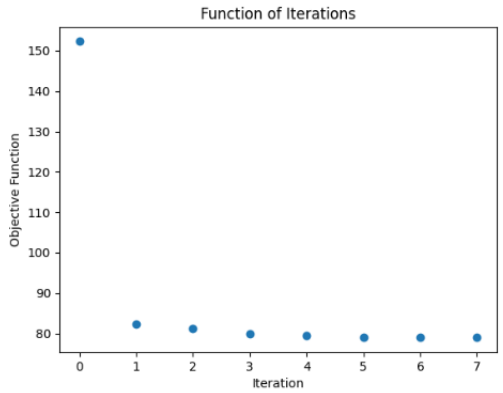
Hunter Welch

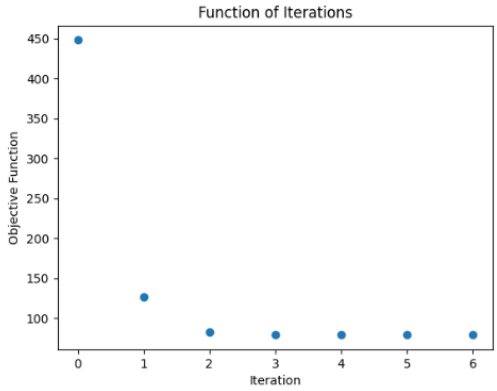
Ai P2 Write Up

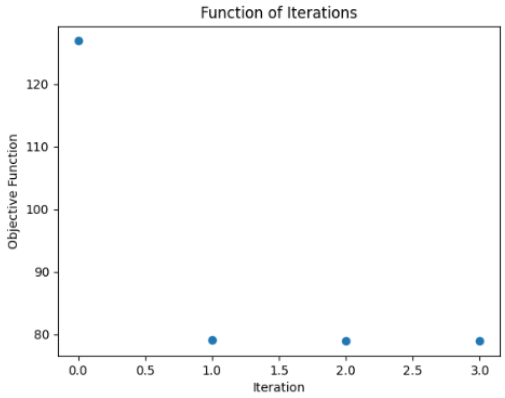
Ex 1.

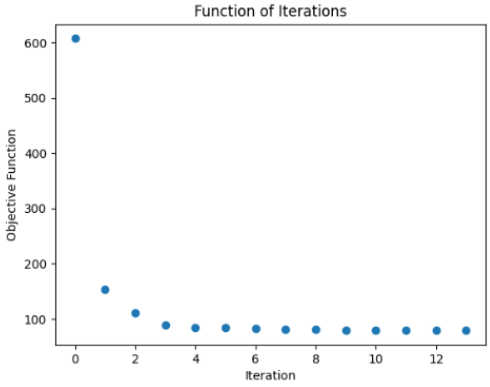
a. see code in ex1.py

b. Graph varies depending on initial means. Here are some examples (utilizing k=3):





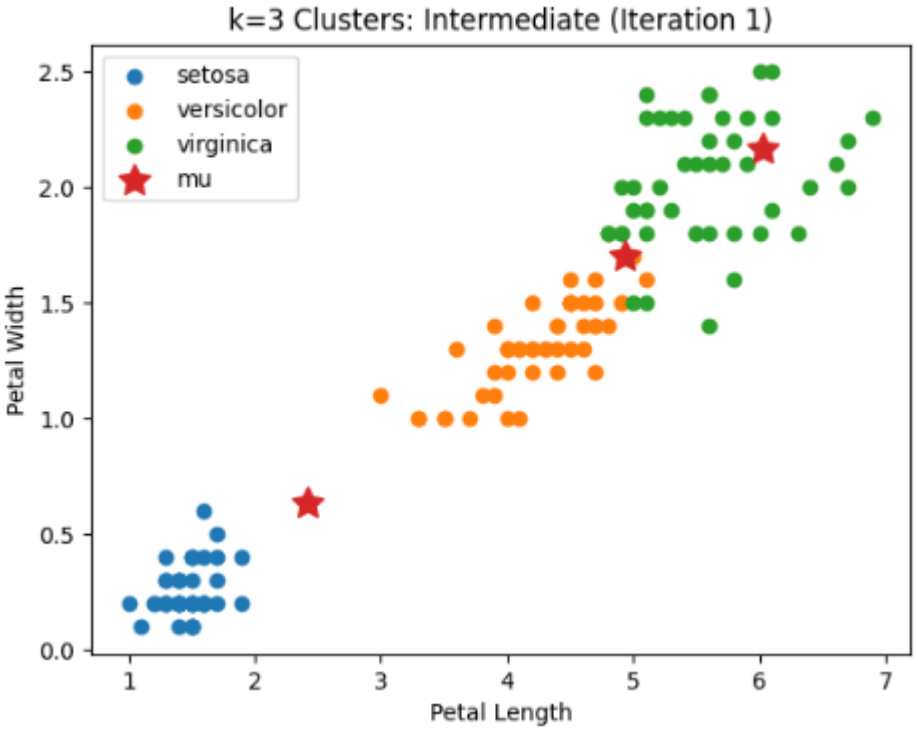


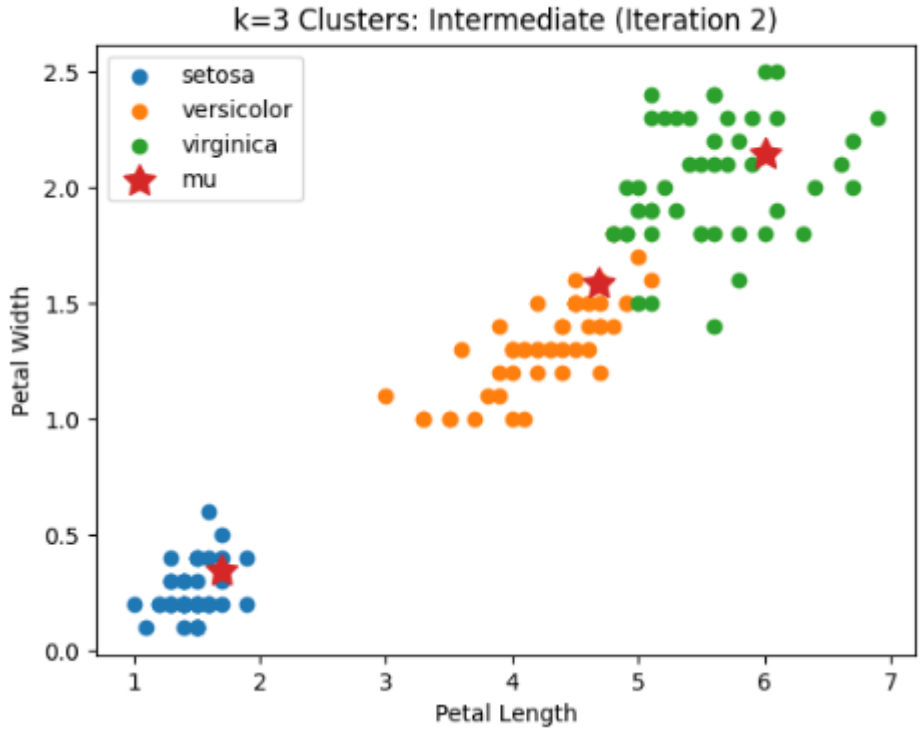


c. For the intermediate graphs I had the program print a new graph per iteration but for the write up I will only include the first two intermediate graphs to avoid the write up being excessively long. The first two intermediate graphs also show the most drastic changes, as can be seen by b.

k=3 clusters:

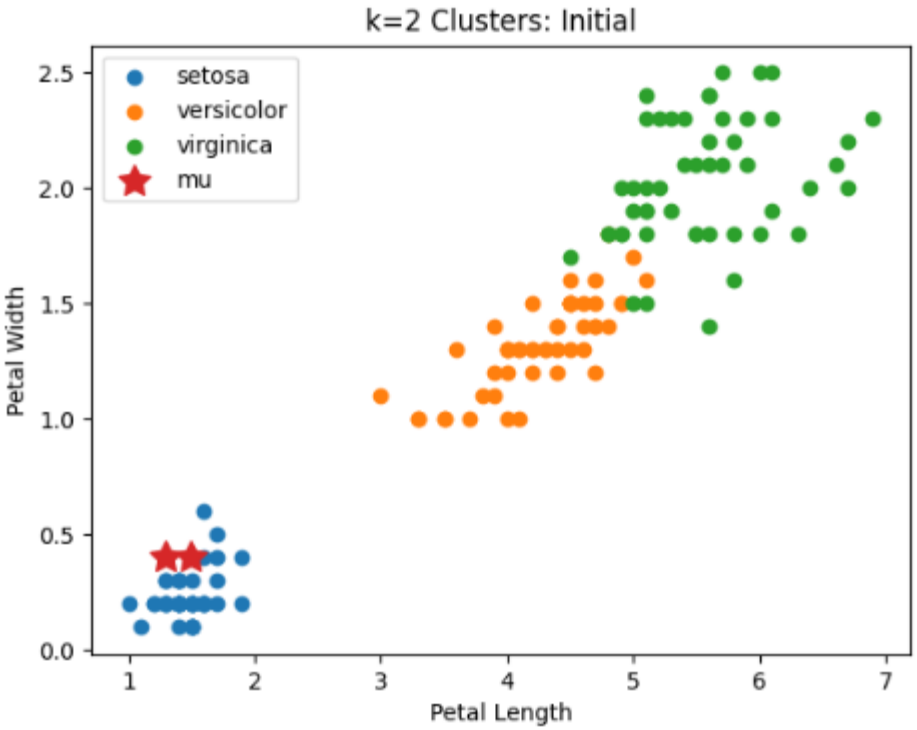

k=3 Clusters: Initial
setosa
versicolor
virginica
mu
Petal Width
Petal Length
0.0
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7

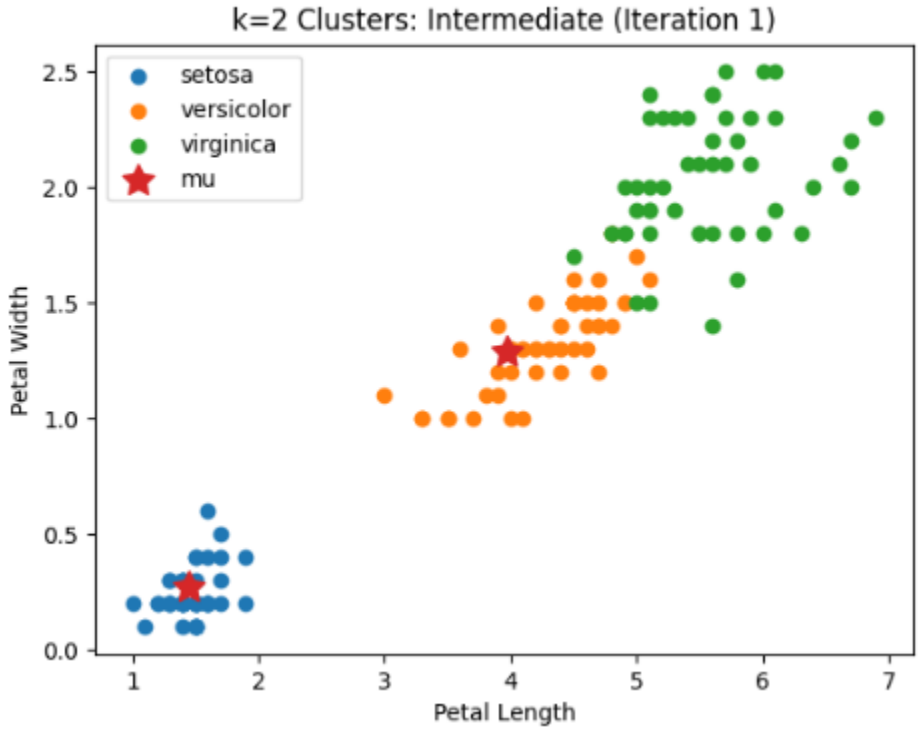

k=3 Clusters: Intermediate (Iteration 1)
setosa
versicolor
virginica
mu
Petal Width
Petal Length
0.0
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7

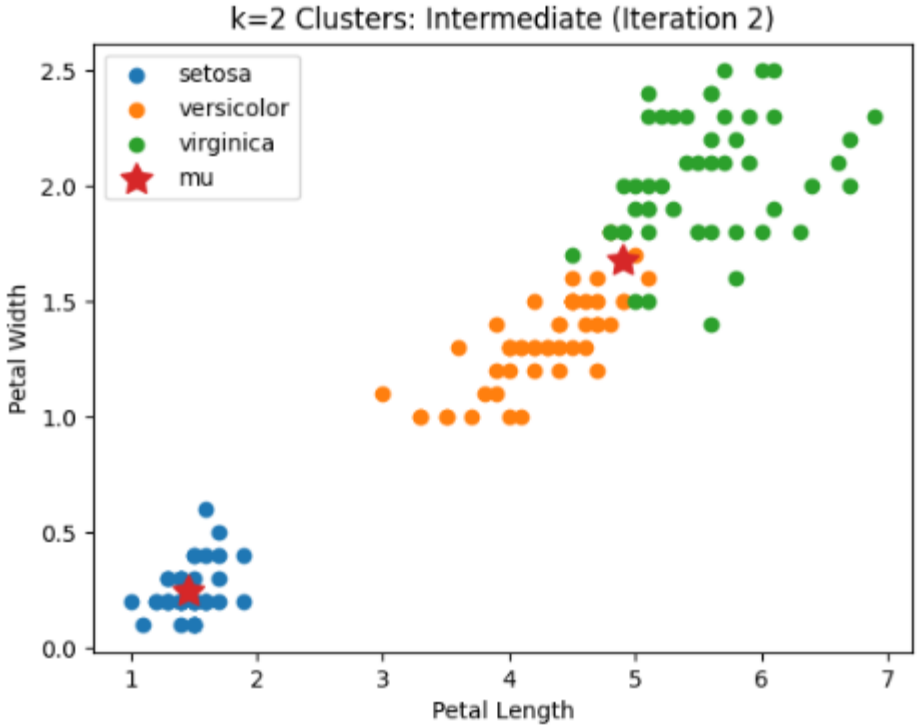

k=3 Clusters: Intermediate (Iteration 2)
setosa
versicolor
virginica
mu
Petal Width
Petal Length
0.0
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7


k=3 Clusters: Converged
setosa
versicolor
virginica
mu
Petal Width
Petal Length
0.0
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7

k=2 clusters

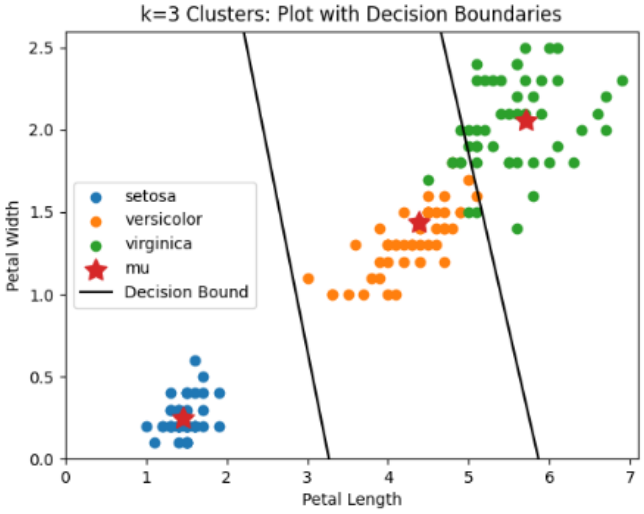

k=2 Clusters: Initial
setosa
versicolor
virginica
mu
Petal Width
Petal Length
0.0
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7

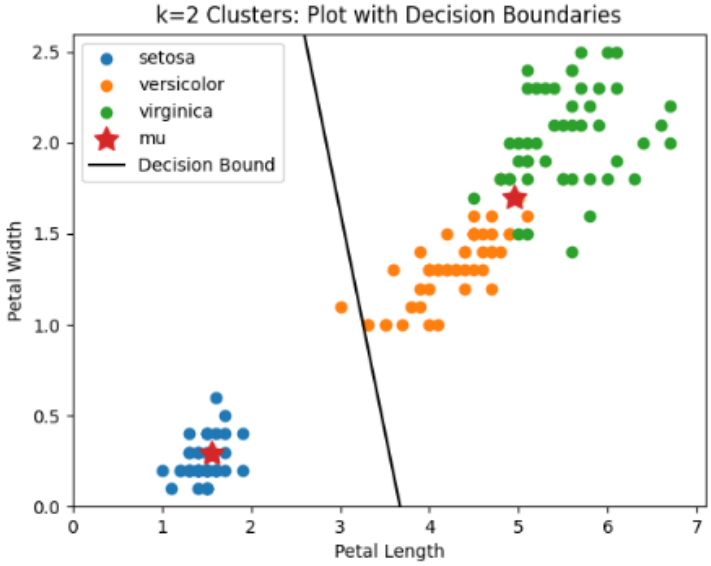

k=2 Clusters: Intermediate (Iteration 1)
setosa
versicolor
virginica
mu
Petal Width
Petal Length
0.0
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7


k=2 Clusters: Intermediate (Iteration 2)
setosa
versicolor
virginica
mu
Petal Width
Petal Length
0.0
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7


k=2 Clusters: Converged
setosa
versicolor
virginica
mu
Petal Width
Petal Length
0.0
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7

d. To plot the decision boundaries for this data set utilizing the optimized parameters, my approach was to plot the decision boundaries when the cluster centers converge. Decision boundaries display a line where a point would switch from one cluster to the other. Since the optimized parameters were all on different points on the x axis, it was able to create decision boundaries that were linear between the middle of the points. To do so, I ordered the means, calculated the midpoints between two of them as well as the slope, and used those values to find the y-intercept. With the slope intercept formula, I plotted the line.





Code:

```
# function to get y for the decision boundaries i.e. perpendicular line formula at midpoint at a given point x
```

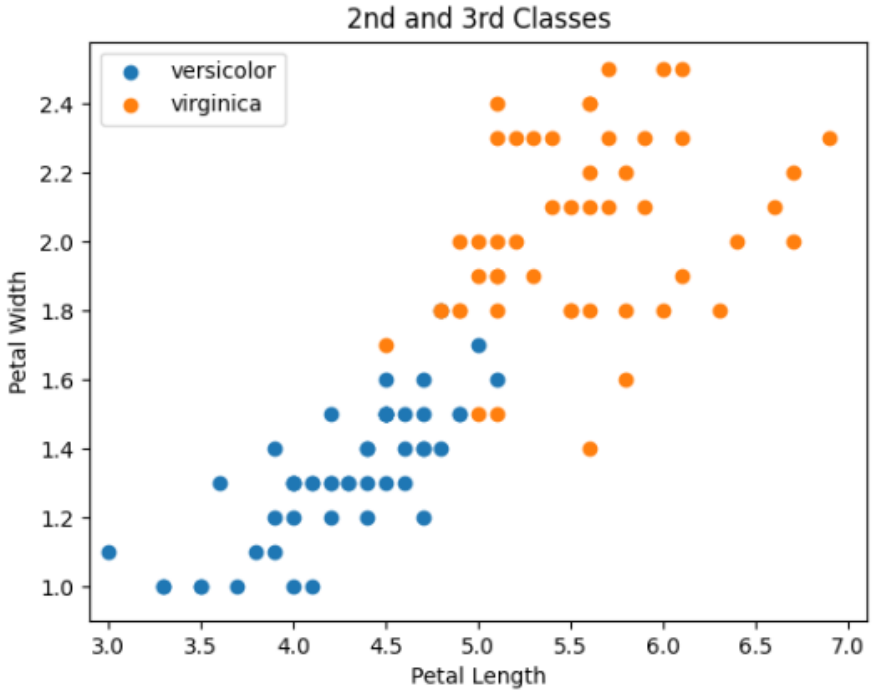
```
def calculate_decision_bounds(pt1_x, pt1_y, pt2_x, pt2_y, x):
    midpt_x = (pt2_x + pt1_x) / 2
    midpt_y = (pt1_y + pt2_y) / 2
    slope = (pt2_y - pt1_y) / (pt2_x - pt1_x)
    perp_slope = -1 / slope
    b = midpt_y - (perp_slope * midpt_x)
    return (perp_slope * x) + b
```

Code inside plotting function:

```
if plot_decision_boundaries:
    # sort the means
    ordered_means = sorted(mus, key=lambda x: x.petal_length, reverse=False)
    for e in ordered_means:
        print(e)
    # keep calculating lines between means while there are means there
    i = 0
    x = np.arange(0, 7, 0.01)
    while i < mus.size - 1:
        y = calculate_decision_bounds(ordered_means[i].petal_length,
ordered_means[i].petal_width,
                                      ordered_means[i + 1].petal_length, ordered_means[i + 1].petal_width, x)
        plt.plot(x, y, color='black', label='Decision Bound' if i == 0 else "")
        i += 1
```

Ex 2.

a.



b. For the sigmoid nonlinearity I ran into an issue of inputs and outputs following an understandable distribution. Initially I normalized the data and resolved the errors but after attending a TA office hour I learned that my weights were the cause of the issue instead.

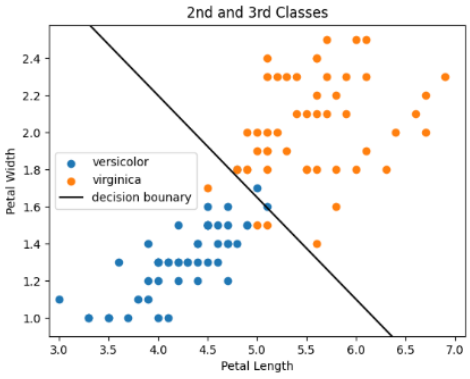
Code:

```
# function to bind weights and inputs together and find sum
def summation_function():
    return np.dot(data, w)


# activation function -- in this case sigmoid function
def activation_function(z):
    return 1. / (1. + np.exp(-z))


# function to get the sigmoid nonlinearity
def get_nonlinearity():
    z = summation_function()
    a = activation_function(z)
    return a
```

c. For c I solved for the w0, w1, and w2 values by choosing two points that seemed to be on the line that would separate the classes. Through doing so, I was able to plot a line using these three values and plot an approximate decision boundary.



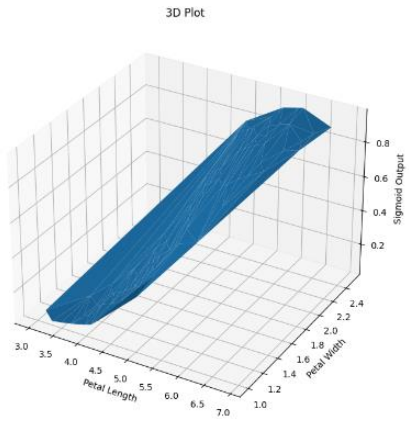
Code:

```
def get_decision_boundary(x1):
    return -((w[1] * x1 + w[0]) / w[2])
```

Code to run:

```
if pt == 'c':
    x_axis = np.arange(2.9, 7.1, 0.1)
    plt.plot(x_axis, get_decision_boundary(x_axis), color='black', label="decision bounary")
```

d.



Code:

```
def plot_3d():
    z = get_nonlinearity()
    out = []
    for zz in z:
        out.append(zz[0])
    z = np.array(out)
    x = data[:, 1]
    y = data[:, 2]
    fig = plt.figure(figsize=(8, 8))
    ax = plt.axes(projection='3d')
    # Creating plot
    ax.plot_trisurf(x, y, z)
    # show plot
    ax.set_title('3D Plot')
    ax.set_xlabel('Petal Length')
    ax.set_ylabel('Petal Width')
    ax.set_zlabel('Sigmoid Output')
    plt.show()
```

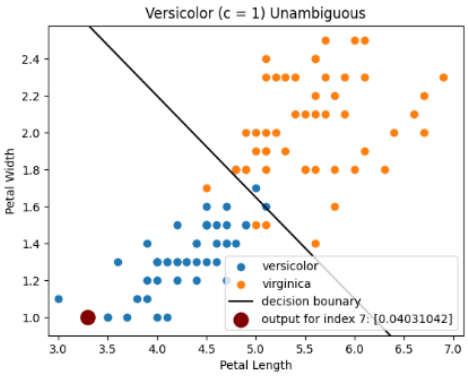
e. It can be seen that ambiguous points give outputs near 0.5 but unambiguous points give values near 0 if versicolor (c1) or near 1 if virginica (c2).

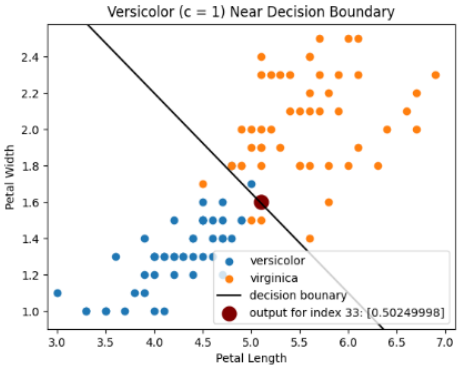
code:

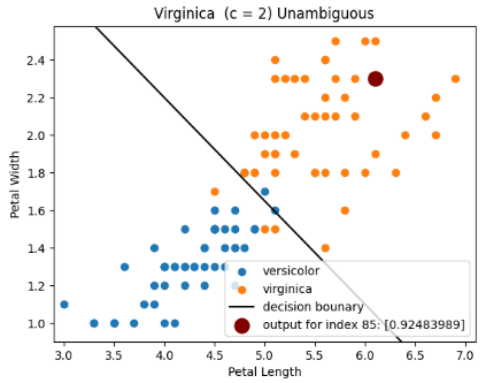
```
def compare_data(dataset, outputs, index):
    plt.scatter(dataset[index][1], dataset[index][2], 150, color='maroon', label=f'output for index
{index}: {outputs[index]}')
 is {outputs[index]}')
```

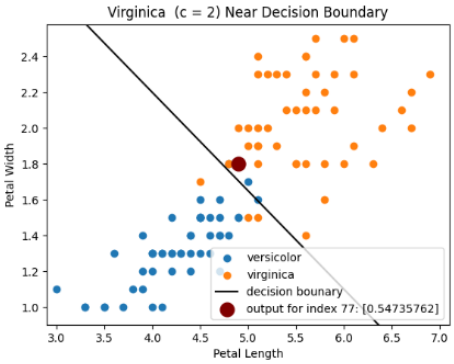
code to run:

```
def run_pt_e():
    plot_classes("Versicolor (c = 1) Unambiguous", 'e', 7)
    plot_classes("Versicolor (c = 1) Near Decision Boundary", 'e', 33)
    plot_classes("Virginica  (c = 2) Unambiguous", 'e', 85)
    plot_classes("Virginica  (c = 2) Near Decision Boundary", 'e', 77)
```









Ex 3.

a.

code:

```
# get the mean-squared error
def get_mean_squared(data_vectors, weights, pattern_classes):
    global data, w
    data = data_vectors
    w = weights
    summation = 0
    y_actual = get_nonlinearity()
    count = 0
    while count < 100:
        summation += (y_actual[count] - pattern_classes[count]) ** 2
        count += 1
    return summation / 2
```

b.

low error:

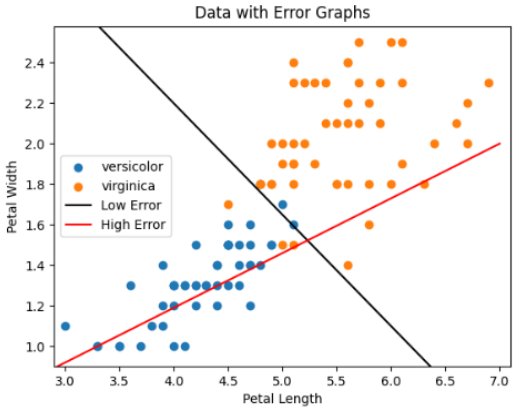
w0 = -8.8, w1 = 1.1, w2 = 2

[3.62340432]

high error:

w0 = 2 w1 = 5 w2 = -18.5

[27.43531901]



Code:

```
def compute_for_weights():
    print("low error:")
    print("w0 = -8.8, w1 = 1.1, w2 = 2")
    w0 = -8.8
    w1 = 1.1
    w2 = 2
    print(get_mean_squared(data, np.array([[w0], [w1], [w2]]), expected_y))
    print("high error:")
    print("w0 = 2 w1 = 5 w2 = -18.5")
    w0b = 2
    w1b = 5
    w2b = -18.5
    print(get_mean_squared(data, np.array([[w0b], [w1b], [w2b]]),
expected_y))
    plot_classes("Data with Error Graphs", w0, w1, w2, w0b, w1b, w2b)

initialize_params()
compute_for_weights()
```

A1 P2 Ex3

3c Find the gradient of the objective function w/ respect to the weights.

objective function: $E = \frac{1}{2}\sum_n \sum_k (A - y_{pred})^2$

where

A = sigmoid function = $\frac{1}{1+\exp(-z)}$

Z = binding function = XW^T

so to take the gradient we can ignore $\frac{d}{dw}(y_{pred}) = 0$

so

$$\frac{dE}{dw} = \frac{dE}{dA}\frac{dA}{dZ}\frac{dZ}{dw}$$

we must derive E with respect to A & A with respect to Z in order to get it in terms of weights

Using the chain rule & derivative of A we get

$$\boxed{\frac{dE}{dw} = \sum_n (A)(1-A)(A - y_{pred})X_n}$$

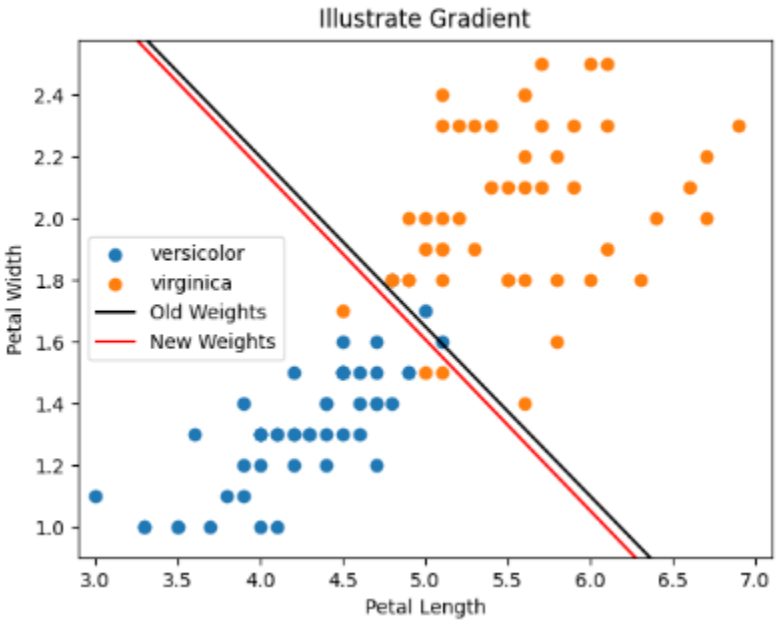
3d To represent the gradient in scalar:

$$\frac{dE}{dw} = \sum_n \sum_i (a_i)(1-a_i)(a_i - y_{i\,pred})X_{ni}$$

To represent the gradient in vector:

$$\frac{dE}{dw} = \sum_n (A)(1-A)(A - y_{pred})X_n$$

e.



Code:

```
# function to get the summed gradient for an ensemble of patterns
def get_gradient(data_vectors, pattern_class):
    a = get_nonlinearity()
    y_exp = pattern_class
    ones = np.ones(100)
    unsummed_gradient = []
    count = 0
    while count < 100:
        unsummed_gradient.append(data_vectors[count] * a[count] *
(ones[count] - a[count]) * (a[count] - y_exp[count]))
        count += 1
    gradient = np.sum(np.array(unsummed_gradient), axis=0)
    return gradient


# function to update the weights based on the gradient
def update_weights(data_vectors, weights, pattern_class, epsilon):
    dw = get_gradient(data_vectors, pattern_class)
    dw = np.array([[dw[0]], [dw[1]], [dw[2]]])
    return weights - dw * epsilon


# function to compare the updated weights and previous weights
def plot_gradient():
    w_old = np.array([[-8.8], [1.1], [2]])
    w_new = update_weights(data, w_old, expected_y, 0.005)
    plot_classes("Illustrate Gradient", w_old[0], w_old[1], w_old[2],
w_new[0], w_new[1], w_new[2], "Old Weights", "New Weights")

initialize_params()
```

```
# compute_for_weights()
plot_gradient()
```

We can see that the gradient is working because the means error decreased from [3.62340432] to [3.58775676] after utilizing the new means calculated from the gradient displayed on the graph.

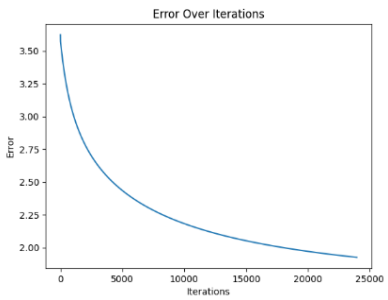
Ex 4.

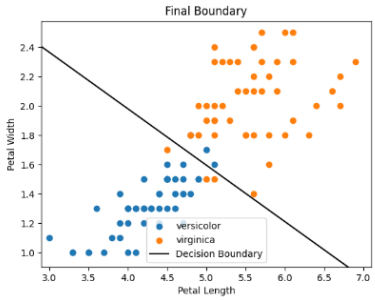
a.

Code:

```
# function to optimize the decision boundary through gradient decent
def optimize_decision_boundary(initial_weights, epsilon):
    global w
    w = initial_weights
    errs = []
    prev_diff = 10000000000
    curr_diff = get_mean_squared(data, initial_weights, expected_y)
    errs.append(curr_diff)
    iterations = 1
    print(curr_diff)
    while prev_diff - curr_diff > 0.00001:
        prev_diff = curr_diff
        w_new = update_weights(data, w, expected_y, epsilon)
        # get_mean_squared updates global w
        curr_diff = get_mean_squared(data, w_new, expected_y)
        errs.append(curr_diff)
        iterations += 1
        print(curr_diff)
    plot_classes("Final Boundary", w[0], w[1], w[2], "Decision Boundary")
    plot_iterations(np.arange(iterations), np.array(errs))
```

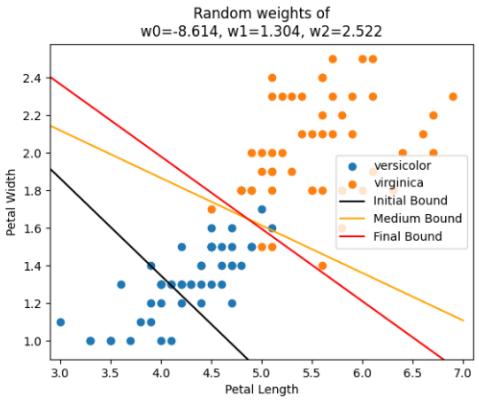
b.

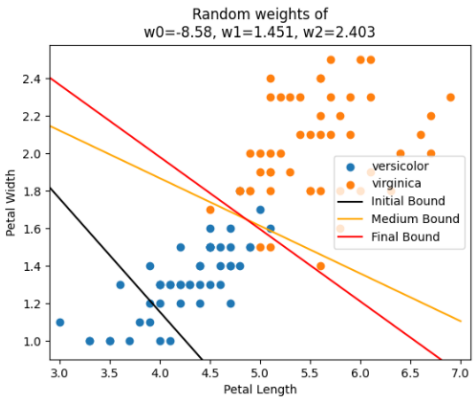




c. For this part of the problem, I ran into a consistent error of having my initial bound not appear on the graph. In these cases, where I used true random numbers the errors still landed on the ideal point but I could not plot the initial graph. To avoid this, I added a random number to a seed of the weights I had been using previously in other parts in order to make sure it would

appear on the graph. This way the weights themselves were still randomized but would be constricted.





Code:

```
# function to optimize the decision boundary through gradient decent
def optimize_decision_boundary(initial_weights, epsilon):
    errs = []
    middle_bound_weight = []
    prev_diff = 10000000000
    curr_diff = get_mean_squared(data, initial_weights, expected_y)
    errs.append(curr_diff)
    iterations = 1
    print(curr_diff)
    while prev_diff - curr_diff > 0.00001:
        prev_diff = curr_diff
        w_new = update_weights(data, w, expected_y, epsilon)
        # get_mean_squared updates global w
        curr_diff = get_mean_squared(data, w_new, expected_y)
        errs.append(curr_diff)
        iterations += 1
        if iterations == 6000:
            middle_bound_weight = w
        print(curr_diff)
    plot_all_bounds(f'Random weights of\nw0={round(initial_weights[0][0],
3)}, w1={round(initial_weights[1][0], 3)}, w2={round(initial_weights[2][0],
3)}', initial_weights, middle_bound_weight, w)
    plot_classes("Final Boundary", w[0], w[1], w[2], "Decision Boundary")
    plot_iterations(np.arange(iterations), np.array(errs))


# function to get somewhat random numbers as weights
def get_pseudo_random_weights():
    w0 = -8.8
    w1 = 1.1
    w2 = 2
    w0 += random.random()
    w1 += random.random()
    w2 += random.random()
    return np.array([w0, w1, w2])

initialize_params()
```

```
# plot_gradient()
arr = get_pseudo_random_weights()
a = arr[0]
b = arr[1]
c = arr[2]
optimize_decision_boundary(np.array([[a], [b], [c]]), 0.005)
```

d. To choose the gradient step size, I tested a few different values. I started testing with 0.5, learned that the function would often skip over the minimum with this value and then began continuing to decrease the step size. Through trial and error I found that 0.005 worked well without being too small.

e. To choose the stopping criteria, I initially set it to when the function completely converged, or the difference between the previous error and the current error is 0. However, I learned that due to the nature of the function, this will never happen or will eventually happen due to computer rounding but will take a very long time. Instead, I used the stopping criteria of the difference in error is > 0.00001. I found that this value allowed my function to get to a point where it nearly converged without completely converging.