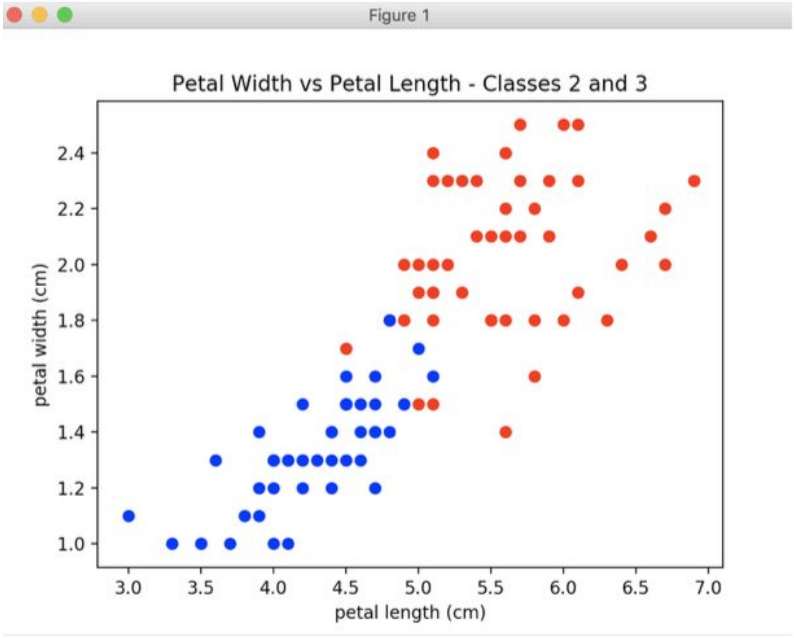
CSDS391 P2
Marcus Lin
Alec Peltekian

Note: Since different problems may utilize similar functions in this project, we add if conditions in the code to specify additional problems some functions are also addressing with.

1)
A) We obtained the dataset for virginica and versicolor species from the excel and made a scatter plot against the petal length and petal width. The virginica species are colored red and the versicolor species are colored blue.

```
def problem_1_a(problem_letter, x_coords_line, y_coords_line):
    colors = ['red', 'blue']
    species = ['setosa', 'virginica', 'versicolor']
    for i in range(1, 3):
        species_df = data[data['species'] == species[i]]
        plt.scatter(species_df['petal_length'], species_df['petal_width'], color=colors[i - 1])

    plt.xlim(0, 10)
    plt.ylim(0, 6)
    plt.xlabel('petal length (cm)')
    plt.ylabel('petal width (cm)')
    plt.title('Petal Width vs Petal Length - Problem #1(A)')
```

The output of this scatter plot is below.



B)
Our function is defined as
sigmoid(x_test[i][0] * weight_one + x_test[i][1] * weight_two + bias)
where x_test[i][0] represents the ith petal length and x_test[i][1] represents the ith petal length. We chose weight_one = 0.32, weight_two = 3.70 and bias = -7.61 as the coefficients of the function, and it consistently got above 90s of the test data correctly. The following code shows the function that defines problem 1(B)

```
for i in range(size):
    if (y_test[i] == 'versicolor'):
        actual_class = 0

    else:
        actual_class = 1

    z = x_test[i][0] * weight_one + x_test[i][1] * weight_two + bias
    sigmoid_result = sigmoid(z)
    if (sigmoid_result < 0.5):
        predicted_class = 0
    else:
        predicted_class = 1

    print("actual class: ", actual_class)
    print("predicted class: ", predicted_class)
    if (actual_class != predicted_class):
        num_wrong += 1
    else:
        num_right += 1
    print("--------")
```

The following is a part of the result after passing the dataset to the one-layer neural network, which will be displayed in the terminal after you run the program.

```
.
--------
actual class:  0
predicted class:  0
--------
actual class:  0
predicted class:  0
--------
part 1b-amount correct:  0.95
```

C) For this part, we needed to plot the linear classifying line. The code that does this is below and calls problem_1_a() in this code because it has the code that plots the scatterplot of iris data. Since the decision boundary is $y = m^T x + b = 0$

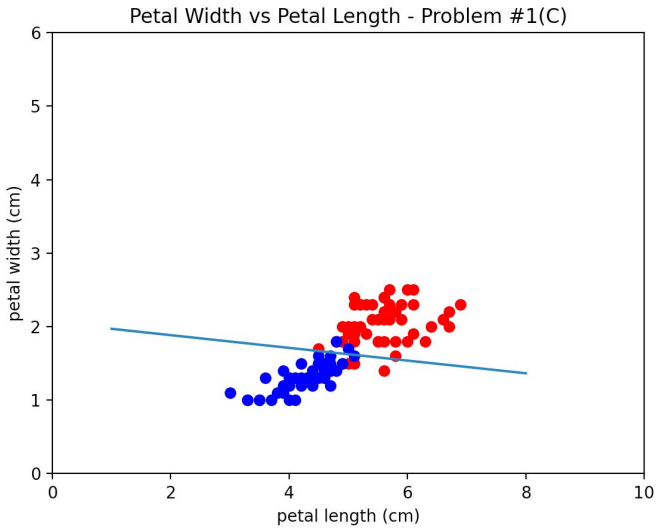
We can get

$$\begin{aligned} m_1 x_1 + m_2 x_2 &= -b \\ \Rightarrow x_2 &= -\frac{m_1 x_1 + b}{m_2} \end{aligned}$$

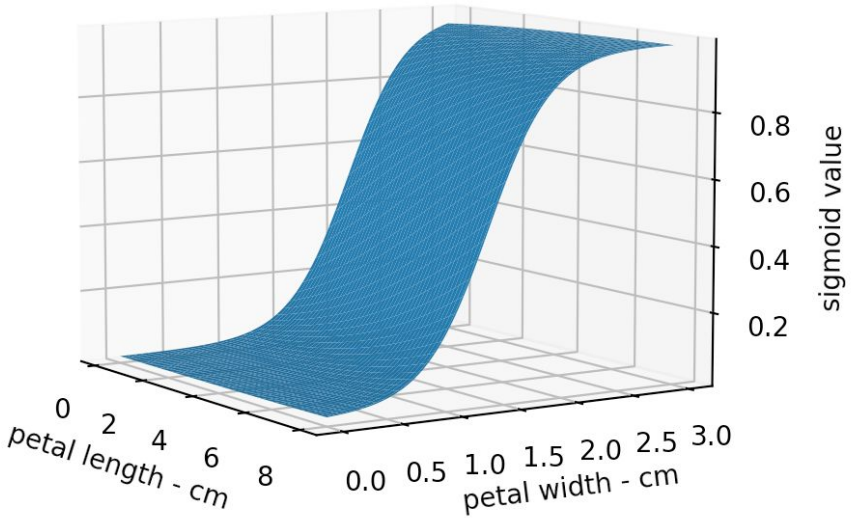
According using this equation, we obtained the x and y coordinates in the get_line_coords() function

```
def problem_1_c(weight_one, weight_two, bias):
    line_coords = get_line_coords(weight_one, weight_two, bias)
    line_x_coords = line_coords[0]
    line_y_coords = line_coords[1]

    problem_1_a('c', line_x_coords, line_y_coords)
```

The line overlaid on the scatter plot of Iris data is shown below.



D) To draw the 3D plot, we passed three inputs to the surface plotting function: a list of petal length values, a list of petal width values, and a list of sigmoid values for the corresponding petal lengths and petal widths. We are using the same weight as 1b. The plot is shown below.



E) For this part, we manually gave the classifier test data that was near the boundary line. We then gave the classifier test data that was far from the boundary line. As expected, the classifier performed really well when the data was far from the boundary line. It only performed okay when the data was close to the boundary line.

- Two datasets we chose:

```
if problem_letter == 'e':
    x_test = [[5.1, 2], [5, 1.9], [5.1, 1.9], [4.8, 1.8], [5.1, 1.8]]
    y_test = ['virginica', 'viginica', 'versicolor', 'versicolor', 'virginica']
```

```
if problem_letter == 'ee':
    x_test = [[6.6, 2.1], [6.3, 1.8], [6.1, 2.5], [6.7, 2.2], [4, 1.3], [4.6, 1.5], [3.3, 1], [3.9, 1.4]]
    y_test = ['virginica', 'virginica', 'virginica', 'virginica', 'versicolor', 'versicolor',
              'versicolor', 'versicolor']
```

- Result:

```
--------
actual class:  1
predicted class:  1
--------
actual class:  0
predicted class:  1
--------
part 1e-amount correct when looking near the decision boundary:  0.6

actual class:  1
predicted class:  1
--------
actual class:  1
predicted class:  1
--------
actual class:  1
predicted class:  1
--------
part 1e-amount correct when looking far from the decision boundary:  1.0
```

2)

A) For this part, we needed to calculate the mean squared error. To do this, we computed the squared error for each different data point and averaged this by the number of data points. More specifically, We use $\sigma(z) = \frac{1}{1+e^{-z}}$ to represent the sigmoid function.
The logistic regression hypothesis is defined as

$$h_{\boldsymbol{w}}(\boldsymbol{x}) = \sigma(\boldsymbol{w} \cdot \boldsymbol{x}) = \frac{1}{1+e^{-\boldsymbol{w}\cdot\boldsymbol{x}}}$$

where **w** · **x** = bias + weight_one * petal_length + weight_two * petal_width.
So,

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(h_{\boldsymbol{w}}(\boldsymbol{x_n}) - c_n)^2$$

where c_n indicates the nth data pattern with 0 representing versicolor class and 1 representing virginica class. **N** is the number of total trials.

$$\boldsymbol{x_n} = \begin{pmatrix} x_{n,1} \\ x_{n,2} \\ x_{n,3} \end{pmatrix} = \begin{pmatrix} 1 \\ pedal\ length\ of\ nth\ data \\ pedal\ width\ of\ nth\ data \end{pmatrix} \qquad \boldsymbol{w} = \begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} bias \\ weight\_one \\ weight\_two \end{pmatrix}$$

The code is shown below. We passed more arguments to this function instead of three, since we split the parameters defining the neural network as weight_one, weight_two and bias. Also, it now is used in #3 as well, which is why we also added the learning_rate as a parameter to the function and extended it to return the updated weights and bias as well as the computed mean squared error. We asked professor Luwicki and he said it was acceptable. The code for this function is below:

```
# computes the mean squared error and does gradient descent. Dataset is the values of petal
# length and petal width. Patterns is the values of the class of each datapoint
def mean_squared_error(dataset, patterns, weight_one, weight_two, bias, learning_rate):
    squared_error_running_total = 0.0
    weight_one_update = 0
    weight_two_update = 0
    bias_update = 0

    size = len(dataset)
    for i in range(size):
        z = (dataset[i][0] * weight_one) + (dataset[i][1] * weight_two) + bias
        sigmoid_result = sigmoid(z)
        derivative_result = derivativeSigmoid(sigmoid_result)
        if patterns[i] == 'versicolor':
            actual_class = 0
        else:
            actual_class = 1

        err_result = (sigmoid_result - actual_class)
        squared_error_result = err_result ** 2
        weight_one_update += (err_result * derivative_result) * dataset[i][0]
        weight_two_update += (err_result * derivative_result) * dataset[i][1]
        bias_update += (err_result * derivative_result)
        squared_error_running_total += squared_error_result
```

```
# computes new weights based on the updated weights computed in the above loop
new_weight_one = weight_one - learning_rate * weight_one_update
new_weight_two = weight_two - learning_rate * weight_two_update
new_bias = bias - learning_rate * bias_update

# computes the mean squared error
squared_error_mean = (squared_error_running_total / size)
# returns all of the above computed values
return_array = [squared_error_mean, new_weight_one, new_weight_two, new_bias]
return return_array
```

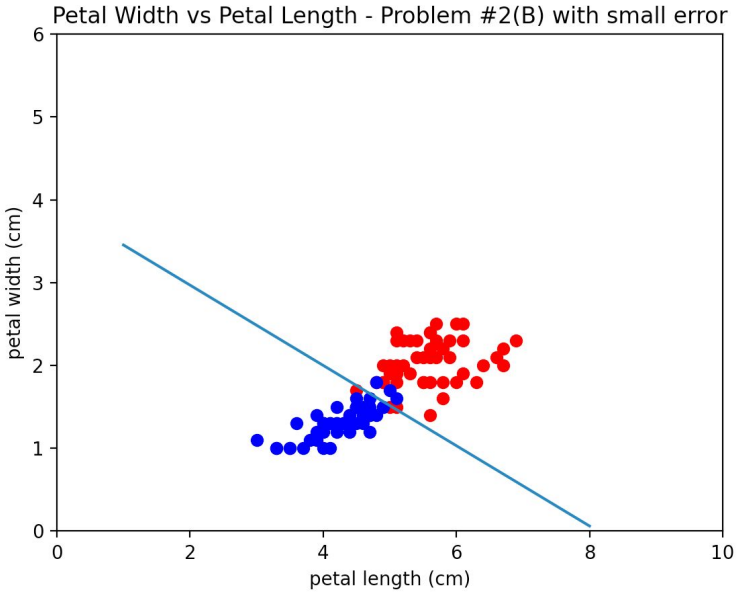
B) We wanted to calculate the mean squared error on the entire dataset for a “good” set of weights and bias and a “bad” set of weights and bias. We tested out different manual values of the weights and bias and came up with the following code that demonstrates “good” and “bad” values.

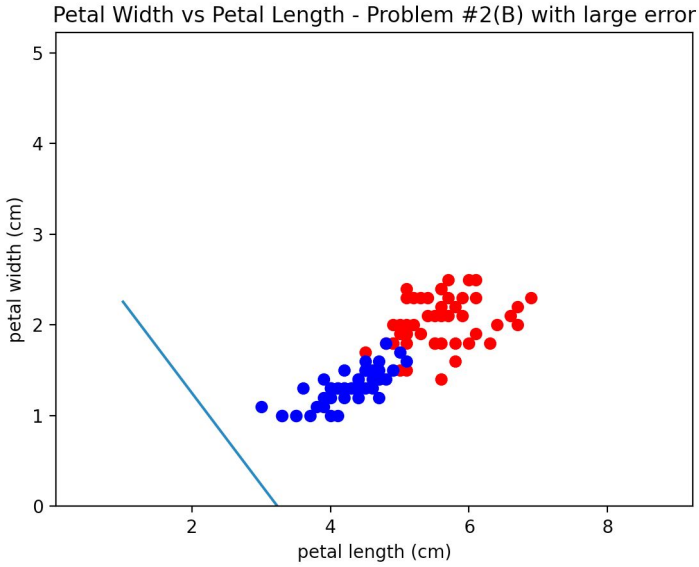
```
def problem_2_b():
    dataset = data.iloc[50:, [2, 3]].values
    patterns = data.iloc[50:, 4].values
    good_parameters = [.48, .99, -3.9]
    bad_parameters = [.99, .98, -3.2]
    good_mean_squared = mean_squared_error(dataset, patterns, good_parameters[0], good_parameters[1], good_parameters[2]
                                           , .001)
    bad_mean_squared = mean_squared_error(dataset, patterns, bad_parameters[0], bad_parameters[1], bad_parameters[2],
                                          .001)

    print("Good parameters mean squared error: ", good_mean_squared[0])
    print("Bad parameters mean squared error: ", bad_mean_squared[0])
```

The output is below. The first value is the low mean squared error, and the second value is the higher mean squared error

```
Good parameters mean squared error:  0.1297405497529293
Bad parameters mean squared error:  0.4041877935535348
```

Both boundaries are plotted shown below.





(C) To compute the gradient of the objective function. As obtained in 2(A), we have

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n}) - c_n)^2$$

According to the property of the logistic function, its derivative is

$$\sigma'(z) = \sigma(z)(1-\sigma(z))$$

Base on the derivative function we just derived, we can firstly get $\sigma'(w \cdot x_n)$ to get ΔMSE

$$\frac{\partial(\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n}))}{\partial \boldsymbol{w}} = \sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big(1-\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big)\frac{\partial(\boldsymbol{w}\cdot\boldsymbol{x_n})}{\partial \boldsymbol{w}} = \sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big(1-\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big)\cdot\boldsymbol{x_n}$$

Then, we can get the gradient

$$\Delta MSE = \frac{\partial MSE}{\partial \boldsymbol{w}} = \frac{2}{N}\sum_{n=1}^{N}(\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n}) - c_n)\cdot\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big(1-\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big)\cdot\boldsymbol{x_n}$$

(D)

For **vector form**:

Since

$$\frac{\partial(\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n}))}{\partial \boldsymbol{w}} = \sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big(1-\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big)\frac{\partial(\boldsymbol{w}\cdot\boldsymbol{x_n})}{\partial \boldsymbol{w}} = \sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big(1-\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big)\cdot\boldsymbol{x_n}$$

$$\Delta MSE = \frac{\partial MSE}{\partial \boldsymbol{w}} = \frac{2}{N}\sum_{n=1}^{N}(\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n}) - c_n)\cdot\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big(1-\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big)\cdot\boldsymbol{x_n}$$

For **Scalar form:**

$$\frac{\partial(\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n}))}{\partial w_i} = \sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big(1-\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big)\frac{\partial(\boldsymbol{w}\cdot\boldsymbol{x_n})}{\partial w_i} = \sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big(1-\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big)\cdot x_{n,i}$$

$$\Delta MSE = \frac{\partial MSE}{\partial w_i} = \frac{2}{N}\sum_{n=1}^{N}(\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n}) - c_n)\cdot\frac{\partial(\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n}))}{\partial w_i}$$

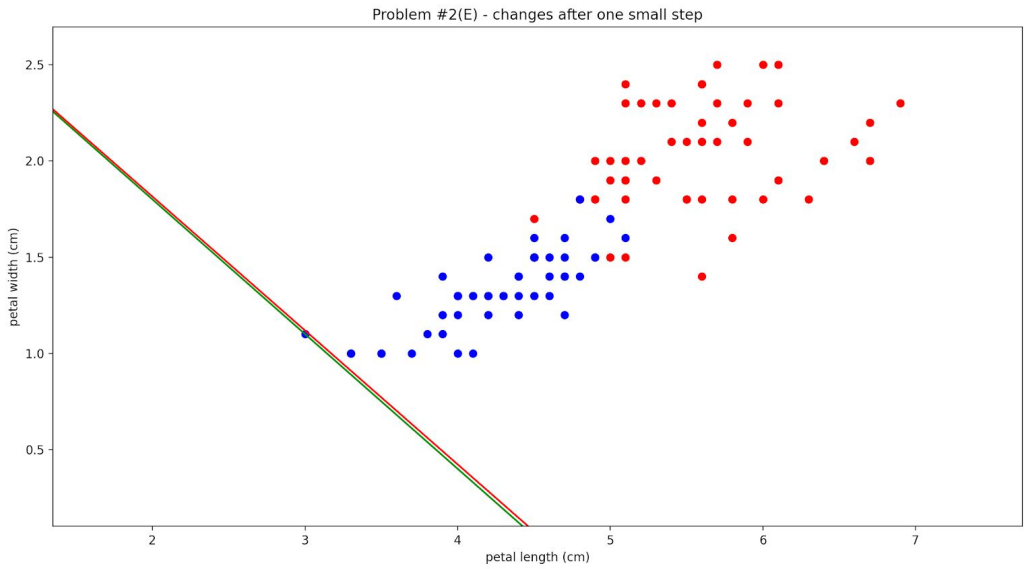
$$= \frac{2}{N}\sum_{n=1}^{N}(\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n}) - c_n)\cdot\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big(1-\sigma(\boldsymbol{w}\cdot\boldsymbol{x_n})\big)\cdot x_{n,i}$$

E) As mentioned above, in order to reduce repetition, our mean_squared_error() function also iterates through all the data and computes the summed gradient according to the function in 2C and 2D. Then, it computes and returns the MSE and the updated new weight coefficients. The code is shown below.

```
def mean_squared_error(dataset, patterns, weight_one, weight_two, bias, learning_rate):
    squared_error_running_total = 0.0
    weight_one_update = 0
    weight_two_update = 0
    bias_update = 0

    size = len(dataset)
    for i in range(size):
        z = (dataset[i][0] * weight_one) + (dataset[i][1] * weight_two) + bias
        sigmoid_result = sigmoid(z)
        derivative_result = derivativeSigmoid(sigmoid_result)
        if patterns[i] == 'versicolor':
            actual_class = 0
        else:
            actual_class = 1

        err_result = (sigmoid_result - actual_class)
        squared_error_result = err_result ** 2
        bias_update += (err_result * derivative_result)
        weight_one_update += (err_result * derivative_result) * dataset[i][0]
        weight_two_update += (err_result * derivative_result) * dataset[i][1]
        squared_error_running_total += squared_error_result

    # computes new weights based on the updated weights computed in the above loop
    new_weight_one = weight_one - learning_rate * (weight_one_update * 2 / size)
    new_weight_two = weight_two - learning_rate * (weight_two_update * 2 / size)
    new_bias = bias - learning_rate * (bias_update * 2 / size)

    # computes the mean squared error
    squared_error_mean = (squared_error_running_total / size)
    # returns all of the above computed values
    return_array = [squared_error_mean, new_weight_one, new_weight_two, new_bias]
    return return_array
```

We wrote a function which shows the decision boundary being recalculated after one iteration using gradient descent by calling the mean_squared_error() function with step size 0.01.
Our output and plot is shown below. The output shows the MSE is slightly reduced and the weights are also changed slightly. And two boundaries are drawn in the same plot. To make it clear, the red one is the boundary after the change. We can see it is approaching in the correct direction. When you run our program, you can enlarge the plot to see more clearly.

```
----------Problem_2e----------------
The MSE before the step:  0.28222177295162443
The weights for the step: [0.7, 1, -3.2]
The MSE after the step:  0.27899569256990414
The weights after the step : [0.6947125719033699, 0.9983741893729534, -3.2012633435455005]
----------------------------------
```



3)

A/B/C)

Parts A, B, and C all went together so we made a function that reflects this and will also talk about them together. We used the mean_squared_error function from part 2(E). We returned the updated weight one, weight two, and bias from this function and used these to feed back into itself to do iterations. The code for this is a little too long to be entirely put in the report, but the function for this is "problem_3_a_b_c()", and we show partial code below.

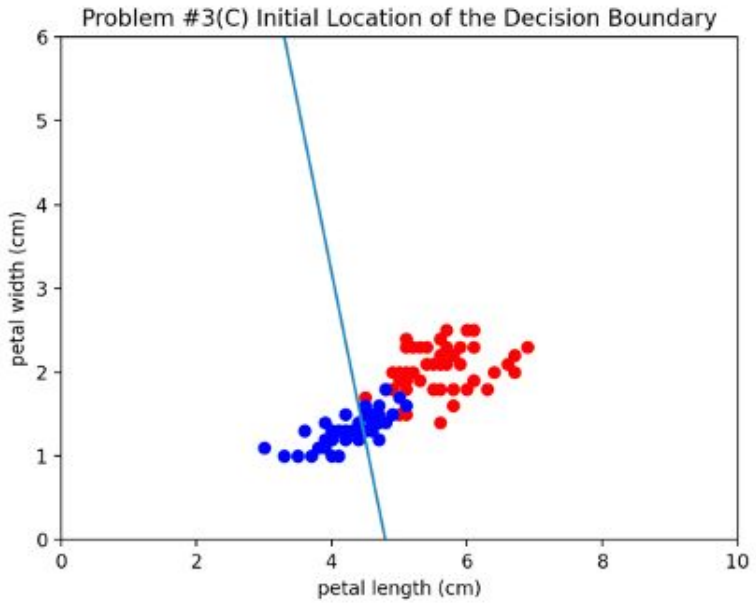
```
for i in range(20000):
    model_result = mean_squared_error(dataset, patterns, weight_one, weight_two, bias, 0.01)
    weight_one = model_result[1]
    weight_two = model_result[2]
    bias = model_result[3]
    mean_squared_error_list.append(model_result[0])
    x_value_list.append(i)

    if model_result[0] < .1:
```

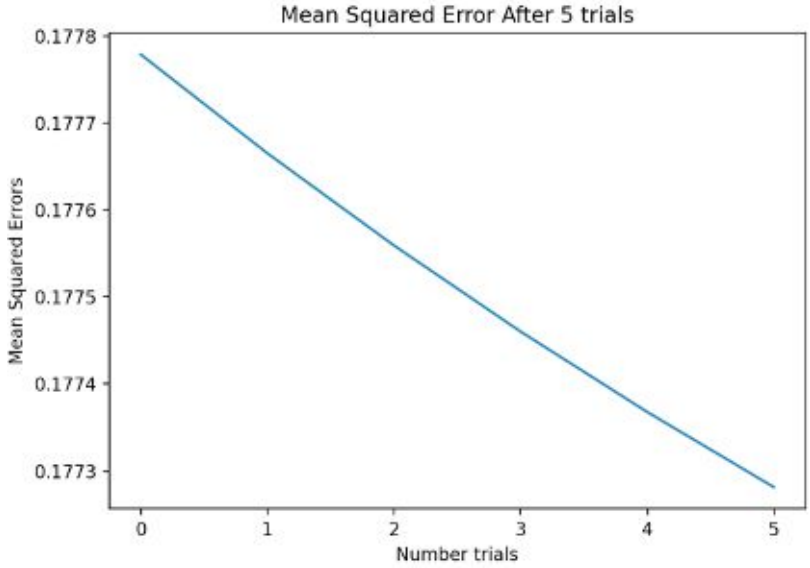
For part B, since the mean_squared_error function returns all three weight coefficients at each iteration, we can draw the decision boundary whenever we stop. Also, we are appending the MSE(model_result[0] in the code) to the list as the iteration goes, so we can also draw the learning curve whenever we stop.

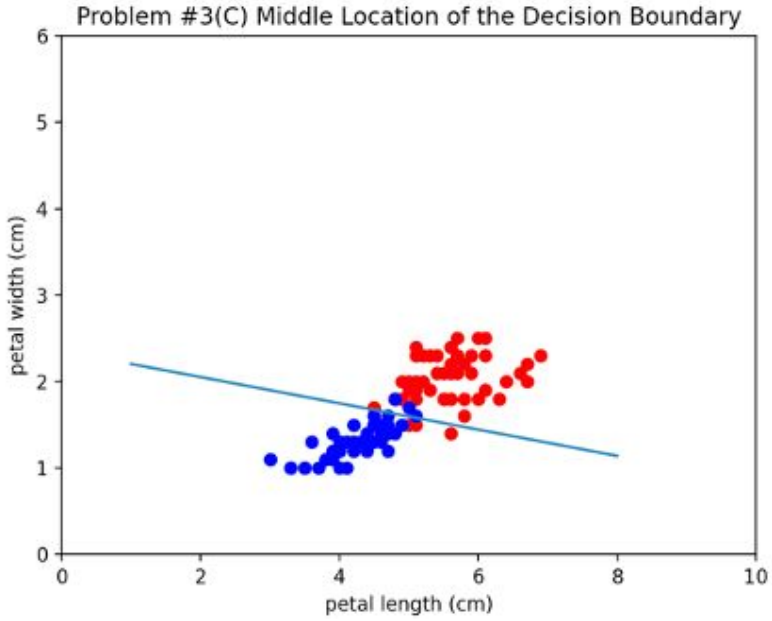
For part C, this is how we randomly initialize our coefficients to make it be shown in the plot.

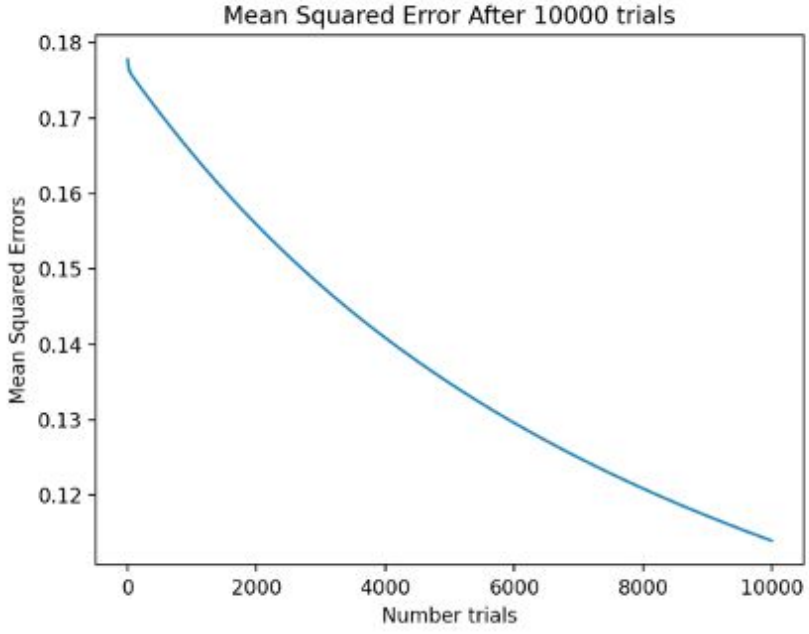
```
weight_one = random.random()
weight_two = random.random()
bias = -random.random()
bias_multiplier = randint(1, 4)
bias = bias * bias_multiplier
```

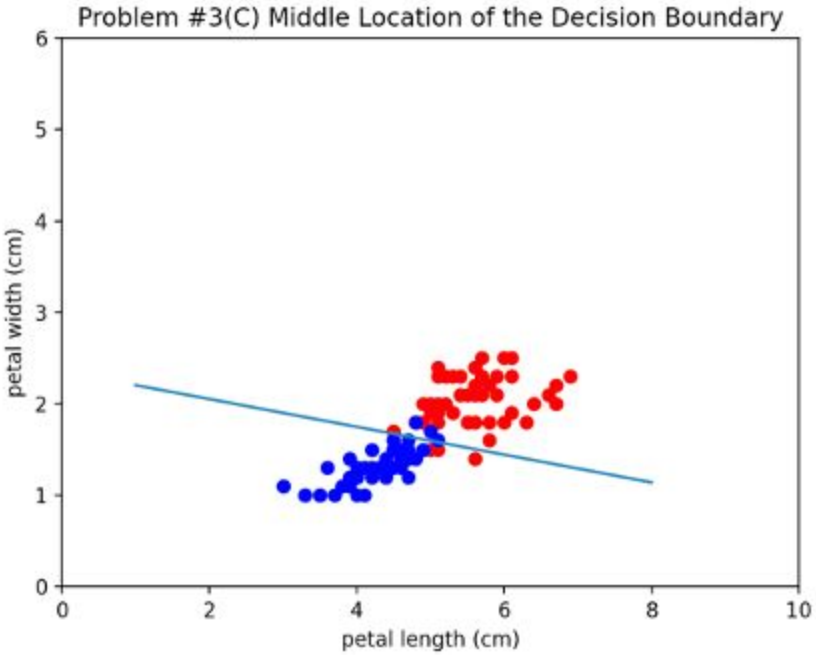
We have included plots of the number of trials versus the mean squared error as well as the decision boundary being plotted by the weights and bias. We did plots for the initial, middle, and final locations of the decision boundary.

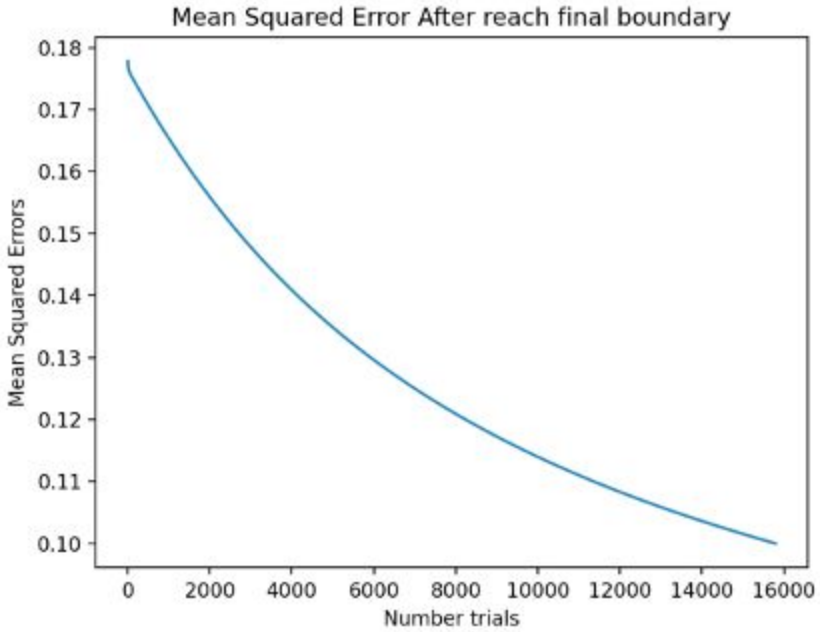





Problem #3(C) Middle Location of the Decision Boundary
petal width (cm)
petal length (cm)
0
1
2
3
4
5
6
0
2
4
6
8
10


Mean Squared Error After 10000 trials
Mean Squared Errors
Number trials
0.12
0.13
0.14
0.15
0.16
0.17
0.18
0
2000
4000
6000
8000
10000





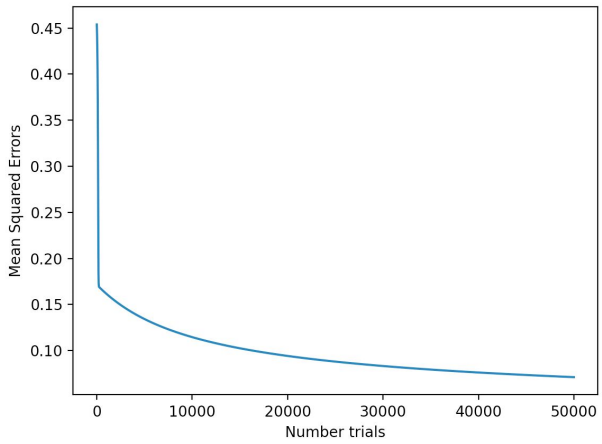
D)

We chose the gradient step size by a lot of trial and error. When we were testing our project, we experimented with different step sizes and ended up settling on one that works most of the time, which is 0.01. The algorithm will not converge and jump around the minimum if the step size is too large, and the convergence will take place pretty slowly when the step size is too small. With

step size of 0.01, the program is also approaching the correct answer with a reasonable amount of time on our machine.

E)

We chose the stopping criterion by doing trials and observations. Basically, we have two stopping conditions. The first condition is if the mean squared error drops below a certain value. In our case, it is 0.1, since the curve will converge much slower after. We ran the trial 50000 times, the MSE is still converging to 0.07, which is not efficient. The graph is shown below.



The other stopping condition is when the number of iterations reaches 20,000. We set 20,000 since it took a reasonable amount of time to run and the MSE is about to be around 0.1. It can also avoid having an infinite loop if the mean squared error never dips below the first stopping criterion from above.

4)

For part 4, we learnt Jason Brownlee's tutorial published July 2019, and used the Keras library to create the neural network.

First, we load the entire dataset from the irisdata file. Then, we store the four data dimensions, sepal length, speal width, petal length, petal width, into **x**, and the patterns into **y.**

We use the train_test_split function from the sklearn to make 70% of our data to be trained and 30% data to test.

Second, we define the model.

We define our keras model with `model = keras.Sequential()` and use a fully-connected network structure with two layers, which is defined by the Dense class.

```
model.add(keras.layers.Dense(4, input_dim=4, activation='tanh'))
model.add(keras.layers.Dense(3, activation='softmax'))
```

The first hidden layer has 4 neurons and uses the tanh activation function, with four input variables specified by the input_dim argument. And the output layer uses the softmax activation function.

Third, we compile, fit and evaluate our Keras model

```
model.compile(keras.optimizers.Adam(lr=0.04), 'categorical_crossentropy', metrics=['accuracy'])
```

The cross-entropy will calculate a score that summarizes the average difference between the actual and predicted probability distributions for all classes, so we specify the cross-entropy as the loss function by 'categorical_crossentropy'. Also, because it is a classification problem, we need to collect and report the classification accuracy, which is defined via the metrics argument: 'accuracy'.

```
model.fit(x_train, y_train, batch_size=128, epochs=100, verbose=0)
accuracy = model.evaluate(X_test, y_test)[1]
error = model.evaluate(X_test, y_test)[0]
print('Accuracy: {}'.format(accuracy))
print('Error: {}'.format(error))
y_pred = model.predict(X_test)
print(classification_report(y_test,y_pred > 0.5))
```

As the code shown above, we train our model on our test data with the fit() function over epochs which are split into batches. Then, we can evaluate the performance of the network, and we print out the accuracy and error, from the returning list, as well as the classification report.

We also train the data with the KNeighborsClassifier to compare the results.

The image below shows our results obtained from the neural network and also the knn. As we can see, the accuracy achieves 1.0 with our two-layer neural network, with 0.044 as the error. The accuracy of the knn is 0.977 which is also high. However, the neural network achieves the precision at 1.00 and performs better.

```
------------------------------------------------------------------
2/2 [==============================] - 0s 471us/step - loss: 0.0440 - accuracy: 1.0000
2/2 [==============================] - 0s 407us/step - loss: 0.0440 - accuracy: 1.0000
Accuracy: 1.0
Error: 0.04403883218765259
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        15
           1       1.00      1.00      1.00        15
           2       1.00      1.00      1.00        15

   micro avg       1.00      1.00      1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
 samples avg       1.00      1.00      1.00        45

2/2 [==============================] - 0s 398us/step - loss: 0.0440 - accuracy: 1.0000
Accuracy: 0.9777777777777777
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        15
           1       0.94      1.00      0.97        15
           2       1.00      0.93      0.97        15

   micro avg       0.98      0.98      0.98        45
   macro avg       0.98      0.98      0.98        45
weighted avg       0.98      0.98      0.98        45
 samples avg       0.98      0.98      0.98        45
```

Citation for our tutorial:
https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/ by Jason Brownlee on July 24, 2019.
Jason did the example for the binary classification, and we use the similar structure but with different layers and activation functions.