CSDS 391 P2 Write Up

Exercise 1. Clustering

1A. IMPLEMENTING K-MEANS:

Down below is the code for the main function:

```
class kMeansImplementation:
    def __init__(self, n_clusters = 3, max_iter = 5, random_state=1):
        '''
        parameters:
        n_clusters: number of clusters
        max_iter: number of iterations before the algorithm stops
        '''
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.random_state = random_state

    def kmeans_fit(self, points):
        iteration = 0
        current_sse = None
        previous_sse = None
        done = False
        objective_function = []

        centroids = initialize_centroids(points, self.n_clusters)


        while (done == False):
            classes, new_centroids, current_sse = methods(points, centroids)

            objective_function.append(current_sse)
            if(iteration == self.max_iter) or (iteration != 0 and abs(previous_sse - current_sse)<2):
                done = True
                plot_sse(objective_function)
                return classes, new_centroids, current_sse, objective_function

            else:
                previous_sse = current_sse
                centroids = new_centroids
                iteration += 1
                done = False
```

The constructor takes in *n_cluster*, representing the number of clusters, as well as *max_iter*, representing the number of iterations the function will run before the algorithm ends. **Kmeans_fit** is the function that takes in the data points and returns a plot of the sum of squared errors as a function of iterations.

Variables:

- *Iteration*: the current number of iterations
- *Current_sse*: the current sum of squared errors
- *Previous_sse*: the previous iteration's sum of squared errors
- *Done*: a boolean variable that helps exit the algorithm's while loop
- *Objective_function*: an array that takes in each iteration's sum of squared errors in order to plot the function

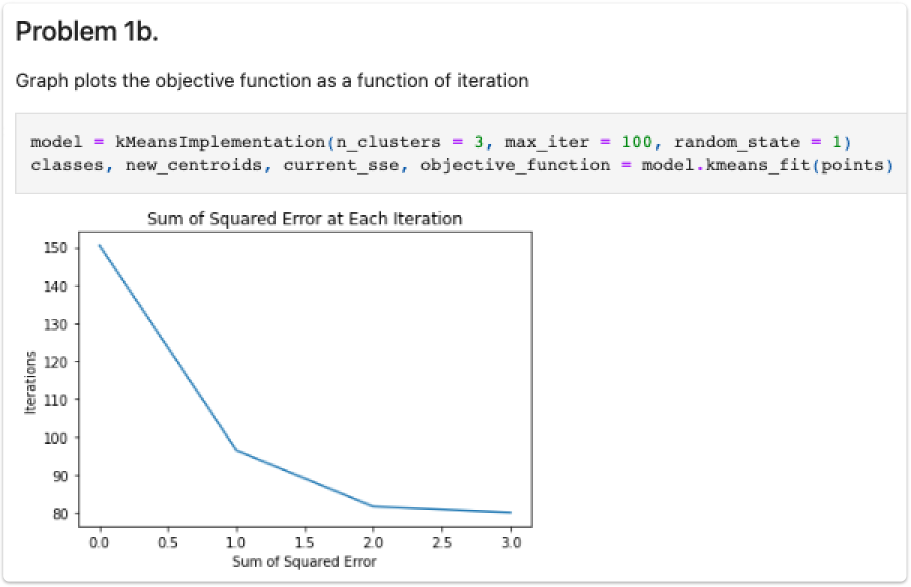
Helper Functions for *kmeans_fit*:

- *Initialize_centroids*(points, k) which helps choose the initial k number of centroids.
- *Methods*(points, centroids) combines all the helper methods into one call, in order of: 1. find_closest_centroid, 2. reinitialize_centroids, 3. error_function
- *Find_closest_centroid*(points, centroids) reclassifies each point to their nearest centroid.
- *Reinitialize_centroids*(points, centroids, classes) generates the new centroids based on the points within their class.
- *Error_function*(points, centroids, classes) calcualtes the sse.

1B. PLOTTING THE VALUE OF THE OBJECTIVE FUNCTION

Down below is the code for plotting the SSE function

```
def plot_sse(objective_function):
    domain = list(range(0,len(objective_function)))
    plt.figure()
    plt.plot(domain, objective_function)
    plt.title("Sum of Squared Error at Each Iteration")
    plt.xlabel("Sum of Squared Error")
    plt.ylabel("Iterations")
    plt.show()
```



Problem 1b.

Graph plots the objective function as a function of iteration

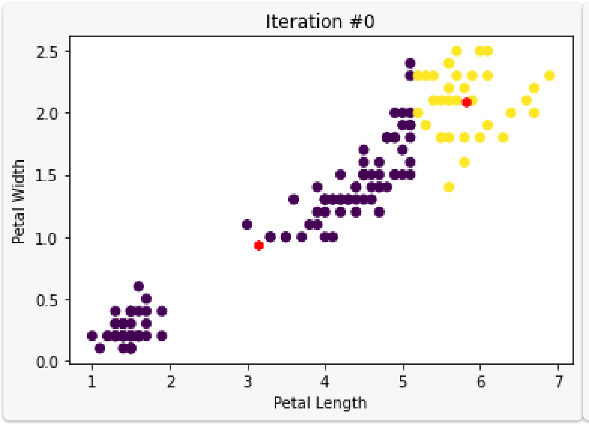
```
model = kMeansImplementation(n_clusters = 3, max_iter = 100, random_state = 1)
classes, new_centroids, current_sse, objective_function = model.kmeans_fit(points)
```

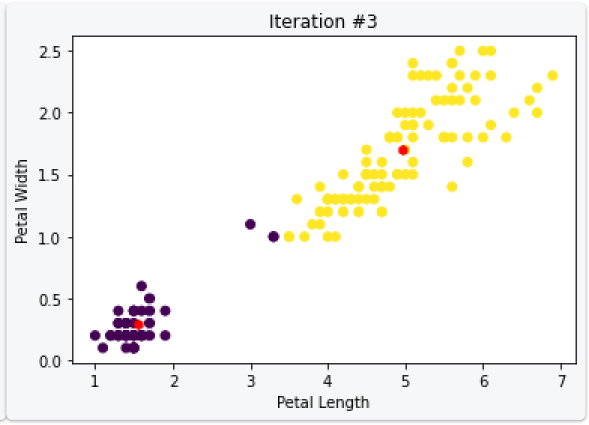
The way I implemented k-means, the algorithm would stop either when the maximum number of iterations is reached OR when the difference between the previous iteration's sse and the current iteration's sse is less than 2, meaning that the SSE function is converging. This is demonstrated by the graph above as you can see that the algorithm stops at iteration 3 as the function stagnates.

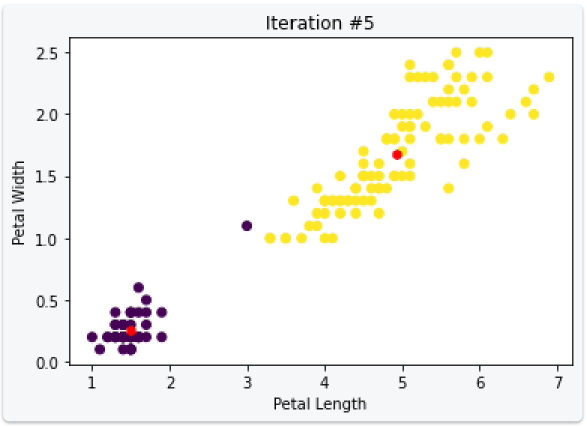
1C. PLOTTING THE RESULTS OF THE LEARNING PROCESS

K=2:

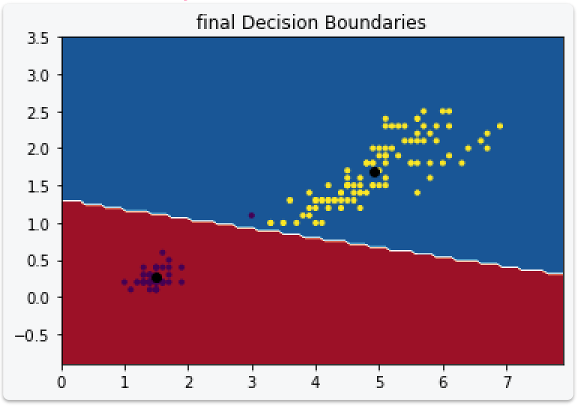
Iteration 0: Initial Cluster, *Iteration 3*: Intermediate Cluster, *Iteration 5*: Converged Cluster





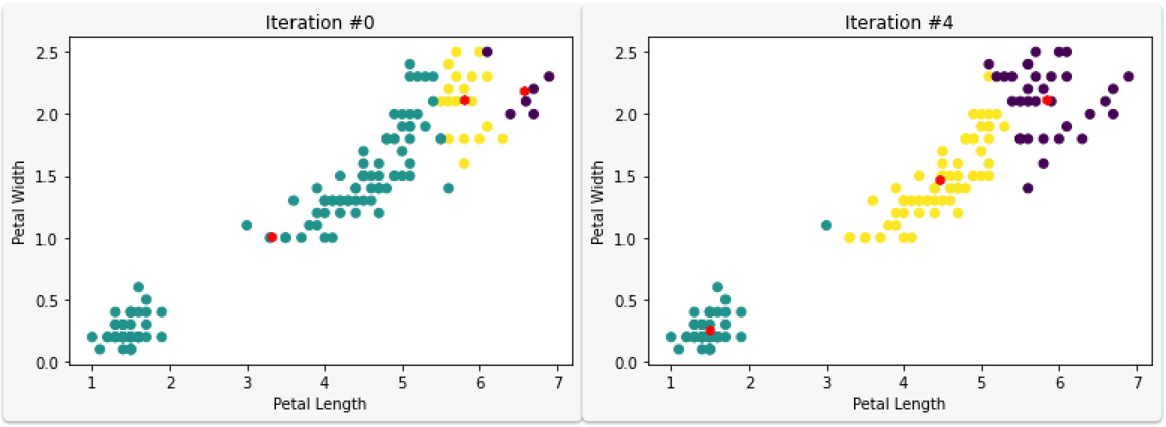


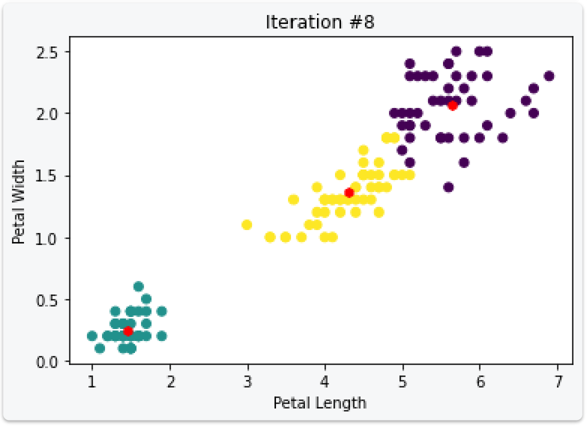
Decision Boundary:



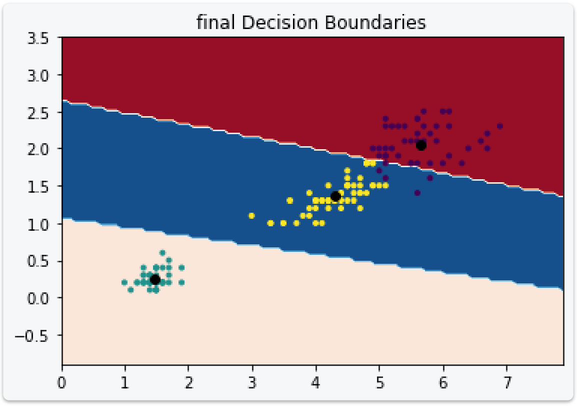
K=3:

Iteration 0: Initial Cluster, *Iteration 4*: Intermediate Cluster, *Iteration 8*: Converged Cluster





Decision Boundary:



1D. PLOT DECISION BOUNDARIES FOR OPTIMIZED PARAMETERS

The two graphs above show the decision boundareis for when k=2 and k=3. I plotted the decision boundaries by generating all possible points within x and y axis. Then I classified each of those points to the nearest centroid, using a contour map. That way, there is a definite linear decision boundary between each of the centroids.

Exercise 2. Linear Decision Boundaries

2A. LOADING THE IRIS DATA SET AND PLOTTING VIRGINICA AND VERSICOLOR CLASSES

Down below is the code for the loading the data and plotting the classes:

```
def read_data(file_name):
    df = pd.read_csv(file_name)
    df = df[(df['species']=='virginica')|(df['species']=='versicolor')]
    features = df[['petal_length', 'petal_width']].to_numpy()
    classes = df[['species']].to_numpy()
    df = df[['petal_length', 'petal_width', 'species']].to_numpy()
    return df, features, classes

def plot_classes(class1_features, class2_features):
    color = ["blue" if label=='virginica' else 'red' for label in classes]
    plt.scatter(np.array(features)[:,0], np.array(features)[:,1], marker='o', c = color)
    plt.xlabel('petal length')
    plt.ylabel('petal width')
    plt.title('Linear Decision Boundary using Sigmoid Function')
    plt.show()
```

2B. DEFINE A FUNCTION TO COMPUTE THE OUTPUT OF A SIMPLE ONE-LAYER NEURAL NETWORK

Down below is the code for computing the output of the neural network:

```
def compute_output(self, values):
    output = np.matmul(values, np.transpose(self.w[1:3])) + self.w[0]
    return 1/(1+np.exp(-output))

def individual_compute_output(self, x1, x2):
    output = self.w[0] + x1*self.w[1] + x2*self.w[2]
    return sigmoid(output)
```

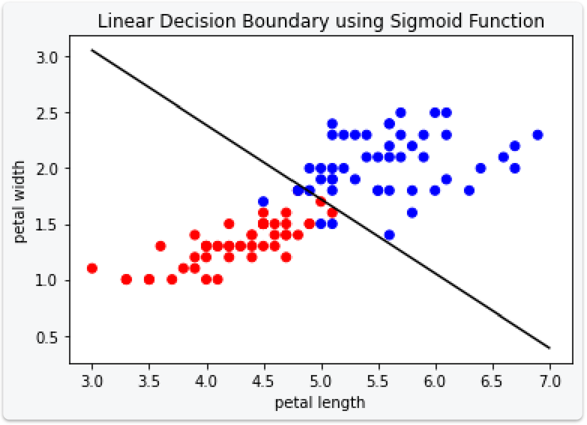
The first function computes the output of the neural network given the whole array of values. The second function compute the output using the single inputs for petal length and petal width. The output is computed through this equation:
$\hat{y} = w_1 * x_1 + w_2 * x_2 + b \rightarrow \frac{1}{1+exp(-\hat{y})} \rightarrow$ output.

2C. PLOT THE DECISION BOUNDARY

Down below is the code for plotting the decision boundary as well as classifying the classes.

```
def decision_boundary_plot(features, classes, weights, b):
    neuralNetwork = singleNN(weights, b)
    petal_length = np.linspace(3,7,40)
    petal_width = neuralNetwork.second(petal_length)

    plt.plot(petal_length, petal_width, c="black")
    color = ["blue" if label=='virginica' else 'red' for label in classes]
    plt.scatter(np.array(features)[:,0], np.array(features)[:,1], marker='o', c = color)
    plt.xlabel('petal length')
    plt.ylabel('petal width')
    plt.title('Linear Decision Boundary using Sigmoid Function')
    plt.show()

def classification(petal_length, petal_width, weights, b):
    sNN = singleNN(weights, b)
    z = sNN.individual_compute_output(petal_length, petal_width)
    if z<0.5:
        return 0
    else:
        return 1

weights = np.array([0.6, 0.9])
b = -4.55
decision_boundary_plot(features, classes, weights, b)
```

The first method takes in the parameters of the neural network, feature data as well as the associated classes. Then it plots the decision boundary on top of the scatter line. The second method classifies the flower whether it is above or below the line to their respective classes.

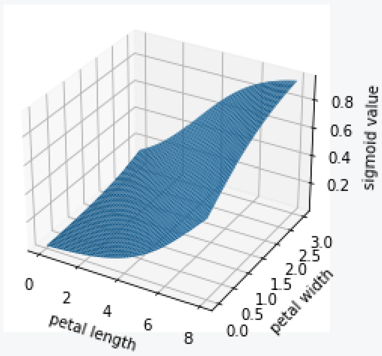


2D. PLOT THE SURFACE PLOT

Down below is the code for plotting the surface plot.

```
#Problem 2d
def surface_plot(weights, b):
    ax = plt.axes(projection= "3d")
    x = np.linspace(0,8,100)
    y = np.linspace(0,3,100)
    x1, y1 = np.meshgrid(x, y)

    sNN = singleNN(weights, b)
    output = sNN.individual_compute_output(x1,y1)
    surface = ax.plot_surface(x1, y1, output)

    ax.set_xlabel('petal length')
    ax.set_ylabel('petal width')
    ax.set_zlabel('sigmoid value')
    ax.set_title('Problem 2d: 3D plot of Neural Network')
    plt.show()
```

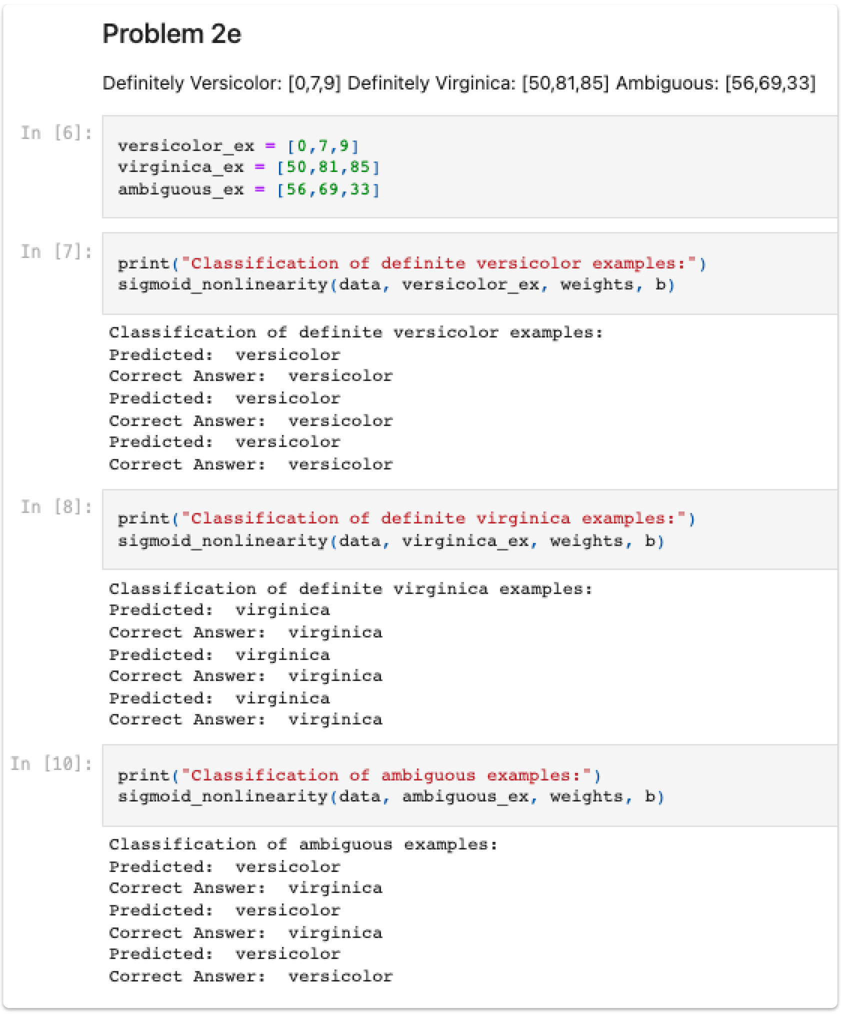
Problem 2d: 3D plot of Neural Network



2E. SHOW THE OUTPUT OF YOUR SIMPLE CLASSIFIER USING EXAMPLES

The code below

```
def sigmoid_nonlinearity(data, examples, weights, b):
    sNN = singleNN(weights, b)
    output = []
    correct_answer = []
    for i in examples:
        correct_answer.append(data[i][2])
        predicted = classification(data[i][0],data[i][1], weights, b)
        if predicted == 0:
            output.append('versicolor')
        else:
            output.append('virginica')
    for j in range(len(output)):
        print("Predicted: ",output[j])
        print("Correct Answer: ",correct_answer[j])
```



Problem 2e

Definitely Versicolor: [0,7,9] Definitely Virginica: [50,81,85] Ambiguous: [56,69,33]

```
In [6]:  versicolor_ex = [0,7,9]
         virginica_ex = [50,81,85]
         ambiguous_ex = [56,69,33]
```

```
In [7]:  print("Classification of definite versicolor examples:")
         sigmoid_nonlinearity(data, versicolor_ex, weights, b)
```

```
Classification of definite versicolor examples:
Predicted:  versicolor
Correct Answer:  versicolor
Predicted:  versicolor
Correct Answer:  versicolor
Predicted:  versicolor
Correct Answer:  versicolor
```

```
In [8]:  print("Classification of definite virginica examples:")
         sigmoid_nonlinearity(data, virginica_ex, weights, b)
```

```
Classification of definite virginica examples:
Predicted:  virginica
Correct Answer:  virginica
Predicted:  virginica
Correct Answer:  virginica
Predicted:  virginica
Correct Answer:  virginica
```

```
In [10]: print("Classification of ambiguous examples:")
         sigmoid_nonlinearity(data, ambiguous_ex, weights, b)
```

```
Classification of ambiguous examples:
Predicted:  versicolor
Correct Answer:  virginica
Predicted:  versicolor
Correct Answer:  virginica
Predicted:  versicolor
Correct Answer:  versicolor
```

As you can see, all the examples not close to the decision boundary were classified correctly. However, for the ambiguous examples, only one out of three were classified correctly.

Exercise 3. Neural Networks

3A/E. CALCULATE THE MSE, COMPUTE THE SUMMED GRADIENT

Down below is the code for calculating the MSE function

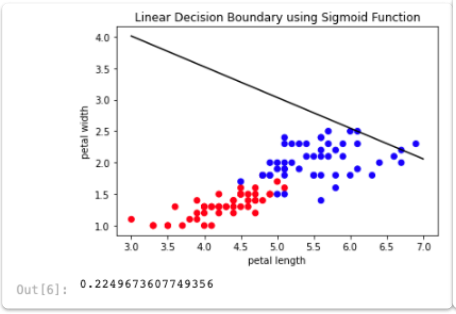
```
def mse_given_data(data, pattern, weights, b, learning_rate):
    sNN = singleNN(weights, b)
    w = weights
    w1_update = 0
    w2_update = 0
    b_update = 0
    total_squared_residual = 0
    for i in range(len(data)):
        output = b + (data[i][0] * weights[0]) + (data[i][1] * weights[1])
        sig = sigmoid(output)
        dSig = derivative_sigmoid(sig)

        if pattern[i] == 'versicolor':
            actual = 0
        else:
            actual = 1

        residual = sig-actual
        squared_residual = (residual)**2
        b_update += (residual * dSig)
        w1_update += (residual * dSig) * data[i][0]
        w2_update += (residual * dSig) * data[i][1]
        total_squared_residual += squared_residual

    w[0] = weights[0] - learning_rate * (w1_update * 2 / (len(data)))
    w[1] = weights[1] - learning_rate * (w2_update * 2 / (len(data)))
    b = b - learning_rate * (b_update * 2 / (len(data)))

    mse = total_squared_residual / (len(data))

    return mse, w, b
```

I computed the mean squared error by calculating the error between the sigmoid value and the respective class for every example, and then squaring the total error and dividing it by the total number of examples.

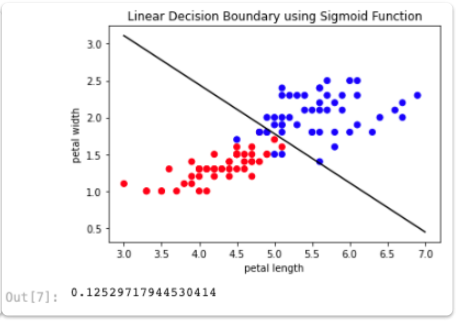
The above method also incorporates the summed graident by calculating the derivative sigmoid value and then multiplying that to the residual error and respective data feature at each iteration of the for loop and adding it to the total "update" value needed for that specific weight. Then I would multiply the "learning rate" with the total update value times two. Then I would divide that by the total number of examples, and subtract it from the original weight, resulting in the new weight. The same would be repeated for the bias without multiplying the derivative sigmoid value with the data feature.

3B. COMPUTE THE MEAN SQUARED ERROR FOR TWO DIFFERENT WEIGHTS

The two graphs below show the decision boundary for two separate weights. The first one shows the large error weight/bias with an mse of 0.225. The second graph shows the small error weight/bias with an mse of 0.125.



```
Out[6]:  0.2249673607749356
```



```
Out[7]:  0.12529717944530414
```

Example:

Weights: Large error[0.41, 0.84], Small error[0.6, 0.9]
Bias: Large error[-4.6], Small error[-4.6]

3C/D. MATHEMATICAL DERIVATION THE GRADIENT OF THE OBJECTIVE FUNCTION IN SCALAR/VECTOR FORM

Down below is the step by step process of the derivation:

Objective function: $(y-\sigma(z))^2$ where y represents the ground truth and $\sigma(z)$ represents the output of the sigmoid function.

$$z = w^T x_n = w_1 x_1 + w_2 x_2 + \dots + w_n x_n$$

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

$$\frac{\partial}{\partial w_0}(y-\sigma(z))^2 = 2(y-\sigma(z)) \cdot \frac{\partial}{\partial w_0}\sigma(z)$$

$$= 2\left(y-\frac{1}{1+e^{-w^T x}}\right)\left[\frac{\partial}{\partial w_0}\frac{1}{1+e^{-w^T x}}\right] \rightarrow \left[\frac{\partial}{\partial w_0}\frac{1}{1+e^{-(w_1 x_1+\dots-w_n x_n)}} = \left(\frac{e^{w^T x}}{(e^{w^T x}+1)^2}\right)(-x_0) \text{ where } x_0 = 1 \text{ because } w_0 \text{ is the bias} = \left(\frac{e^{w^T x}}{(e^{w^T x}+1)^2}\right)(-1)\right]$$

$$= -2\left(y-\frac{1}{1+e^{-w^T x}}\right)\left(\frac{e^{w^T x}}{(e^{w^T x}+1)^2}\right)$$

$$\frac{\partial}{\partial h}\frac{1}{1+e^{-h}} = \frac{e^h}{(e^h+1)^2}$$

$$\frac{\partial}{\partial w_1}(y-\sigma(z))^2 = -2\left(y-\frac{1}{1+e^{-w^T x}}\right)\left(\frac{e^{w^T x}}{(1+e^{w^T x})^2}\right)x_1$$

$$\frac{\partial}{\partial w_2}(y-\sigma(z))^2 = -2\left(y-\frac{1}{1+e^{-w^T x}}\right)\left(\frac{e^{w^T x}}{(1+e^{w^T x})^2}\right)x_2$$

Scalar Format:

$$w_0 = -2\left(y-\frac{1}{1+e^{-w^T x}}\right)\left(\frac{e^{w^T x}}{(e^{w^T x}+1)^2}\right)$$

$$w_1 = -2\left(y-\frac{1}{1+e^{-w^T x}}\right)\left(\frac{e^{w^T x}}{(e^{w^T x}+1)^2}\right)x_1$$

$$w_2 = -2\left(y-\frac{1}{1+e^{-w^T x}}\right)\left(\frac{e^{w^T x}}{(e^{w^T x}+1)^2}\right)x_2$$

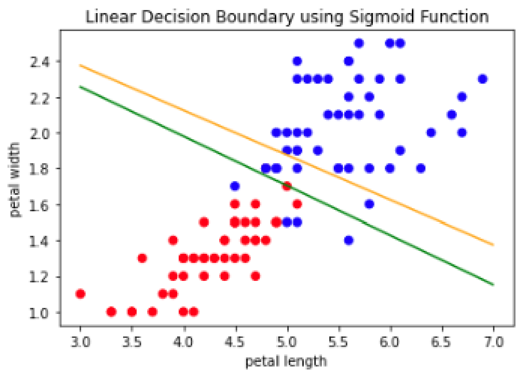
Vector Format:

$$\left[\frac{\partial}{\partial w_i}\right] = \begin{bmatrix}\frac{\partial}{\partial w_0}\\ \frac{\partial}{\partial w_1}\\ \frac{\partial}{\partial w_2}\end{bmatrix} = \begin{bmatrix}-2\left(y-\frac{1}{1+e^{-w^T x}}\right)\left(\frac{e^{w^T x}}{(e^{w^T x}+1)^2}\right)\\ -2\left(y-\frac{1}{1+e^{-w^T x}}\right)\left(\frac{e^{w^T x}}{(e^{w^T x}+1)^2}\right)x_1\\ -2\left(y-\frac{1}{1+e^{-w^T x}}\right)\left(\frac{e^{w^T x}}{(e^{w^T x}+1)^2}\right)x_2\end{bmatrix} = -2\left(y-\frac{1}{1+e^{-w^T x}}\right)\left(\frac{e^{w^T x}}{(e^{w^T x}+1)^2}\right)X$$

3E. ILLUSTRATE THE GRADIENT

In [8]:
```
weights = [0.2, 0.8]
b = -2.5
plot_gradient_descent(features, classes, weights, b)
```

```
initial mse:  0.16646612281878398
inital weights:  [0.22338466461904183, 0.8101296968610548]
initial bias:  -2.5
new mse:  0.16123934750561766
new weights:  [0.3596189746983812, 0.8776983441372131]
new bias:  -2.4974500779067528
```



Exercise 4. Learning a decision boundary through optimization

4A. IMPLEMENT GRADIENT DESCENT TO OPTIMIZE THE DECISION BOUNDARY

Refer to the file for the source code:

4B. PLOT THE CURRENT DECISION BOUNDARY AND THE LEARNING CURVE

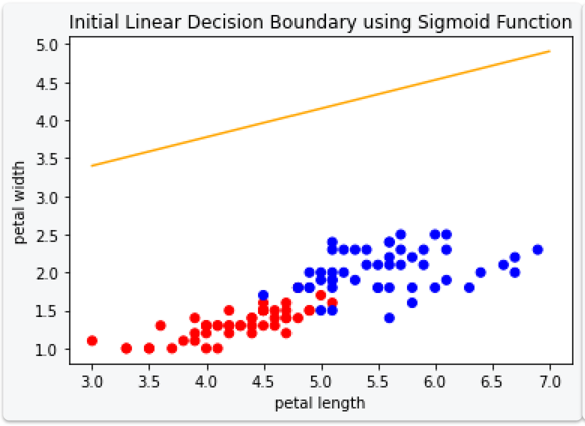
Refer to the file for the source code

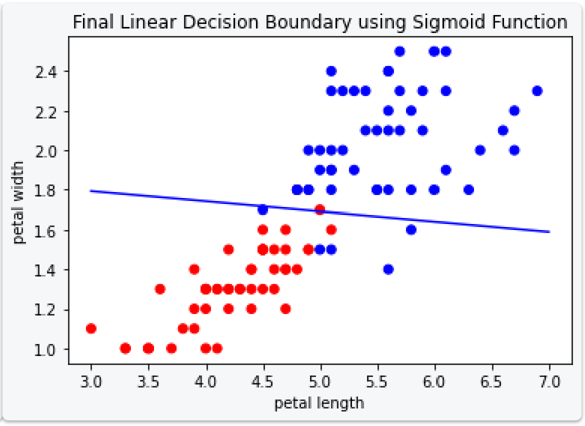
4C. RUNNING THE CODE WITH RANDOM WEIGHTS

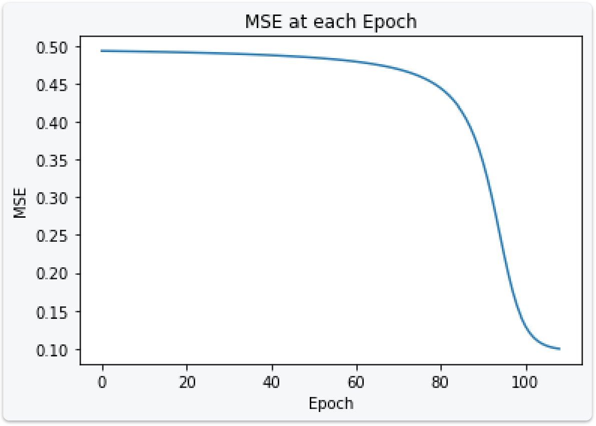
Random Weight and Biases:

Initial: Weight: [-0.830, 2.203], Bias: -4.999
Final: Weight: [0.131, 2.568], Bias: -4.998

The first two graphs show the initial and final decision boundary using the sigmoid function. The third graph plots the MSE (objective function) at each epoch/iteration.







4D. EXPLANATION FOR GRADIENT STEP SIZE

The reason why I chose 0.05 for my step size was so that it would increment my weights at a rate that would not be too slow requiring many more iterations as well as a rate that is not too fast that it would skip over the ideal settings. I observed that a step size of 0.1 would cause the current weight parameters to converge at nearly half the number of required epochs. However, there is a chance that using another set of weights, the gradient descent would completely miss the optimized weights.

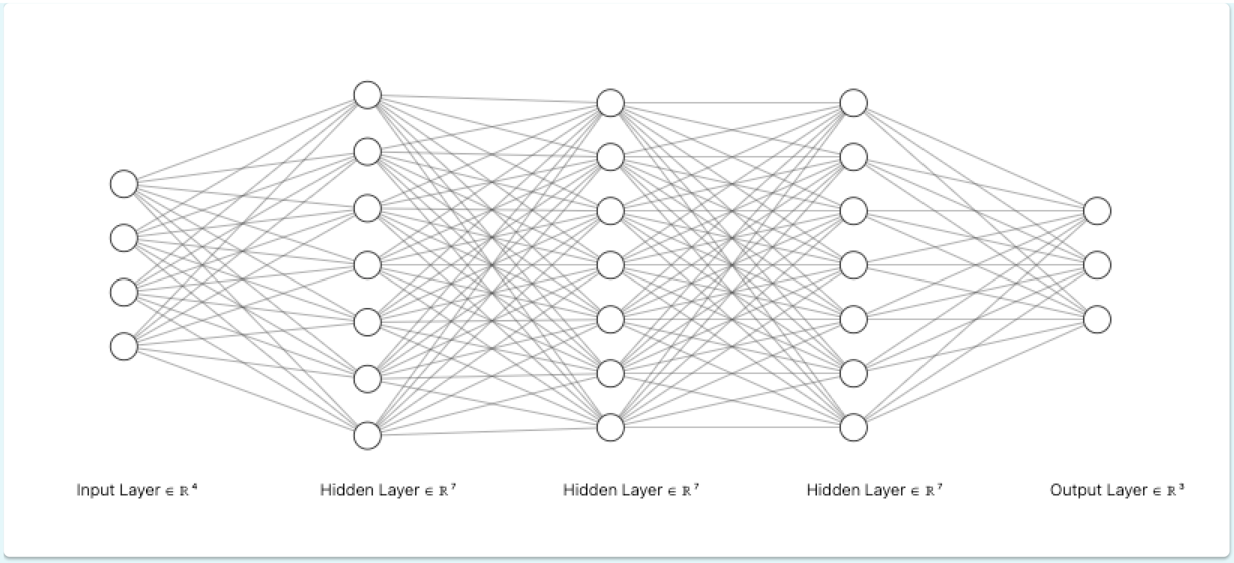
4E. EXPLANATION FOR STOPPING CRITERION

I had three different ways to stop the algorithm. One criterion, the maximum number of iterations, was so that the algorithm would never run infinitely. The second criterion was made while observing the mse function. When an MSE = 0.1, that means that the algorithm is already making minimal mistakes, so there is not much more benefit of optimizing those weights. The third criterion is to end the algorithm early when the learning curve converges, meaning that the difference between the previous MSE and the current MSE is less than 0.00001.

Exercise 5. Using a machine learning toolbox

I decided to use scikit-learn's toolbox of the multilayer perceptron, also abbreviated as mlp. Within sklearn's library, there is the mlp classifier, where you can designate the number of neurons as well as the number of hidden layers within the neural network. I decided to do 3 hidden layers with 7 neurons each. The input layer has 4 neurons for each of the 4 features. The output layer has 3 neurons for the respective classes.

Architecture:



I chose to do 7 neurons per hidden layer because of this article that I read online for the general rule of thumb when choosing the number of neurons.

Rule of Thumb

1. The number of hidden neurons should be between the input size and output size
2. The number of hidden neurons should be 2/3 size of the input + size of the output
3. The number of hidden neurons should be less than twice the size of the input
   Cited: https://medium.com/geekculture/introduction-to-neural-network-2f8b8221fbd3

```python
import pandas as pd
import numpy as np
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn import preprocessing
from sklearn.metrics import accuracy_score

df = pd.read_csv('irisdata.csv')
encoder = preprocessing.LabelEncoder()
encoder.fit(df.species)
df['class'] = encoder.transform(df.species)
x = df[['sepal_length', 'sepal_width', 'petal_length', 'petal_width']].to_numpy()
y = df[['class']]
y = np.ravel(y, order='C')
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=1)

sc = StandardScaler()
sc.fit(x_train)
x_train_std = sc.transform(x_train)
x_test_std = sc.transform(x_test)

model = MLPClassifier(hidden_layer_sizes=(7,7,7), activation = 'relu', solver='lbfgs', max_iter=2000).fit(x_train
prediction = model.predict(x_test_std)
print("The accuracy score is" ,accuracy_score(y_test, prediction))
```

The accuracy score is 0.96667.

First I loaded the data into my notebook and in order for the algorithm to be able to train the data, I had to convert the class strings into integers. For example, setosa, versicolor, and virginica are represented by 0, 1, and 2 respectively. Next I split the data into test and train set so that there would be no test set leakage when training the model. I also further preprocessed the data by normalizing/scaling the data, so that each feature has a respective mean of 0 and a standard deviation of 1. However, scaled the test data using the normalizing constants from the training data so that we are not making any decisions based on the test data. Then I trained the model on the standardized training data, before predicting the class based on the standardized test data. I mainly learned about about the machine learning/neural network workflow. From importing the data set into preprocessing the data handling missing data and feature extraction, all before selecting a activation function/objective function for the neural network.