CSDS 391 P2

Jhean Toledo

November 2022

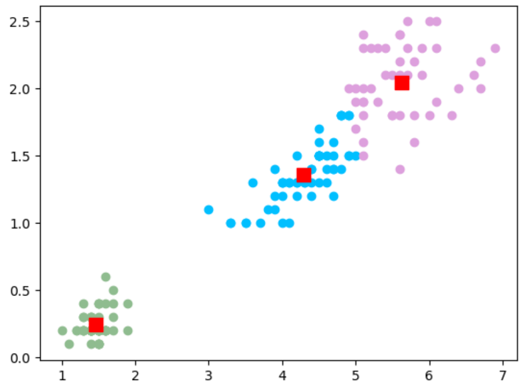
Code Design

My code is broken up into several segments, with one .py file for each exercise, one main function that's just for convenience to have all my outputs in one file, and a data generator file that helps me generate the iris data and initialize certain values.

Exercise 1: Clustering

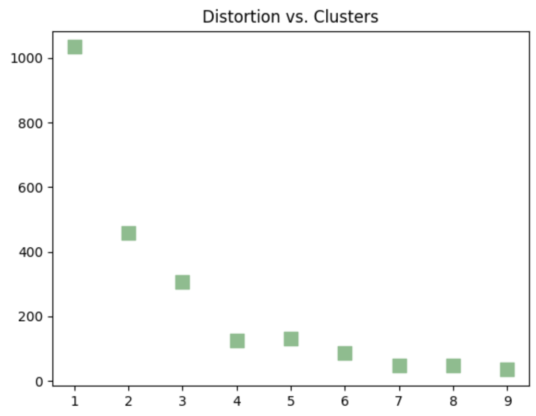
a)

The code for this can be found in exercise1.py in a function called part_a_output()



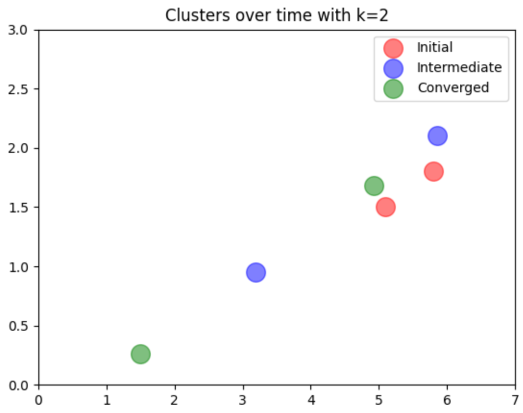
b)

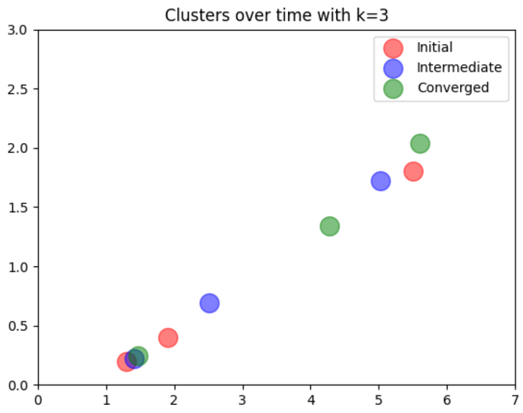
The code for this can be found in exercise1.py in a function called part_b_output(). This demonstrates that the distortion between the clusters decreases as more clusters are introduced



c)

The code for these plots can be found in exercise1.py in a function called part_b_output().

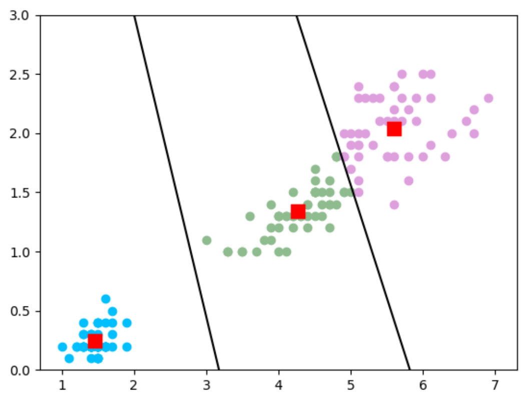




d)

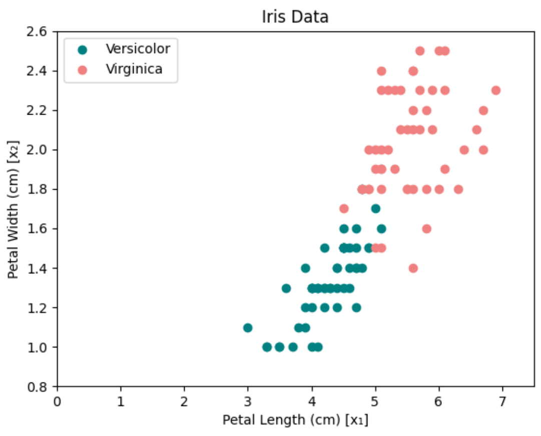
In order to plot the decision boundaries, I wrote a command in my code in exercise1.py called part_d_output. This can be seen below. I went about making the decision boundaries by simply making a line from each centroid and then making a perpendicular line from those intersections. This results in the black lines that we see.

It produces the following graph



Exercise 2: Linear decision boundaries

a)



b)

We can write the function as follows:

$$\sigma(y) = \frac{1}{1+e^{-y}}, \text{ where } y = m_1x_1 + m_2x_x + b$$

Where x is the data vector with x_1 being the petal length and x_2 being the petal width and y is the output of that vector.

c)

A small section of code is listed below. This is part of my plot_iris_data(...) function. The part shown here is the math used in order to plot the points. The same code was used to make the other graphs with an input to indicate whether decision boundaries should be shown or not.

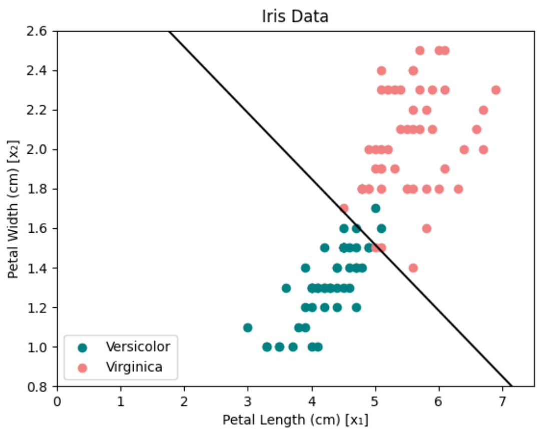
Code:

```
def plot_iris_data(...):
    w = np.zeros(3)
    if len(kwargs) == 2:  # can work with either (w) or (m and b)
        w[0] = kwargs['b']
        w[1:3] = kwargs['m']
    elif len(kwargs) == 1:
        w = kwargs['w']

    versicolor_petal_length = []
    versicolor_petal_width = []
    virginica_petal_length = []
    virginica_petal_width = []
    for p_l, p_w, s in zip(petal_length, petal_width, species):
        if s == 'versicolor':
            versicolor_petal_length.append(p_l)
            versicolor_petal_width.append(p_w)
        elif s == 'virginica':
            virginica_petal_length.append(p_l)
            virginica_petal_width.append(p_w)

    # Drawing the decision boundaries
    if boundaries == True:
        x_ones = np.linspace(0, 7.5, 75)
        x_twos = []
        iris_decision_boundary = decision_boundary(w=w)
        for x_one in x_ones:
            x_twos.append(iris_decision_boundary.get_x_two(x_one))
```

The code produces the following plot when given $m = \begin{bmatrix} 0.6 \\ 1.8 \end{bmatrix}$ and $b = -5.73$.

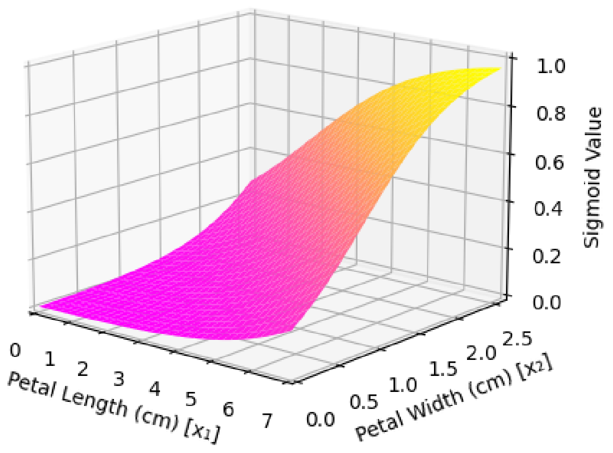


d)

The code used to make this graph is seen below called, 3D_plot(...) which simply takes in m and b. Again, only the calculation portion of the code is shown, not the plotting. For further reference, please see the source code.

```
def surface_plot_input_space(m, b):
    x_one = np.linspace(0, 7, num=70+1)  # Petal Length
    x_two = np.linspace(0, 2.6, num=26+1)  # Petal Width

    fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
    x_one, x_two = np.meshgrid(x_one, x_two)
    iris_data_classifier = simple_classifier(m=m, b=b)
    sigmoid = iris_data_classifier.classify(x_one, x_two)
```



e)

In order to show the output of the decision boundaries, I analyzed the graph created in part 1c and chose 3 random points from 3 different areas. The first area being those that are certainly Versicolor, the other being certainly Virginica, and the last being near the decision boundary to show some uncertainty.

Results Certainly Versicolor:
Petal Length: 4.7 , Petal Width: 1.4 , True Class: versicolor , Simple Classifier Output: 0.4037 (versicolor)
Petal Length: 3.5 , Petal Width: 1.0 , True Class: versicolor , Simple Classifier Output: 0.1382 (versicolor)
Petal Length: 3.8 , Petal Width: 1.1 , True Class: versicolor , Simple Classifier Output: 0.1869 (versicolor)

Certainly Virginica:
Petal Length: 6.0 , Petal Width: 2.5 , True Class: virginica , Simple Classifier Output: 0.9145 (virginica)
Petal Length: 5.7 , Petal Width: 2.3 , True Class: virginica , Simple Classifier Output: 0.8618 (virginica)
Petal Length: 5.6 , Petal Width: 2.4 , True Class: virginica , Simple Classifier Output: 0.8754 (virginica)

Near the Decision Boundary:
Petal Length: 4.5 , Petal Width: 1.7 , True Class: virginica , Simple Classifier Output: 0.5075 (virginica)
Petal Length: 5.0 , Petal Width: 1.5 , True Class: virginica , Simple Classifier Output: 0.4925 (versicolor)
Petal Length: 5.1 , Petal Width: 1.6 , True Class: versicolor , Simple Classifier Output: 0.5523 (virginica)

Exercise 3: Neural Networks

a)

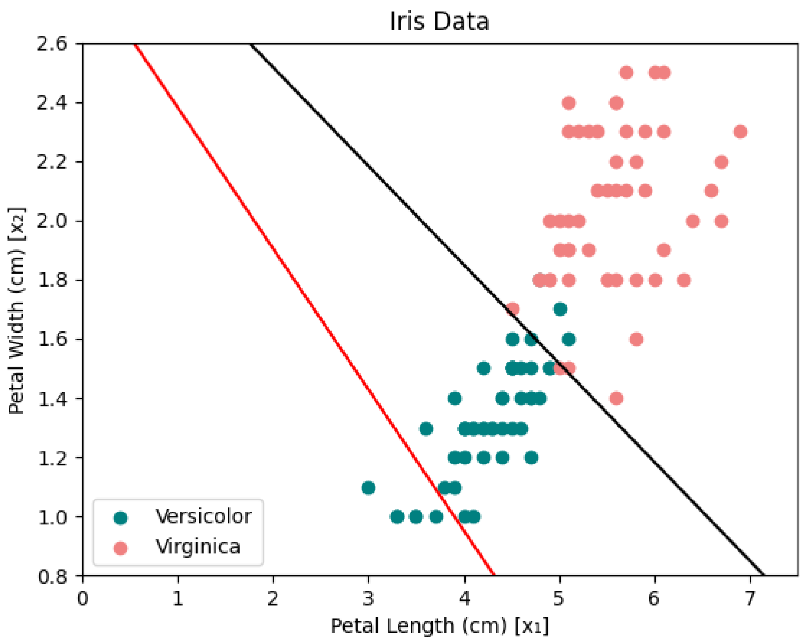
The code for calculating the mean squared error can be found in exercise3.py as a function called mean_squared_error(...)
The code for this can also be seen below for convenience

```
def mean_squared_error(X, m, b, species):
    iris_classifer = exercise2.simple_classifier(m=m, b=b)
    mean_squared_error_sum = 0
    for i in range(len(X[0, :])):
        x_one = X[0, i]
        x_two = X[1, i]
        prediction = iris_classifer.classify(x_one, x_two)
        ground_truth = None
        if species[i] == "versicolor":
            ground_truth = 0
        elif species[i] == "virginica":
            ground_truth = 1
        mean_squared_error_sum += (ground_truth - prediction)**2

    return mean_squared_error_sum/len(X[0, :])
```

b)

I used two sets of parameters to represent a good choice and a bad choice. The parameters I used for a good choice was $m = [0.6|1.8]$ and $b = -5.73$. As for the bad choice, I used $m = [1.0|2.1]$ and $b = -6$. The code I used to plot this data is called plot_iris_data_with_two_boundaries with the plot itself depicted below. The black line will be the decision boundary using a good choice and the red line representing the decision boundary made with a bad choice.



As we can see, the black line does a much better job of classifying the data as compared to the red line.

c)

We can write the objective function as such

$$(y(\mathbf{x}) - \sigma(\boldsymbol{\omega}^T\mathbf{x}))^2$$

Where $y(x)$ =the true class, $\omega = \begin{bmatrix}\omega_0\\\omega_1\\\omega_2\end{bmatrix}$ which are the weights, and $x = \begin{bmatrix}1\\x_1\\x_2\end{bmatrix}$ as the input data

Taking the partial derivative of the objective function with respect to ω_i

$$\begin{aligned}
\frac{\partial}{\partial\omega_i}(y(\mathbf{x}) - \sigma(\boldsymbol{\omega}^T\mathbf{x}))^2 &= 2(y(\mathbf{x}) - \sigma(\boldsymbol{\omega}^T\mathbf{x}))\frac{\partial}{\partial\omega_i}(y(\mathbf{x}) - \sigma(\boldsymbol{\omega}^T\mathbf{x})) && \textit{using chain rule}\\
&= 2(y(\mathbf{x}) - \sigma(\boldsymbol{\omega}^T\mathbf{x}))(-\sigma'(\boldsymbol{\omega}^T\mathbf{x}))\frac{\partial}{\partial\omega_i}(\boldsymbol{\omega}^T\mathbf{x}) && \textit{using chain rule again}\\
&= 2(y(\mathbf{x}) - \sigma(\boldsymbol{\omega}^T\mathbf{x}))(-\sigma'(\boldsymbol{\omega}^T\mathbf{x}))x_i\\
&= -2(y(\mathbf{x}) - \sigma(\boldsymbol{\omega}^T\mathbf{x}))(\sigma(\boldsymbol{\omega}^T\mathbf{x})(1-\sigma(\boldsymbol{\omega}^T\mathbf{x})))x_i && \textit{writing out } \sigma'(\mathbf{x})\\
&= -2(y(\mathbf{x}) - \frac{1}{1+e^{-\boldsymbol{\omega}^T\mathbf{x}}})(\frac{1}{1+e^{-\boldsymbol{\omega}^T\mathbf{x}}}(1-\frac{1}{1+e^{-\boldsymbol{\omega}^T\mathbf{x}}}))x_i && \textit{making the } \sigma\textit{'s explicit}\\
&= -2(y(\mathbf{x}) - \frac{1}{1+e^{-\boldsymbol{\omega}^T\mathbf{x}}})((\frac{1}{1+e^{-\boldsymbol{\omega}^T\mathbf{x}}} - \frac{1}{(1+e^{-\boldsymbol{\omega}^T\mathbf{x}})^2}))x_i && \textit{simplify}\\
&= -2(y(\mathbf{x}) - \frac{1}{1+e^{-\boldsymbol{\omega}^T\mathbf{x}}})((\frac{1+e^{-\boldsymbol{\omega}^T\mathbf{x}}}{(1+e^{-\boldsymbol{\omega}^T\mathbf{x}})^2} - \frac{1}{(1+e^{-\boldsymbol{\omega}^T\mathbf{x}})^2}))x_i\\
&= -2(y(\mathbf{x}) - \frac{1}{1+e^{-\boldsymbol{\omega}^T\mathbf{x}}})((\frac{e^{-\boldsymbol{\omega}^T\mathbf{x}}}{(1+e^{-\boldsymbol{\omega}^T\mathbf{x}})^2}))x_i
\end{aligned}$$

d)

We found the gradient of the objective for one weight in part c as:

$$\frac{\partial}{\partial \omega_i}(y(\mathbf{x}) - \sigma(\boldsymbol{\omega}^T\mathbf{x}))^2 = -2(y(\mathbf{x}) - \frac{1}{1+e^{-\boldsymbol{\omega}^T\mathbf{x}}})((\frac{e^{-\boldsymbol{\omega}^T\mathbf{x}}}{(1+e^{-\boldsymbol{\omega}^T\mathbf{x}})^2}))x_i$$

Therefore, for the **scalar** weights, the gradient can be written as:

for ω_0: $\frac{\partial}{\partial \omega_0}(y(\mathbf{x}) - \sigma(\boldsymbol{\omega}^T\mathbf{x}))^2 = -2(y(\mathbf{x}) - \frac{1}{1+e^{-\boldsymbol{\omega}^T\mathbf{x}}})((\frac{e^{-\boldsymbol{\omega}^T\mathbf{x}}}{(1+e^{-\boldsymbol{\omega}^T\mathbf{x}})^2}))$

for ω_1: $\frac{\partial}{\partial \omega_1}(y(\mathbf{x}) - \sigma(\boldsymbol{\omega}^T\mathbf{x}))^2 = -2(y(\mathbf{x}) - \frac{1}{1+e^{-\boldsymbol{\omega}^T\mathbf{x}}})((\frac{e^{-\boldsymbol{\omega}^T\mathbf{x}}}{(1+e^{-\boldsymbol{\omega}^T\mathbf{x}})^2}))x_1$

for ω_2: $\frac{\partial}{\partial \omega_2}(y(\mathbf{x}) - \sigma(\boldsymbol{\omega}^T\mathbf{x}))^2 = -2(y(\mathbf{x}) - \frac{1}{1+e^{-\boldsymbol{\omega}^T\mathbf{x}}})((\frac{e^{-\boldsymbol{\omega}^T\mathbf{x}}}{(1+e^{-\boldsymbol{\omega}^T\mathbf{x}})^2}))x_2$

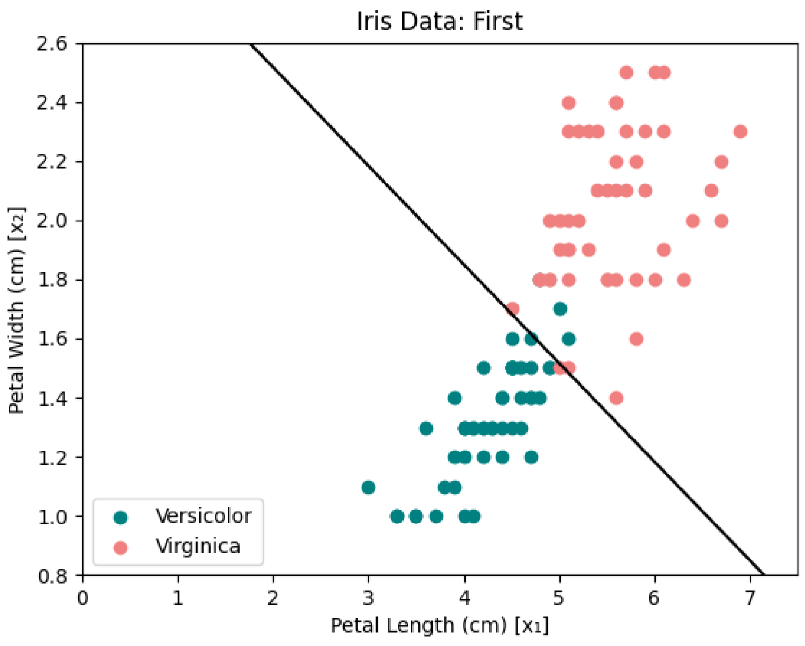
And the **vector** gradient can be written as a collection of the scalar expressions:

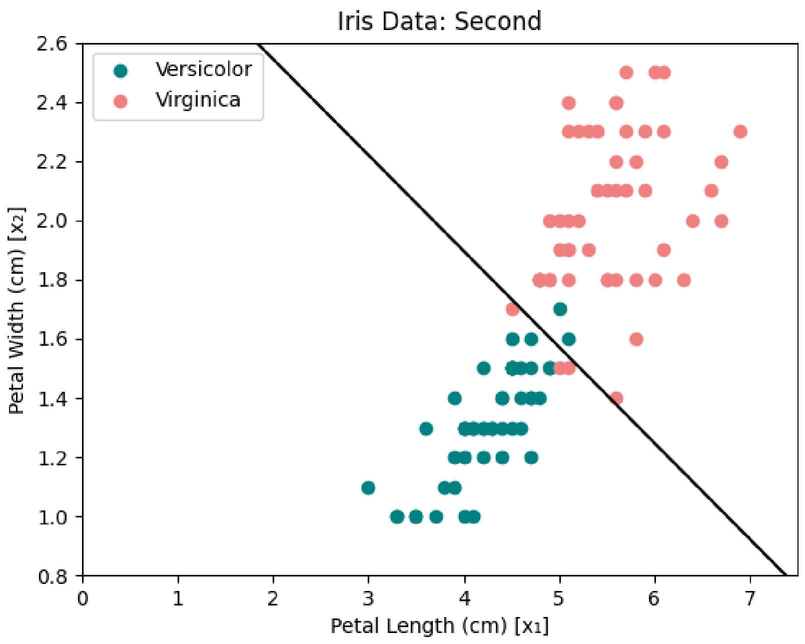
$$\nabla(y(\mathbf{x}) - \sigma(\boldsymbol{\omega}^T\mathbf{x}))^2 = \begin{Bmatrix} \frac{\partial}{\partial\omega_0} \\ \frac{\partial}{\partial\omega_1} \\ \frac{\partial}{\partial\omega_2} \end{Bmatrix} = \begin{Bmatrix} -2(y(\mathbf{x}) - \frac{1}{1+e^{-\boldsymbol{\omega}^T\mathbf{x}}})((\frac{e^{-\boldsymbol{\omega}^T\mathbf{x}}}{(1+e^{-\boldsymbol{\omega}^T\mathbf{x}})^2})) \\ -2(y(\mathbf{x}) - \frac{1}{1+e^{-\boldsymbol{\omega}^T\mathbf{x}}})((\frac{e^{-\boldsymbol{\omega}^T\mathbf{x}}}{(1+e^{-\boldsymbol{\omega}^T\mathbf{x}})^2}))x_1 \\ -2(y(\mathbf{x}) - \frac{1}{1+e^{-\boldsymbol{\omega}^T\mathbf{x}}})((\frac{e^{-\boldsymbol{\omega}^T\mathbf{x}}}{(1+e^{-\boldsymbol{\omega}^T\mathbf{x}})^2}))x_2 \end{Bmatrix}$$

Add some words here

e)

To demonstrate that the code learns, I started with a weight vector $\omega = \begin{bmatrix} 4.0 \\ 0.6 \\ 1.8 \end{bmatrix}$. This gave a semi weak decision boundary on the first run, but then readjusted to be better on the second run.




Iris Data: Second
Versicolor
Virginica
Petal Width (cm) [x₂]
Petal Length (cm) [x₁]
0.8
1.0
1.2
1.4
1.6
1.8
2.0
2.2
2.4
2.6
0
1
2
3
4
5
6
7

Exercise 4: Learning a decision boundary through optimization

a)

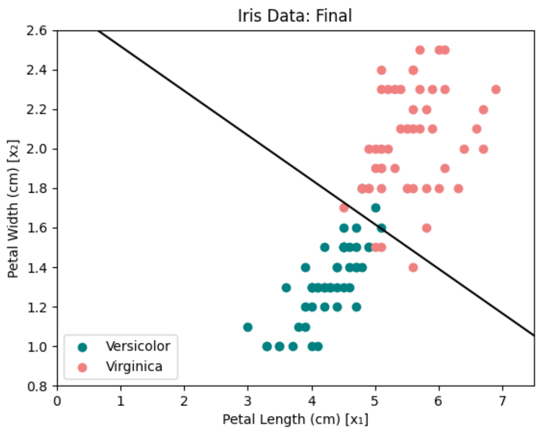
I wrote a function called optimize_boundary(...) in exercise4.py that implements the gradient descent. Part of this can be seen below for your convenience

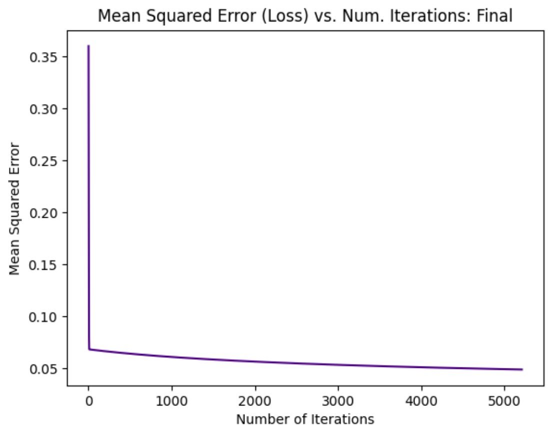
```
def optimize_boundary(...):
    threshold = 0.25
    gradient = 0

    # Parameters for Graphing/Output:
    num_iterations = 0
    weights_store = []
    mse_store = []
    while True:
        num_iterations += 1  # for graphing
        gradient = exercise3.gradient_mean_squared_error(X_augmented, classifier.get_weigh
        new_weights = classifier.get_weights()-step_size*gradient
        if progress_output:
            mse_store.append(exercise3.mean_squared_error(X_augmented[1:3, :], ...
            new_weights[1:3], new_weights[0], species))

    return classifier.get_weights()
```

b)

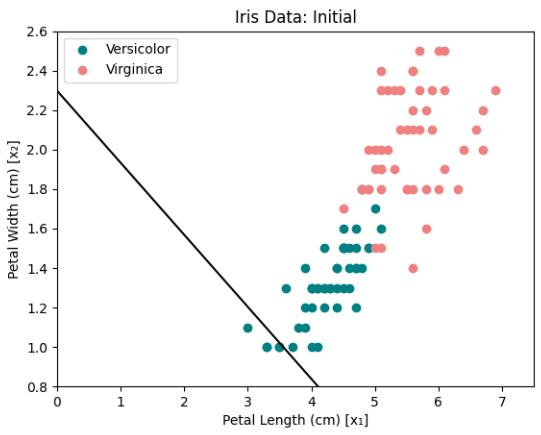

Iris Data: Final
Petal Width (cm) [x₂]
Petal Length (cm) [x₁]
Versicolor
Virginica


Mean Squared Error (Loss) vs. Num. Iterations: Final
Mean Squared Error
Number of Iterations

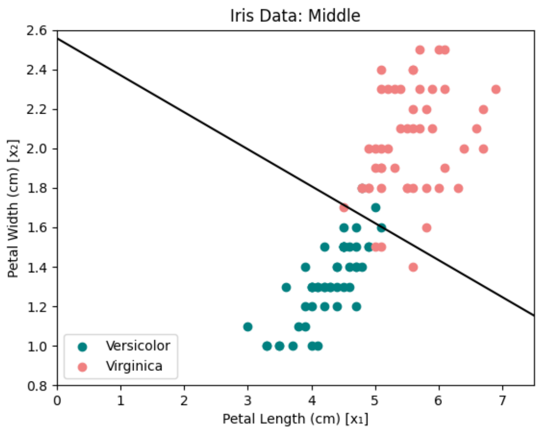
c)

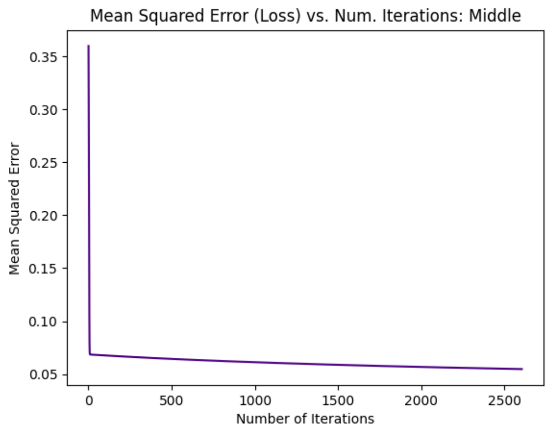
The weight I used at the start was $\omega = \begin{bmatrix} 8.05314604 \\ 1.35341947 \\ 3.53410281 \end{bmatrix}$. After running the code through optimize_boundary(...), it produced the following plots, with increasing accuracy after each iteration.

Initial

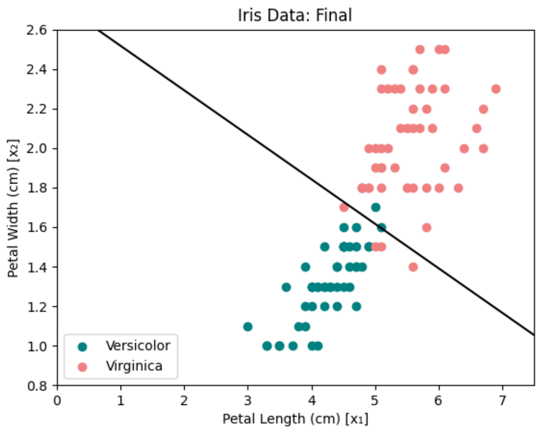


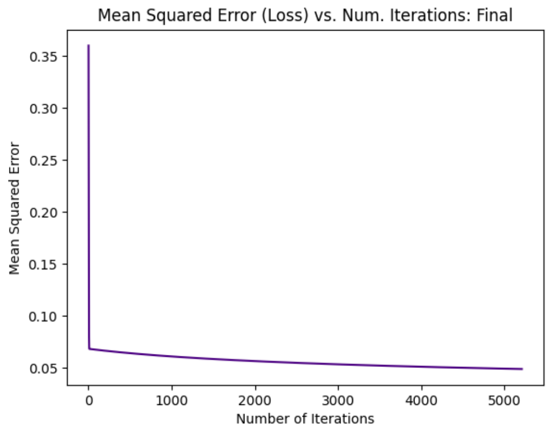
Middle





Final





At the end, I got a weight vector of $\omega = \begin{bmatrix} -12.82893561 \\ 1.05314637 \\ 4.67818169 \end{bmatrix}$

d)

To begin choosing a good step size, I knew that the value would have to be small, since it will approach the minimum loss slowly, meaning that it won't overshoot. The downside however is that it will take much longer to compute. Keeping this in mind, I began with a step size of 0.0001 as it was small. After realizing it took too long, I incremented the step size by factors of ten until I felt as though it would take a reasonable amount of time, jumping to 0.001 and 0.01. However, I realized that having the step size as larger than around 0.005 it made the graph appear weird so I kept changing the step size to numbers around 0.005 until I found a suitable step size of 0.0025.

e)

Using a similar process to choosing a step size, I began roughly around 0.05 to see when the fitting would stop when the gradient mean squared error was less than the threshold I decide to choose. After fiddling around, I found that the value of 0.25 worked the best.