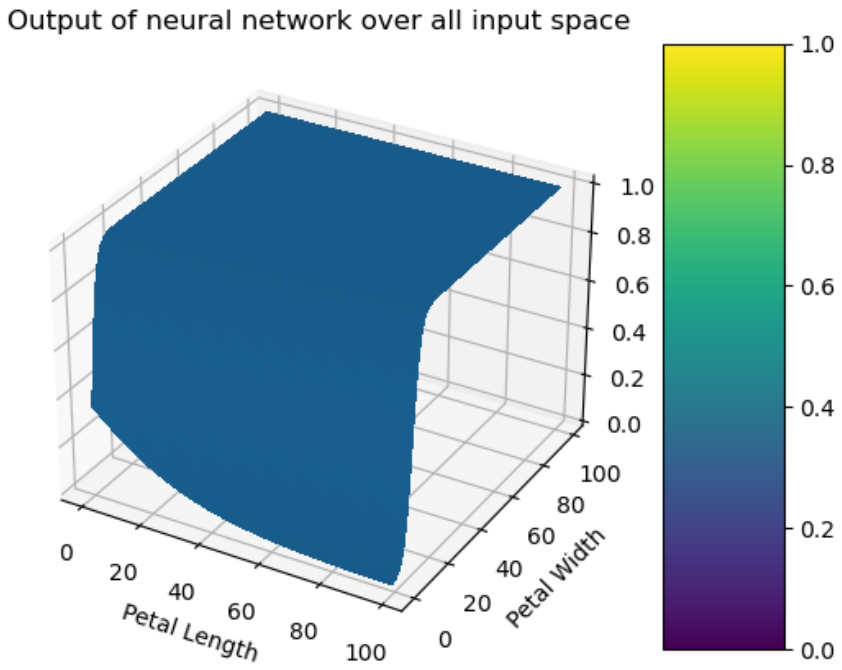
```python
#Function that calculates the value of the sigmoid function
def sigmoid(z):
    return 1.0/(1 + np.exp(-z))

Run Cell | Run Above | Debug Cell
#%%
#Function that predicts the value of the class using a single-layer neural network
def predict(X,weights,bias):
    # X --> Input vector

    # Calculating the value of y using the inner-product (dot product) of the weights vecotr with the data vector and adding a bias t
    y = np.dot(X, weights) + bias
    preds = sigmoid(y)

    # Empty List to store predictions.
    pred_class = []
    for i in range(0,len(preds),1):
        # if preds >= 0.5 --> round up to 1
        # if preds < 0.5 --> round up to 1
        if(preds[i] > 0.5):
            pred_class.append(1)
        else:
            pred_class.append(0)

    #pred_class = [1 if i > 0.5 else 0 for i in preds]

    return np.array(pred_class)
```

(c) The weight parameters that I set by hand were:

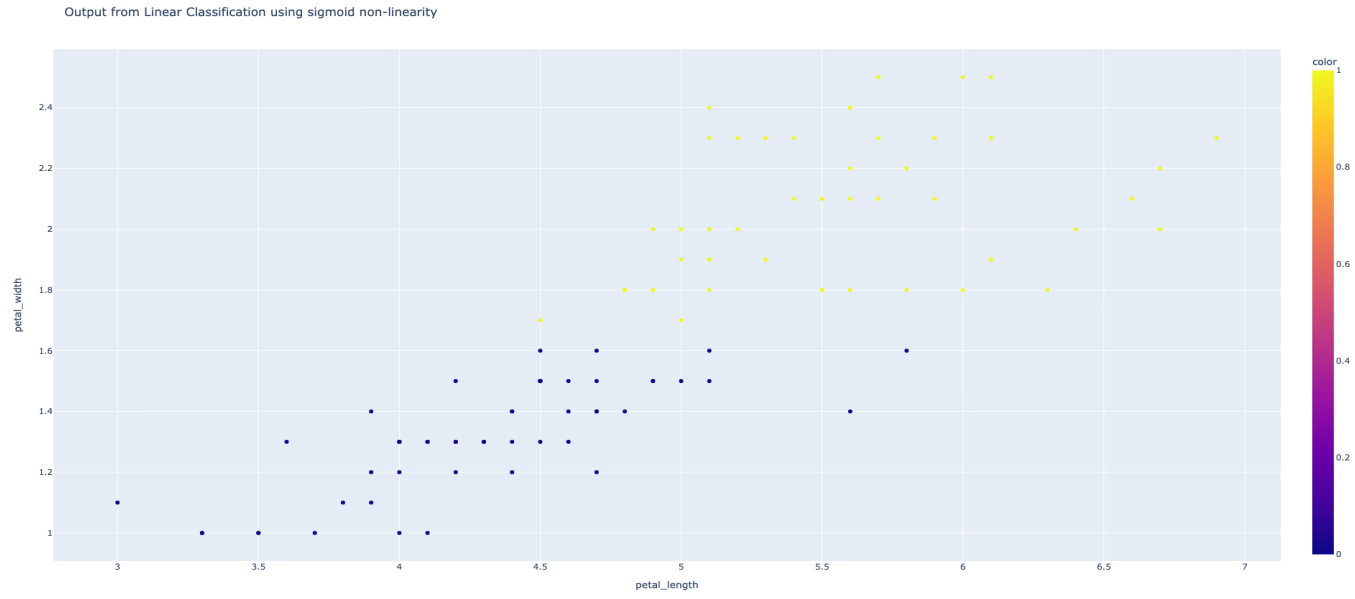
W = [-0.05,0.51]

B = -0.6

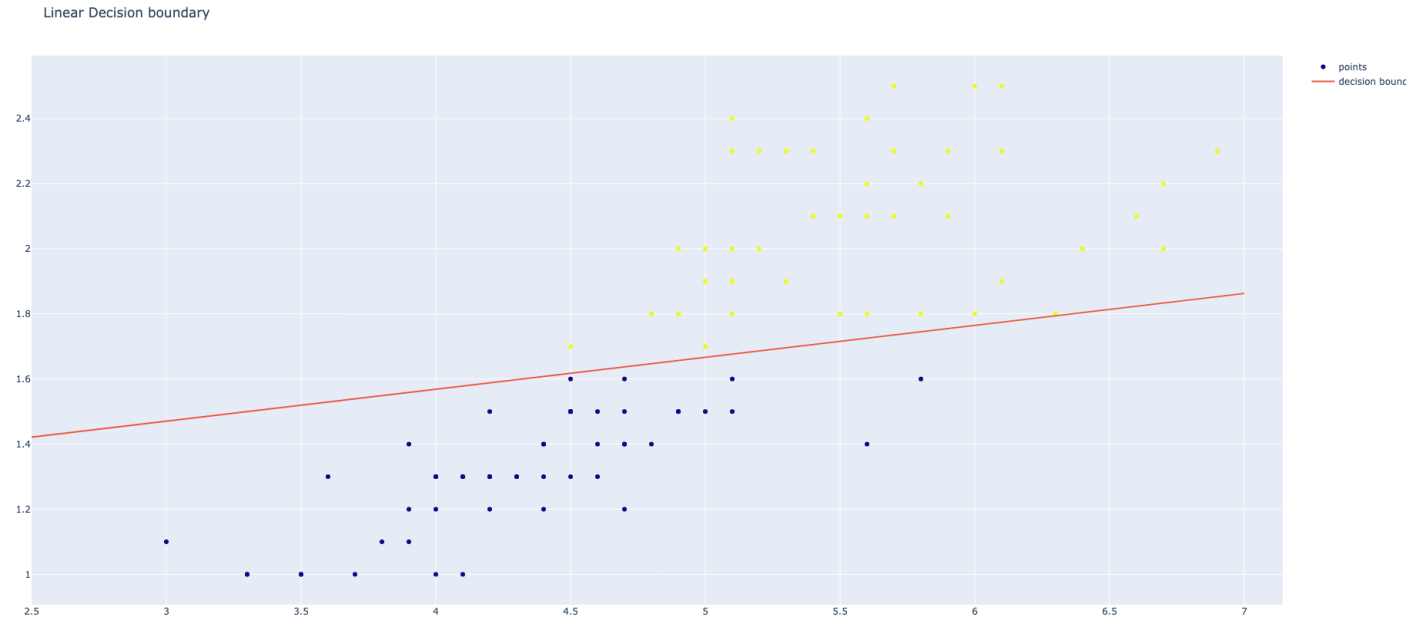
(d) Here is the surface plot over all input space for the neural network:




(e) Here is the classification for the 2nd and 3rd Iris Classes using my simple classifier (Yellow might be slightly hard to see):



Here it is with a decision boundary line (again sorry for yellow):




```
228
229     #Part (e) getting the results for examples from the iris class
230     test_set = np.array([[3.5,1.0],[3.9,1.1],[3.8,1.1],[4.7,1.5],[6.0,2.5],[6.9,2.3],[5.1,1.8],[6.1,1.9]])
231     test_results = predict(test_set,w,b)
232     print("\n\noutput of the prediction:")
233     print(test_results)
234     print("\n\n")
235     #Expected Output: [0,0,0,0,1,1,1,1]
236
237     #Calling the function that will plot the results of the neural network over all in
238     plot_over_space(w, b)
239
240
241
242
243
244
245
246
247
```

```
PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE   JUPYTER

(base) rohansingh@Rohans-MacBook-Pro RohanSingh_P2 % /Users/rohansingh/opt/anaconda3/bin/python /Users/rohansingh/Desktop/RohanSingh_P2/EX_2.p
y


output of the prediction:
[0 0 0 0 1 1 1 1]
```

^These are the results for a smaller subset of the examples from the 2nd and 3rd iris classes

## Exercise 3. Neural Networks:

(a) Formula for total-squared error: To get mean squared error you just to

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

Code for the TSE helper method:

```
#Function to calculate the mean squareed error
def calculate_MSE(data,weights,bias,target):

    #Predicting using the neural network predict function
    y = predict(data,weights,bias)
    mse = 0

    #Iterating through all of the data points
    for i in range(0,len(y),1):
        #Adding the sum of the differences
        mse += ((y[i] - target[i])**2)

    #Dividing the sum by 2
    mse /= 2

    return mse
```

(b) Mean Squared error and plots:

Total Squared Error:- (MSE is divided by 100)

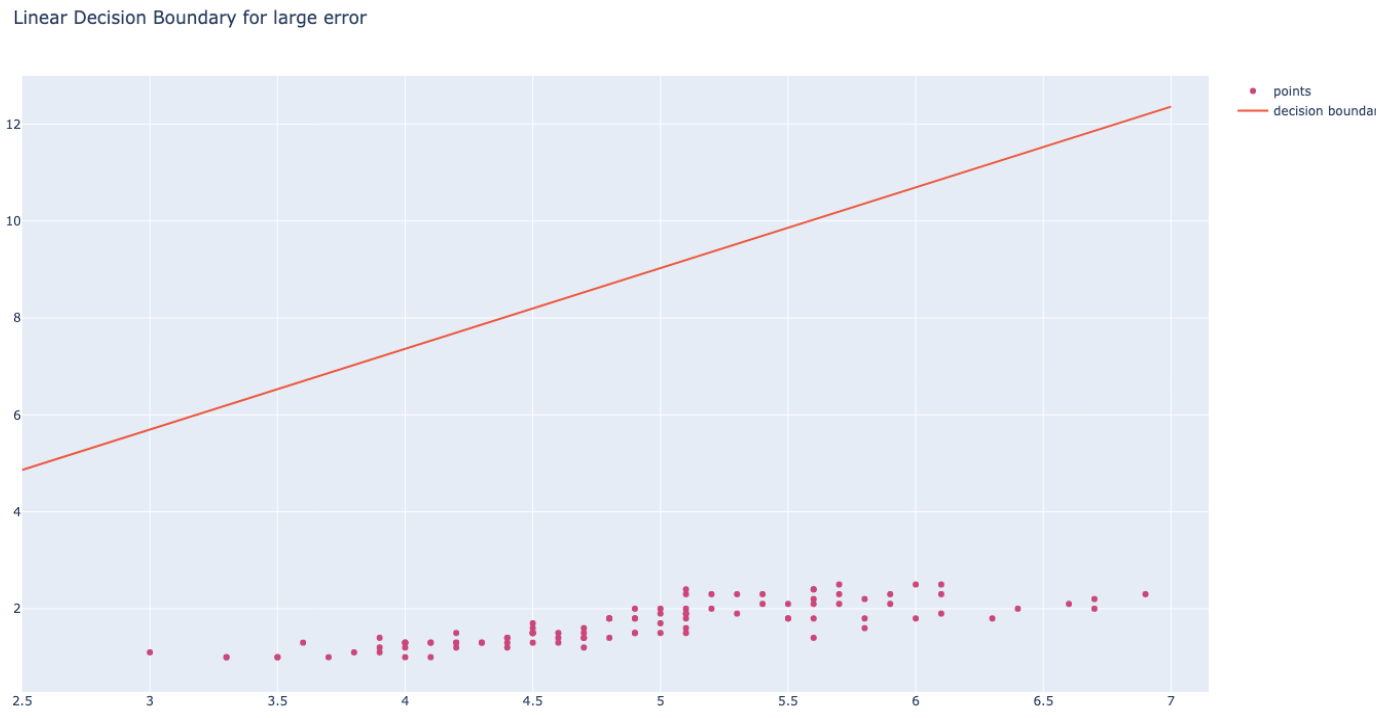
```
(base) rohansingh@Rohans-MacBook-Pro Machine Learning %
de/Machine Learning/P2/Mean_Squared_Error.py"
Optimal MSE:  3.0
Random MSE:  25.0
Close MSE:  9.5
```

Optimal MSE: 0.03

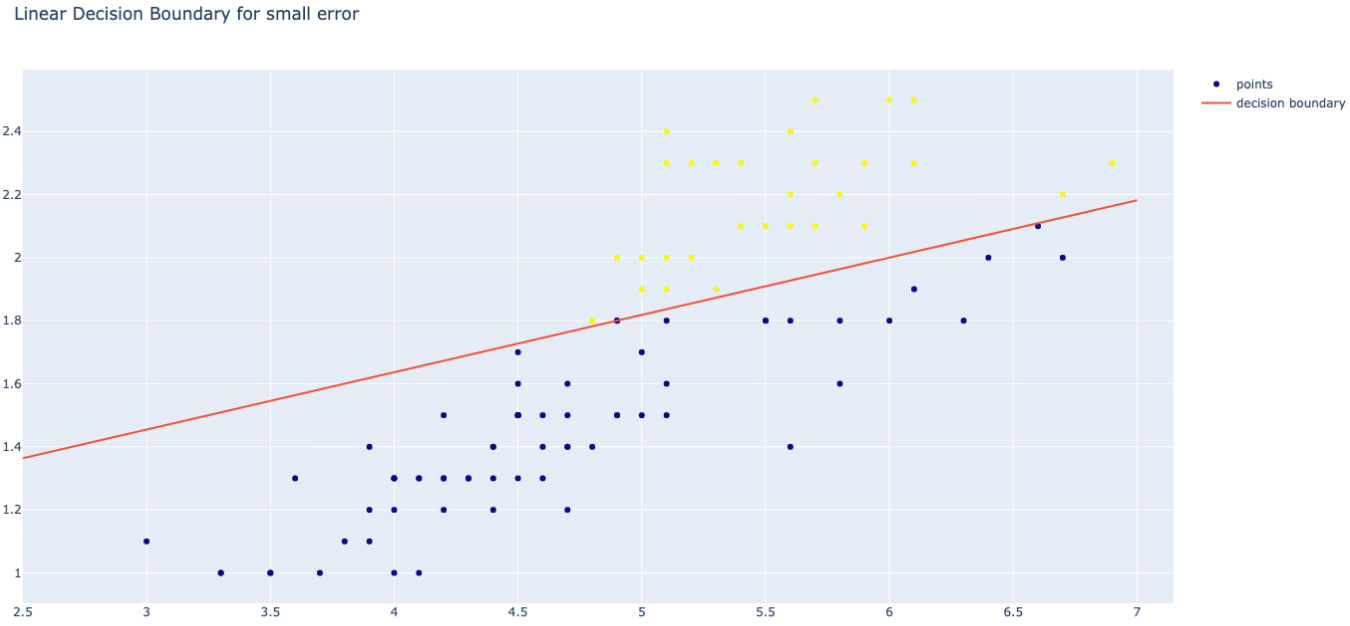
Random MSE: 0.25

Close MSE: 0.095

Plot for Large Error:-

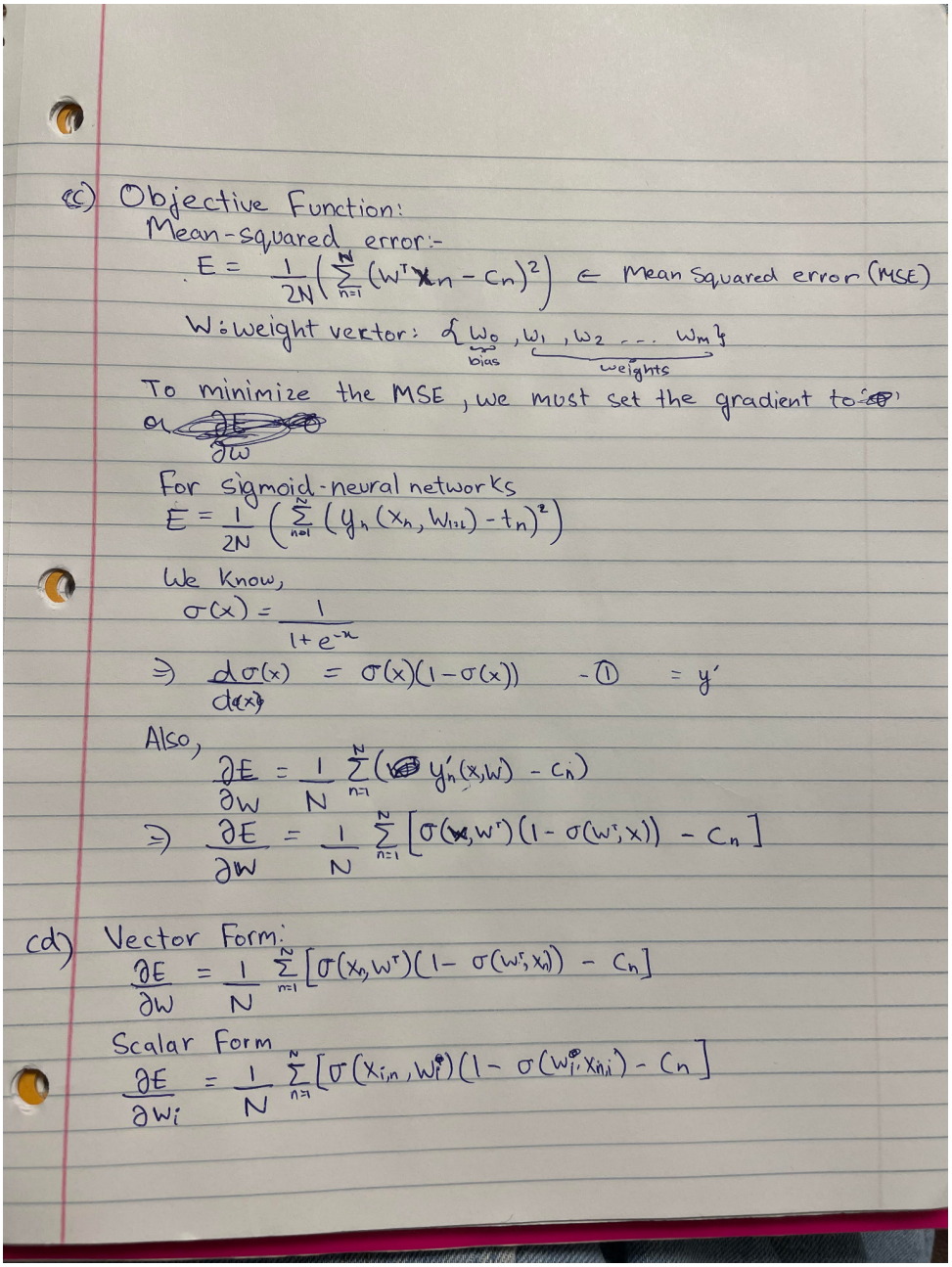



Plot for small error:



(c) Gradient of Objective Function:

(d) Scalar Form and vector form:



(c) Objective Function:

Mean-squared error:-

$$E = \frac{1}{2N}\left(\sum_{n=1}^{N}(w^T x_n - c_n)^2\right) \leftarrow \text{mean squared error (MSE)}$$

W : weight vector: $\{\underbrace{w_0}_{\text{bias}}, \underbrace{w_1, w_2 \ldots w_m}_{\text{weights}}\}$

To minimize the MSE, we must set the gradient to 0

or $\frac{\partial E}{\partial w}$

For sigmoid-neural networks

$$E = \frac{1}{2N}\left(\sum_{n=1}^{N}(y_n(x_n, w_{1:L}) - t_n)^2\right)$$

We know,

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\Rightarrow \frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x)) \quad -① \quad = y'$$

Also,

$$\frac{\partial E}{\partial w} = \frac{1}{N}\sum_{n=1}^{N}(y_n'(x, w) - c_n)$$

$$\Rightarrow \frac{\partial E}{\partial w} = \frac{1}{N}\sum_{n=1}^{N}\left[\sigma(x, w^T)(1-\sigma(w^T x)) - c_n\right]$$

(d) Vector Form:

$$\frac{\partial E}{\partial w} = \frac{1}{N}\sum_{n=1}^{N}\left[\sigma(x_n, w^T)(1-\sigma(w^T x_n)) - c_n\right]$$

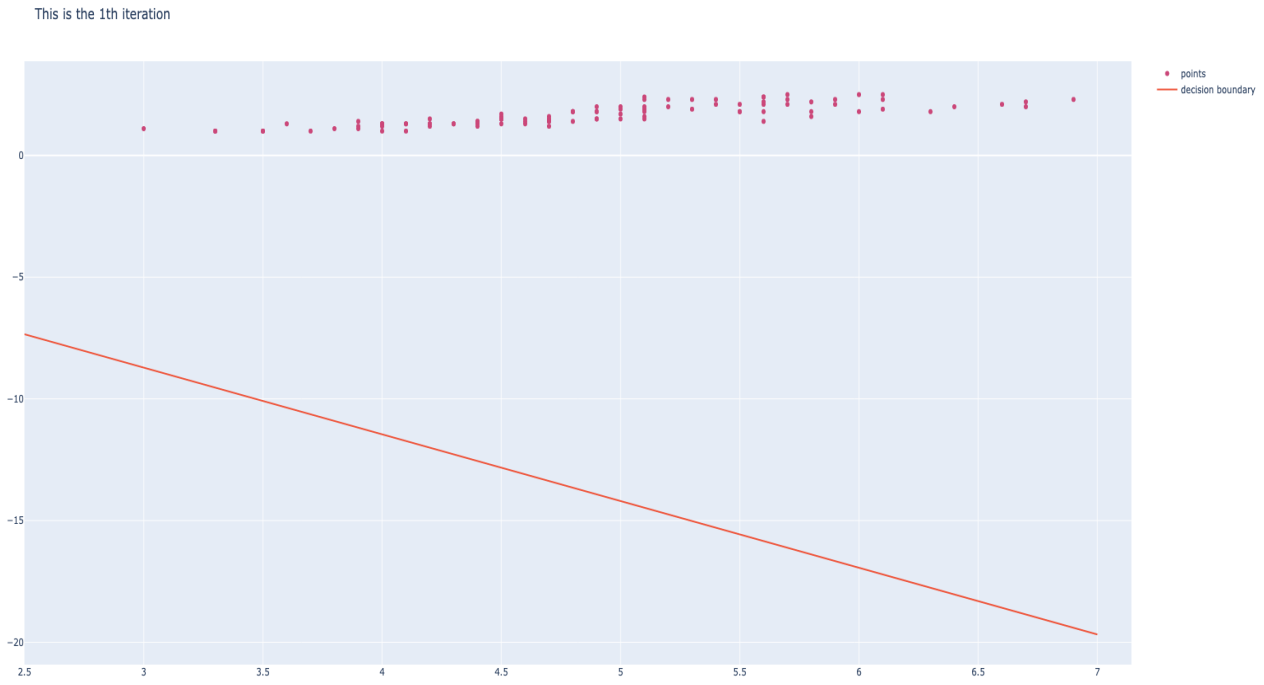
Scalar Form

$$\frac{\partial E}{\partial w_i} = \frac{1}{N}\sum_{n=1}^{N}\left[\sigma(x_{i,n}, w_i)(1-\sigma(w_i x_{n,i}) - c_n\right]$$

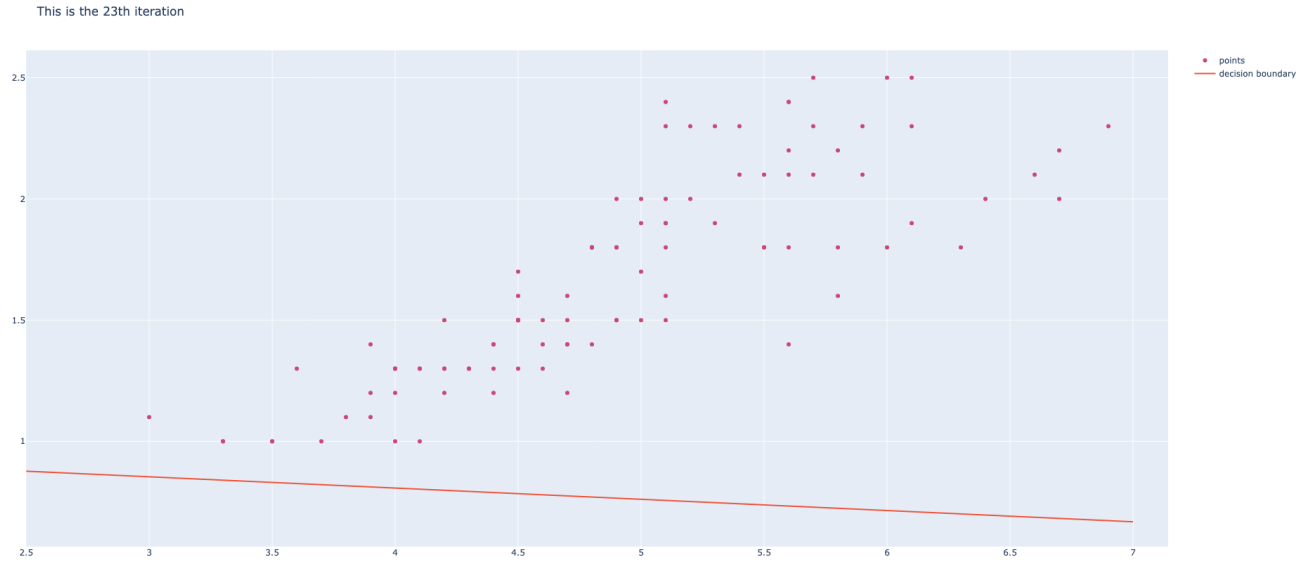
(e) Summed gradient and plots for small step size:

Here are a few plots from running the summed gradient python module, which plots every single linear decision boundary:

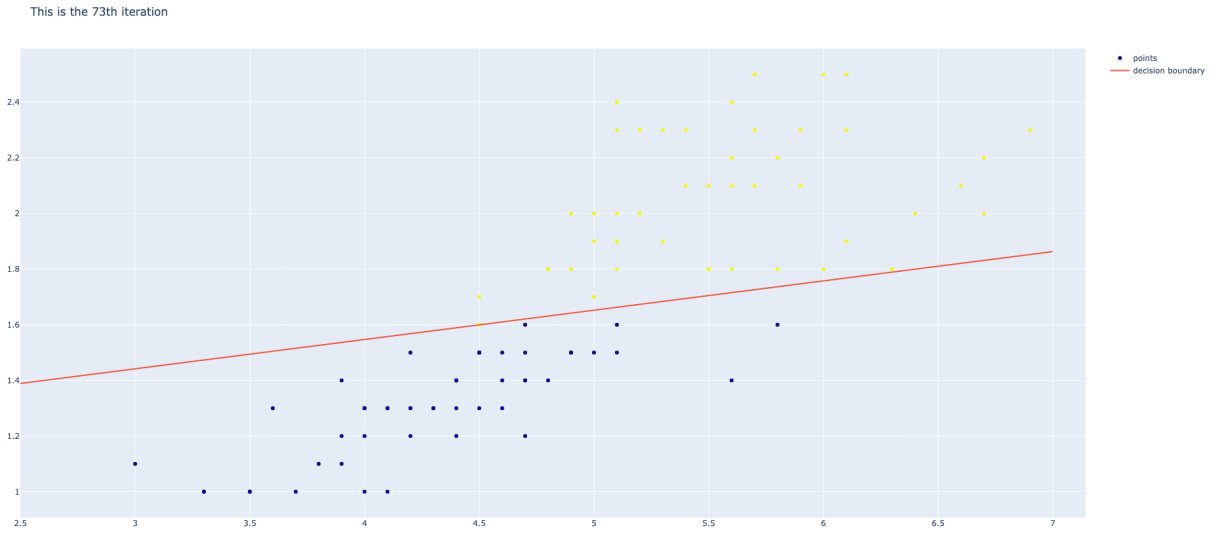
1st iteration:



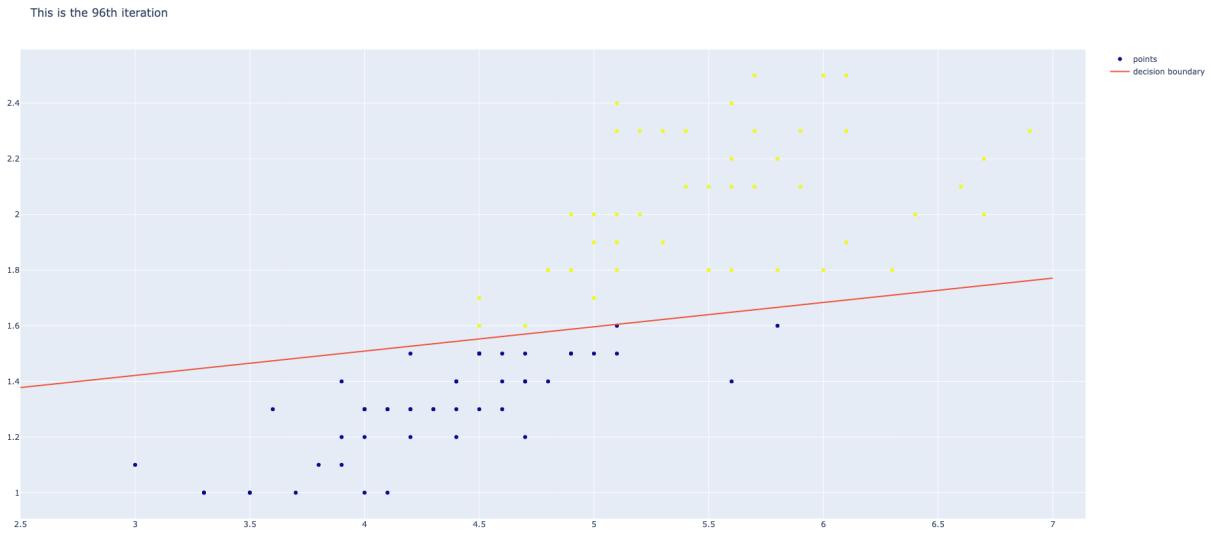

23rd iteration:



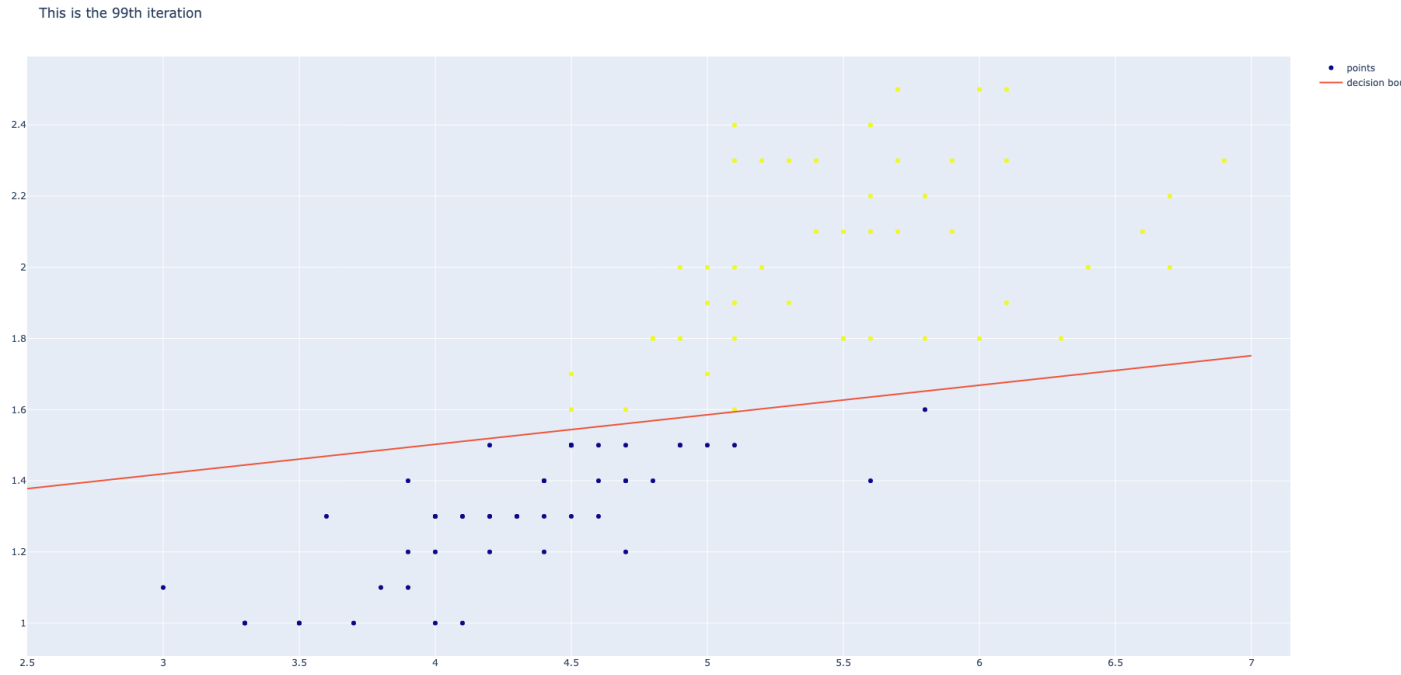
73rd Iteration:




96th Iteration:



99th Iteration (Optimal coincidently)



Exercise 4. Learning through optimization:

(a) Gradient Descent:

Update Rule: This needs to be divided by ‘n’ since we aren’t using Total Squared error

$$w_i^{t+1} = w_i^t - \epsilon \frac{\partial E}{\partial w_i}$$

$$\frac{\partial E}{\partial w_i} = \sum_{n=1}^{N} (\mathbf{w}^T \mathbf{x}_n - c_n) x_{i,n}$$

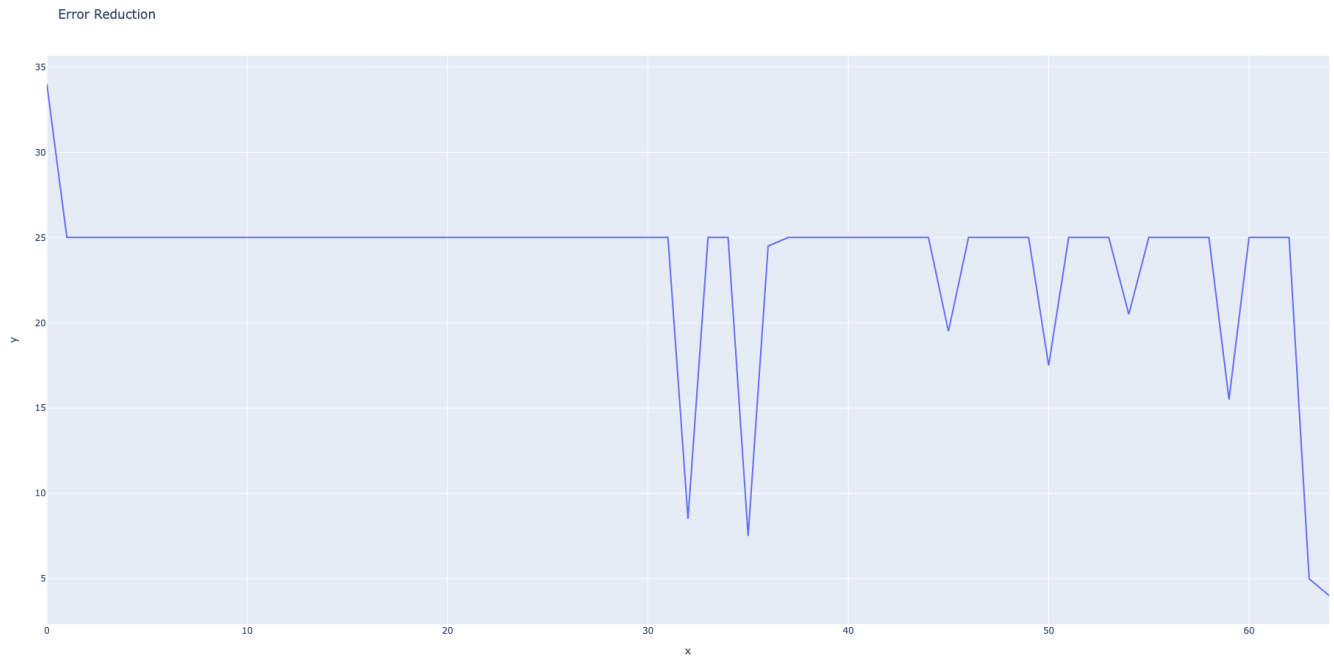
Summed Gradient Helper Function:

```
#Function that calculates the summed gradient
def get_summed_gradient(w,b,data,target):
    gradient = []
    bias_gradient = 0

    _class = predict(data,w,b)

    #For the bias gradient we just have to sum up all of the differences in the prediction and target
    for i in range(0,len(data),1):
        bias_gradient += _class[i] - target[i]

    #Iterating through all of the weights
    for i in range(0,len(w),1):
        grad_sum = 0

        #Iterating through all of the data points
        for j in range(0,len(data),1):
            temp = _class[j] - target[j]
            temp *= data[j][i]
            grad_sum += temp

        gradient.append(grad_sum)

    return np.array(gradient),bias_gradient
```

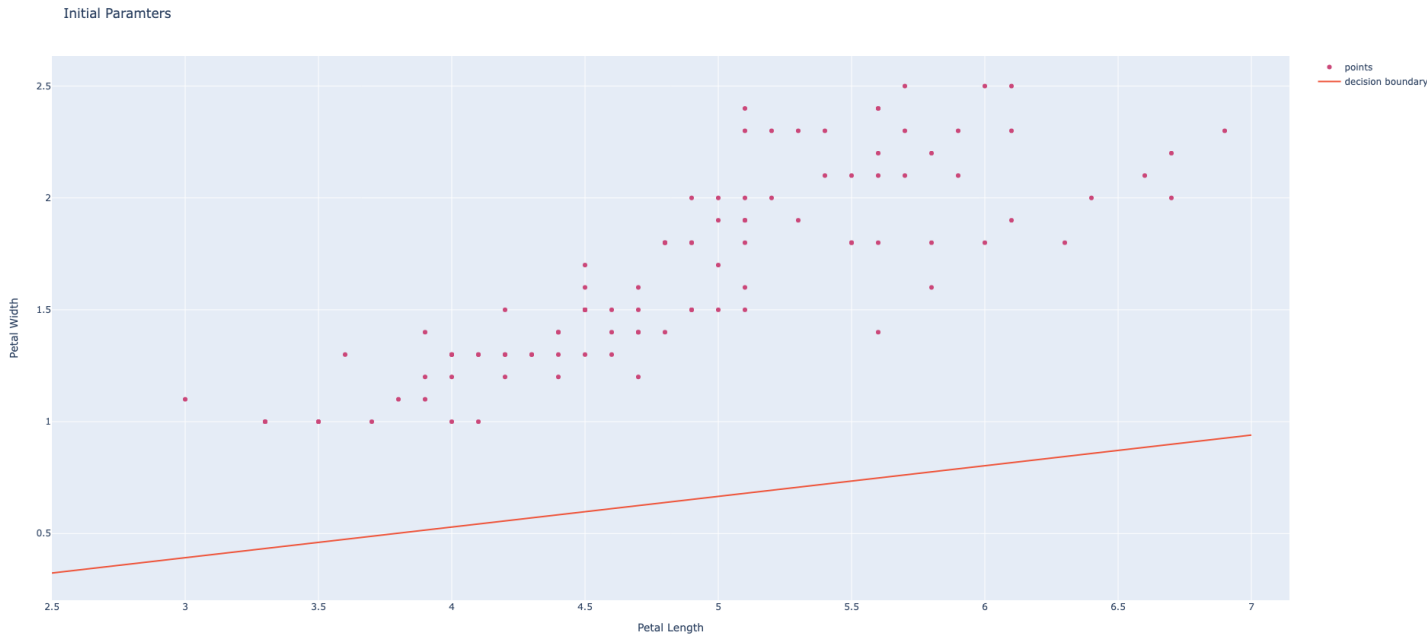
Update Rule Helper Function:

```python
#Function for gradient descent
def gradeient_descent(init_w ,init_b ,data ,target ,eps, max_iter):
    #Initializing the weights and bias
    w = init_w
    b = init_b
    error_list = []

    w_middle = np.array([0.0,0.0])
    b_middle = 0

    w_list = []
    b_list = []

    for i in range(0,max_iter,1):
        #Getting the predicted class
        pred = predict(data,w,b)

        w_list.append(w)
        b_list.append(b)

        #Getting the mean sqaured error
        temp_mse = calculate_MSE(pred, target)
        error_list.append(temp_mse)

        #Leaving if the mean sqaured error is less than 10
        if(temp_mse < 5):
            break

        #Getting the gradients
        weight_grad, bias_grad = get_summed_gradient(w,b,data,target)

        #Updating the weights and bias
        w -= eps*weight_grad
        b -= eps*bias_grad


    return w, b, w_list[int(len(w_list)/2)], b_list[int(len(b_list)/2)], error_list
```

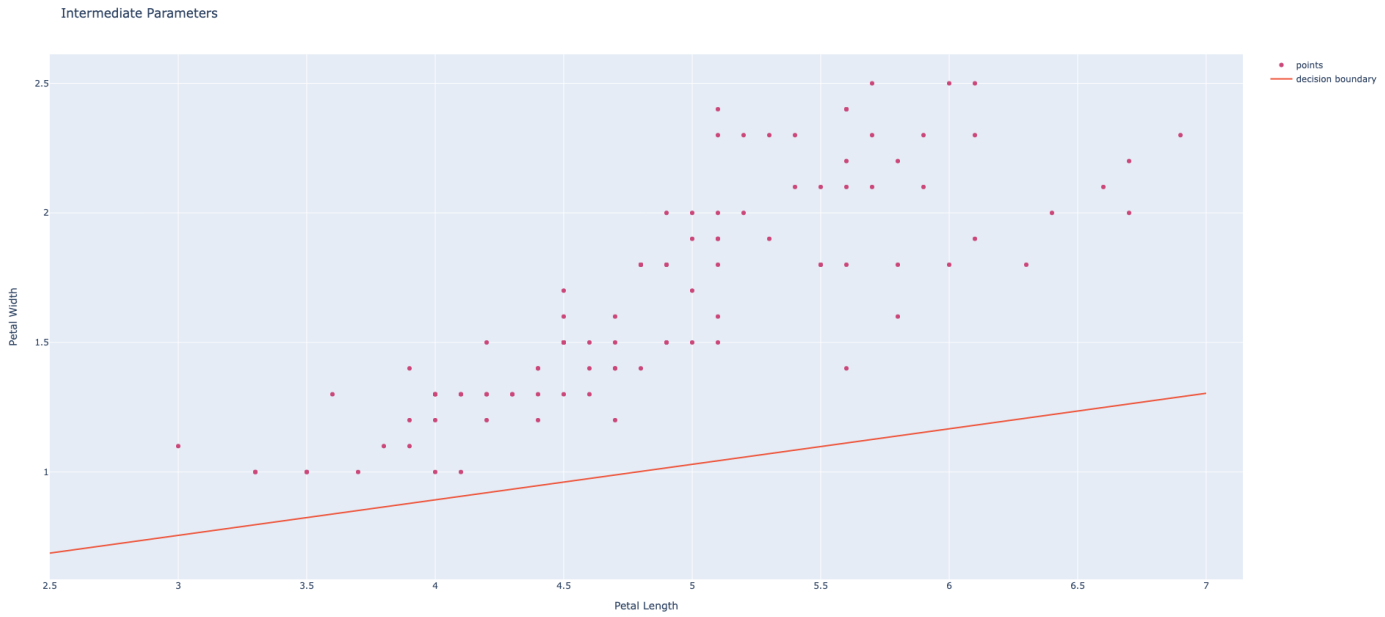
(b) Decision Boundary and Learning Curve, other plots for results are in (c):
Error Reduction Plot over the number of iterations:



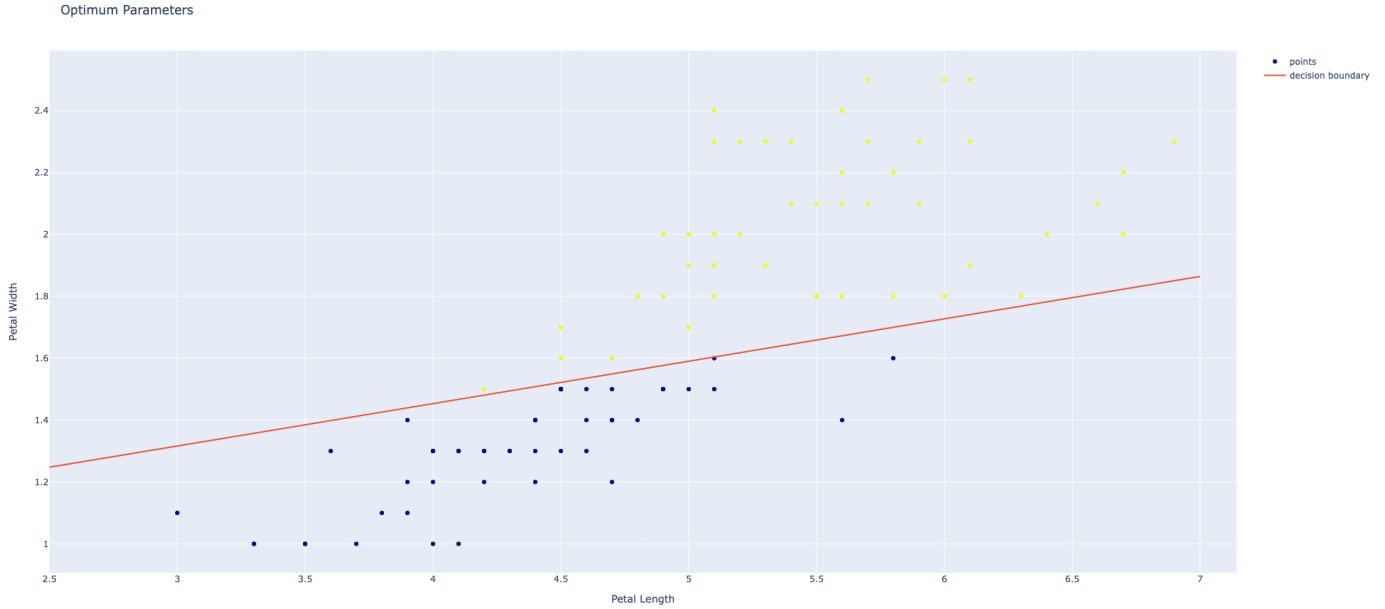

(c) Initial, Middle and Final Decision Boundaries:
Initial:



Intermediate:




Final:



(d) For my step size I looked over at the slides, in which it was mentioned to use 0.1/n as the step size and since I was using Total Squared error, I had to divide it by n. I also tried using values like 1/n, 1/n^2 etc. and found out that 1/n was the optimum step size, Hence I chose my step size to be 0.01 as my step size.

(e) I chose the stopping criteria by using the total squared error function value that I had implemented above as well as having a maximum iteration argument for the method. So whichever one would be reached first would be the stopping criteria. I chose the MSE as 5 and Maximum Iteration count as 1000. I determined the value consistent from this reading: https://www.cs.cmu.edu/~ggordon/10725-F12/scribes/10725_Lecture5.pdf. And the maximum number of steps from https://www.andreaperlato.com/theorypost/gradient-descent-step-by-step/#:~:text=In%20practice%2C%20the%20Minimum%20Step,equal%20to%201000%20or%20greater.