CSDS 391: P2

John Mays

Due and submitted on 11/18/21, Professor Lewicki

A Quick Note on Organization and Code:

In the zipped folder, `johnmays_P2`, there are eight items: this write up as a .pdf, a folder containing the .csv data, and six python files.
There are two helper files:

- `data_generator.py`, which simply reads the .csv data and returns pythonic data structures.
- `exceptions.py`, which contains a few custom python errors.

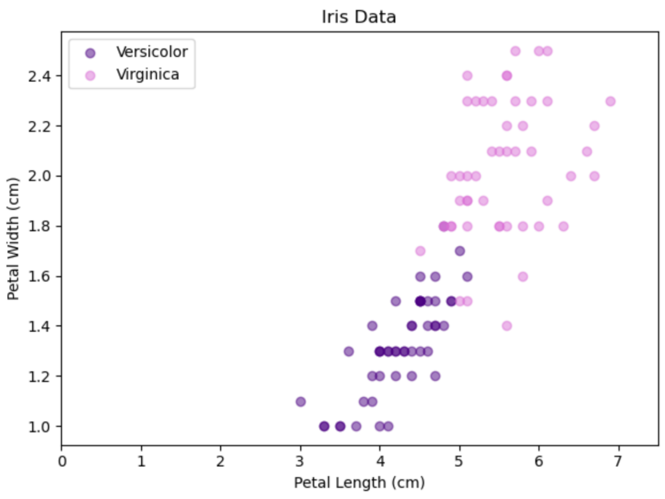
Then there are four files containing the functional code:

- `main.py`, which acts as a hub for the other five files, making all of the main calls and storing the few important global variables such as `step_size`. From here, I have written quite a few functions called "`output_1c()`" or "`output_2e()`", which can be simply called to give the exact output relevant to certain questions.
- `exercise_one.py`, which contains all of the functions implemented for **Exercise 1**.
- `exercise_two.py`, which contains all of the functions implemented for **Exercise 2**.
- `exercise_three.py`, which contains all of the functions implemented for **Exercise 3**.

Additionally, I tried to leave good comments in my code to demonstrate functionality.

Exercise 1: Linear decision boundaries

a)



b)

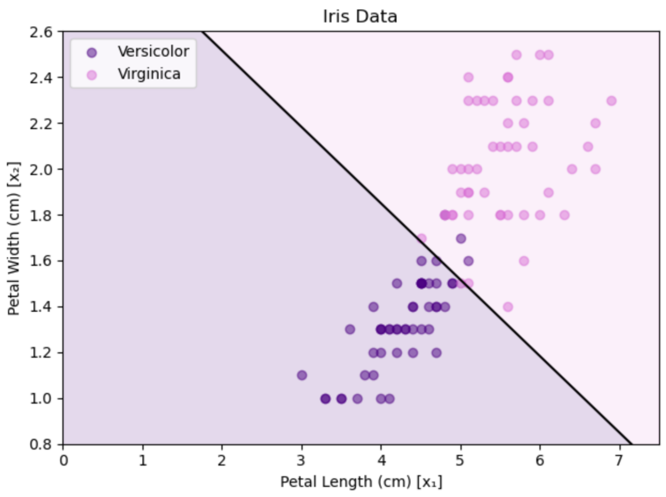
We write our function abstractly as two parts:

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

where $\mathbf{x}$ is the data vector $= \begin{Bmatrix} x_1 = \text{petal length} \\ x_2 = \text{petal width} \end{Bmatrix}$ and y is the output value.

c)

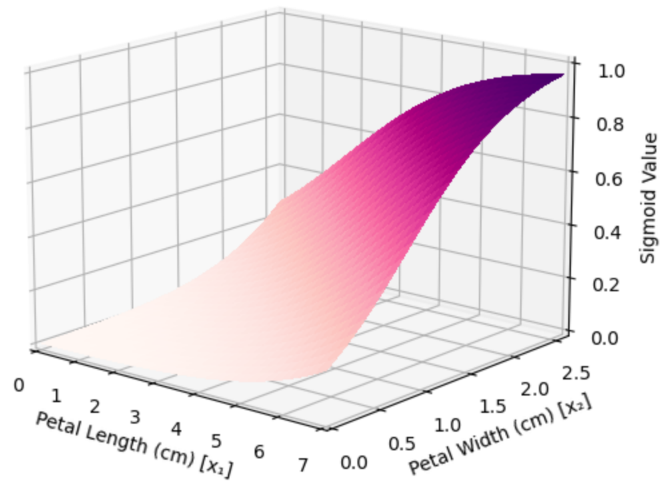
For the parameters, I chose $\mathbf{m} = \begin{bmatrix} 0.6 \\ 1.8 \end{bmatrix}$ and $b = -5.73$



Code:

In `johnmays_P2/exercise_one.py`, refer to `plot_iris_data_with_decision_boundary(...)`, a function incredibly close to `plot_iris_data(...)`, which I used for part **1a**. (This creates the plot above) Additionally, refer to `simple_classifier(x_one, x_two)`, which is the function that actually returns a sigmoid value given input data $\mathbf{x}$.

d)



Code:

Refer to `johnmays_P2/exercise_one.py/surface_plot_input_space(...)`, which plots this figure.

e)

For this question, I wrote a function that, for one data point, prints the petal length, petal width, true species, and simple classifier output. The function is `test_simple_classifier(...)` and, given a CSV row index, calls `simple_classifier(x_one, x_two)` on that data, and prints the required output.
Both functions can be found in `johnmays_P2/exercise_one.py`

Output and Results:

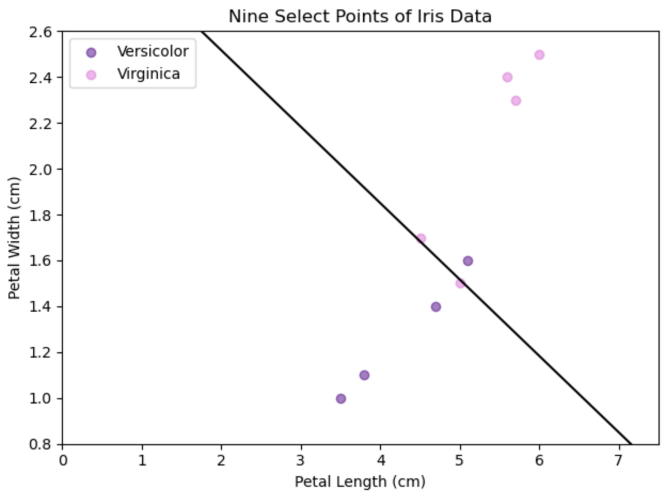
Unambiguously Versicolor:
```
Petal Length:  4.7 , Petal Width:  1.4 ,
True Class:  versicolor , Simple Classifier Output:  0.4037 (versicolor)
Petal Length:  3.5 , Petal Width:  1.0 ,
True Class:  versicolor , Simple Classifier Output:  0.1382 (versicolor)
Petal Length:  3.8 , Petal Width:  1.1 ,
True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

Conclusion:

The classifier returns expected outputs closer to 1 and 0 for the unambiguous data points, but for those near the decision boundary, I chose a virginica (the second one listed) and a versicolor that were both misclassified by the classifier, and another virginica (the first one listed) that was close, but successfully classified. The actual value returned by the classifier for the ambiguous data points is much closer to 0.5.

Data Used:

I plotted the nine data points I used for convenience:



These data points can be found at CSV rows (beginning at 0 with the first setosa) 50, 60, 80, 100, 120, 140, 106, 119, and 83.

Exercise 2: Neural networks

a)

My code essentially calculates this sum:

$$\frac{1}{n}\sum_{i=1}^{n}(\text{class}(\mathbf{x_i}) - \text{prediction}(\mathbf{x_i}))^2$$

My code contains a lot of extraneous information, so I will refer you to `johnmays_P2/exercise_two.py/mse(X, m, b, species)`, and include my pseudocode instead:

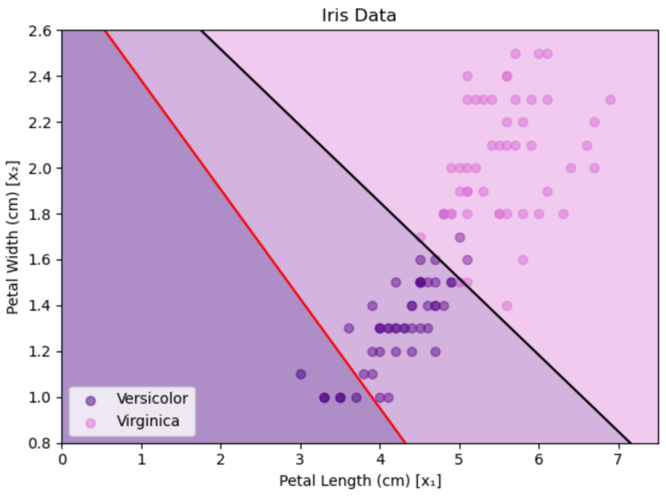
```
MSE($\mathbf{X}$, $\mathbf{m}$, $b$, $species$)
    $sum = 0$
    for column vectors: $\mathbf{x}$ in $\mathbf{X}$
        $sum = sum +$ (binary value corresponding to $\mathbf{x}$'s $species$ $-$ CLASSIFIER($\mathbf{x}$))$^2$
    return $sum \div$ columns in $\mathbf{X}$
```

b)

For a good choice of parameters, I chose what I used in part **1c**: $\mathbf{m} = \begin{bmatrix} 0.6 \\ 1.8 \end{bmatrix}$ and $b = -5.73$

For a bad choice, I chose: $\mathbf{m} = \begin{bmatrix} 1.0 \\ 2.1 \end{bmatrix}$ and $b = -6$.

Both decision boundaries are plotted here:



The original one (black) does a good job splitting the data, but the bad one (red) accurately classifies only a little more than half of the of the data.

Results

Calling my `mse(...)` function with the good parameters returned a mean squared error of 0.0969.
Calling it with the bad parameters returned a mean squared error of 0.2688. Looking at the bad decision boundary, this makes sense, as around half of the data points (mostly virginica and a few versicolor) will be mostly well-classified; however, a large chunk of versicolors are close to the decision boundary, but are misclassified.

Code:

In `johnmays_P2/main.py`, there is a function called `output_2b()` that, when itself called, will call the appropriate functions in `johnmays_P2/exercise_two.py` to plot the decision boundaries and print the mean-squared-error values. Those functions are `mse(...)` and `plot_iris_data_with_two_decision_boundaries(...)`

c)

Setting up the Problem:

With respect to one input data vector $\mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}$, the <u>objective function</u>, which we want to decrease is the mean squared error:

$$\text{MSE}(\mathbf{x}) = (\text{class}(\mathbf{x}) - \text{prediction}(\mathbf{x}))^2$$

To put it in a mathematical form, we will say we have the variables:

$y(\mathbf{x})$ = the true class, $\boldsymbol{\omega} = \begin{bmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \end{bmatrix}$ = the weights, $\mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}$ = the input data,

and the prediction function $\sigma(\boldsymbol{\omega}^T\mathbf{x}) = \sigma(\omega_0 + \omega_1 x_1 + \omega_2 x_2)$.

We can therefore write out the objection function in mathematical terms as:

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

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

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

Specific forms of gradient to be written out in part **2d** $\rightarrow$

d)

The gradient of the objective for one weight, ω_i is:

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

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:

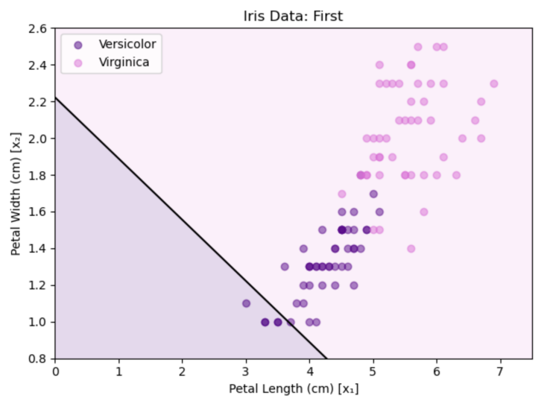
$$\nabla(y(\mathbf{x}) - \sigma(\boldsymbol{\omega}^T\mathbf{x}))^2 = \begin{Bmatrix} \frac{\partial}{\partial \omega_0} \\ \frac{\partial}{\partial \omega_1} \\ \frac{\partial}{\partial \omega_2} \end{Bmatrix} = \begin{Bmatrix} -2(y(\mathbf{x}) - \frac{1}{1+e^{-\boldsymbol{\omega}^T\mathbf{x}}})((\frac{e^{-\boldsymbol{\omega}^T\mathbf{x}}}{(1+e^{-\boldsymbol{\o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

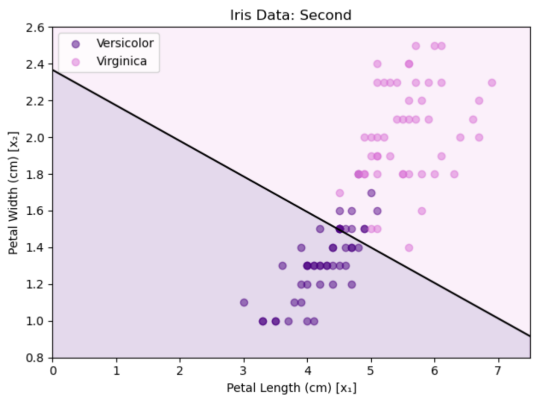
e)

I began with weight vector $\boldsymbol{\omega} = \begin{bmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \end{bmatrix} = \begin{bmatrix} -4.0 \\ 0.6 \\ 1.8 \end{bmatrix}$

Then, I ran my `gradient_mse(...)` function and subtracted the returned gradient sum vector multiplied by a step size of 0.005 from the original weights, to get a new weight vector: $\boldsymbol{\omega} = \begin{bmatrix} -4.0636 \\ 0.3320 \\ 1.7173 \end{bmatrix}$

Here are the first and second weights' decision boundaries plotted:





Code:

In `johnmays_P2/main.py`, there is a function called `output_2e()` that, when itself called, will call a sequence of functions that produce the above output.

The function that actually computes the gradient for a batch of data can be found in `johnmays_P2/exercise_two.py` and is called `gradient_mse(...)`. As its input, it takes a matrix of 3×1 data vectors (with the first entry always being 1), a 3×1 weight vector, and the vector *species*, which contains the ground truth for all of the data points.

To make the plots, I simply call `plot_iris_data_with_decision_boundary(...)` from part **1c**.

Exercise 3: Learning a decision boundary through optimization

a)

In `johnmays_P2/exercise_three.py`, I wrote a function called `fit(...)`, that implements gradient descent.

To `fit(...)`, I pass the X matrix containing 100 3×1 data vectors, the step size variable, and a `simple_classifier(...)` model object that I instantiate with the starting weights beforehand. I also pass it the data vectors `petal_length`, `petal_width`, and `species` so that it can easily generate the required plots for the next section, **3b**.

In addition to producing the appropriate plots (only if you tell it to by passing the argument `progress_output=True`), the `fit(...)` function simply returns a final weights vector.

b)

When calling the `fit(...)` function mentioned in **3a**, if you pass the argument `progress_output=True`, the function will, after learning

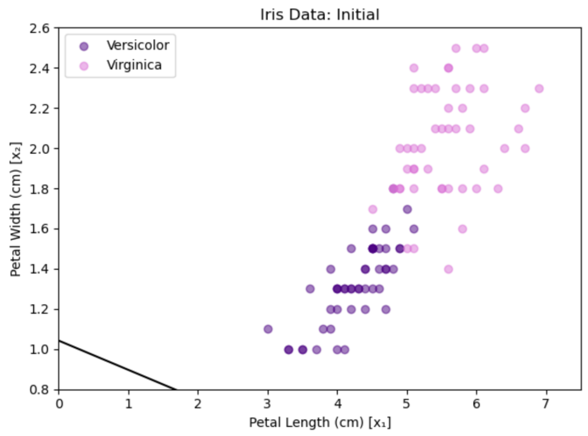
- make three decision boundary plots for the first, middle, and last iteration, by calling the `plot_iris_data_with_decision_boundary(...)` function from `johnmays_P2/exercise_one.py` three times.
- plot the mean squared error (loss) over number of iterations by calling `johnmays_P2/exercise_three.py/plot_loss_over_iterations(...)`.

c)

To begin, I call `johnmays_P2/exercise_three.py/random_weights(...)` to give me a random starting vector. It gave me $\boldsymbol{\omega} = \begin{bmatrix} -3.4153 \\ -0.4774 \\ -3.2777 \end{bmatrix}$

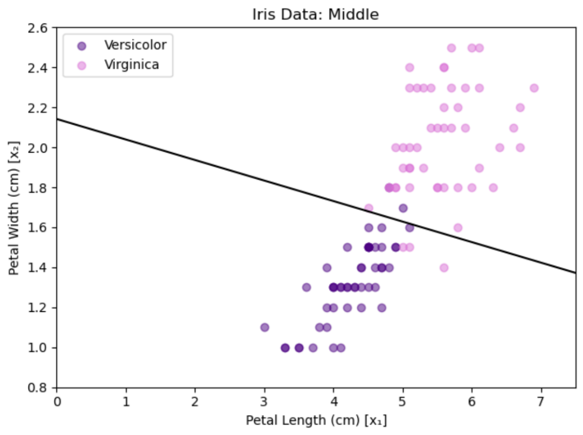
I then instantiate a `simple_classifier(...)` object with the random weights, and call `fit(..., progress_output=True)`, which makes plots for three stages of learning. Here is an example run based on the above random weights:

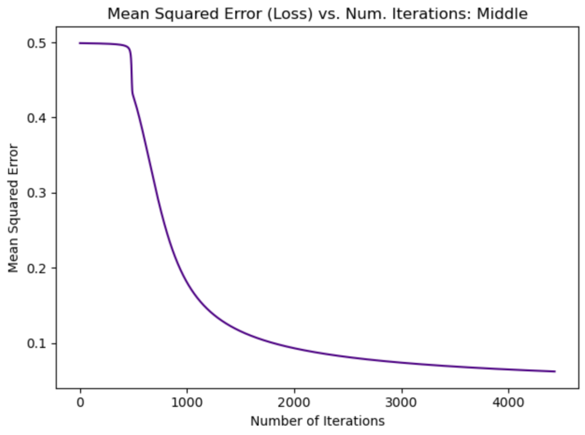
Initial:



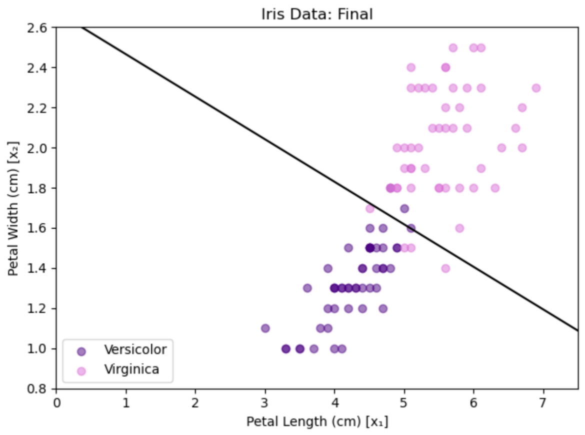
no loss graph because it would be empty at the beginning

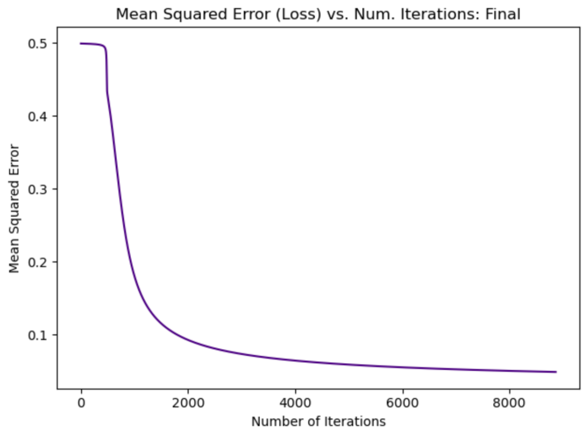
Middle:





Final:





Results:

The `fit(...)` function, once converged, returns a final weight vector of $\boldsymbol{\omega} = \begin{bmatrix} -12.8066 \\ 1.0138 \\ 4.7827 \end{bmatrix}$

Code:

From `johnmays_P2/main.py`, this entire process can be run just with the command `output_3c()`. This command relies on the functions described just above, which can be located in `johnmays_P2/exercise_three.py`.

d)

I landed on `step_size`$= \alpha = 0.0025$.

The objective of setting the correct step size is, given a range of gradient values, to approach the minimum loss slowly enough that you do not overshoot. Given this criteria, you could set it very small, but the smaller it is, the more computing resources are used.
Therefore I, being familiar with typical step sizes as values around $[0.01, 0.0001]$. started with 0.0001, which took longer than I was willing to wait. Therefore I increased it to 0.01, but that let to very wonky behavior in the loss graphs, so I settled on the order of 0.001, and fine-tuned it until I thought the loss graphs were easy to read.

e)

Theoretically speaking, the stopping criterion is when $|\nabla \text{MSE}| = 0$. However, I knew that was probably unrealistic considering step size is not infinitesimal, there are rounding errors, etc. So I decided to test values for a threshold such that the fitting would stop when $|\nabla \text{MSE}|$ is less than the threshold. Inspecting my loss vs. number of iterations graphs, the change in loss plateaued over time, and a threshold value of 0.25 seemed to eliminate most of the redundant plateauing.