CSDS 391 Programming Assignment 2 Writeup

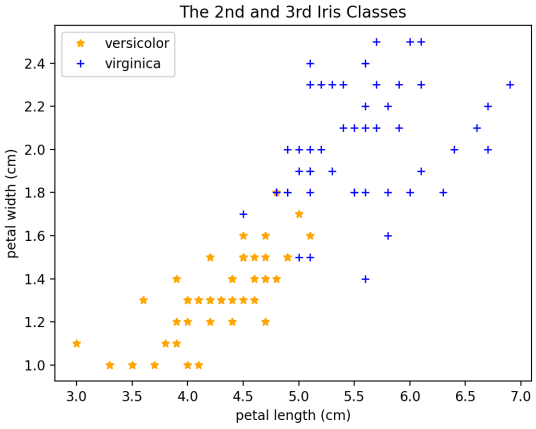
Ningjia Huang, Tianxi Zhao

Due On: December 4, 2020

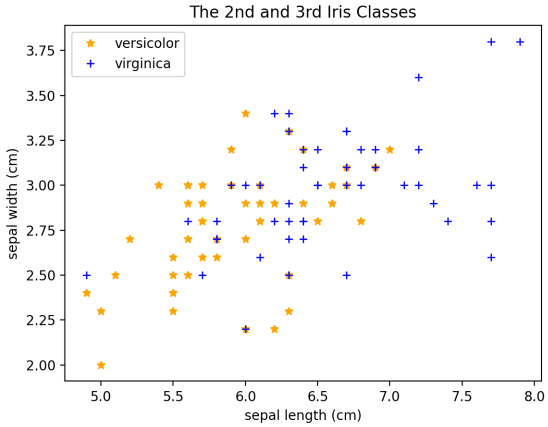
Problem 1:

(a)

The plot of 2nd and 3rd classes of the dataset with petal length v.s. petal width is as following:



The plot of 2nd and 3rd classes of the dataset with sepal length vs. sepal width:



We can see that petal width vs. petal length can better seperate the 2nd and 3rd classes.

(b)

We define the output unit: y = w_1· petal_length + w_2· petal_width + w_0 w_1 and w_2 are weigths, w_0 is bias. The sigmoid function is:

$$sigmoid = \frac{1}{1 + e^{-y}}$$

The following code implements the sigmoid function:

```
#Exercise 1. b. computes the output of simple one-layer neural ne
def sigmoid(length:float, width:float) -> float:
    w = [-34.707, 4.530, 7.680]
    z = w[0] + w[1] * length + w[2] * width
    sigmoid = 1 / (1 + math.exp(-z))
    return sigmoid
```

Implementation of Decision Boundary:

```
w = [-34.707, 4.530, 7.680]
x = dataset['petal_length']
y = [-((w[1] * x_value + w[0])/ w[2]) for x_value in x]
```

(c)

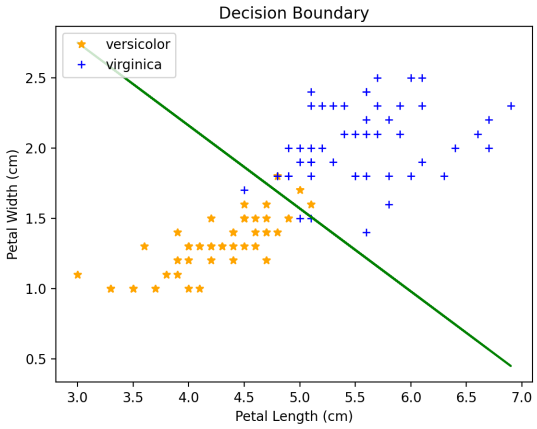
Because the output unit is y = w_1· petal_length + w_2· petal_width + w_0 and the decision boundary is when y = 0, we use x_1 to represent petal length and x_2 to represent petal width:

$$0 = w_1 \cdot x_1 + w_2 \cdot x_2 + w_0$$
$$x_2 = -\frac{w_1 \cdot x_1 + w_0}{w_2}$$

Decision Boundary Code:

```
w = [-34.707, 4.530, 7.680]
x = dataset['petal_length']
y = [-((w[1] * x_value + w[0])/ w[2]) for x_value in x]
```

We draw the decision boundary on the graph using python: $plt.plot(x_1, x_2)$

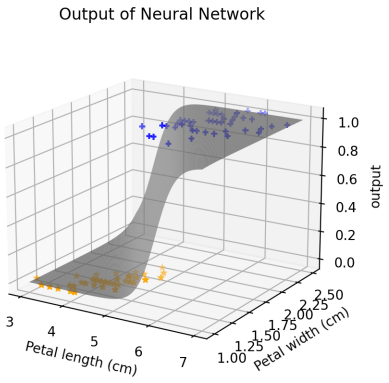


(d)

From the graph generated in (a), we can see that the range of petal length is from about 3.0 to 7.0 cm and the range of petal width is from about 1.0 to 2.5 cm, so we set the range of petal length from 3.0 to 7.0 cm and the range of petal width from 1.0 to 2.5 cm for the surface that we are going to draw. Then we use the output function we defined in (b) to calculate the output of each pair of petal length and petal width.
Code to prepare petal length, petal wieth, and the output:

```
x = np.arange(3.0, 7.0, 0.01)
y = np.arange(1.0, 2.5, 0.01)
X, Y = np.meshgrid(x, y)
z = np.array([sigmoid(a,b) for a,b in zip(np.ravel(X), np.ravel(Y))])
Z = z.reshape(X.shape)
```

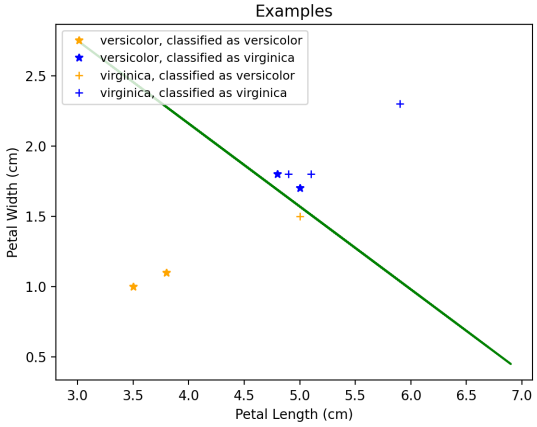
We use the petal length, petal wieth, and the output to draw a 3D neural network:



(e)

We selected 8 examples:

petal length (cm)	petal width (cm)	species
4.8	1.8	versicolor
5.0	1.7	versicolor
3.5	1.0	versicolor
3.8	1.1	versicolor
5.1	1.8	virginica
5.0	1.5	virginica
4.9	1.8	virginica
5.9	2.3	virginica



Problem 2:

(a)

The mean squared error is calculated using the following equation:

$$MSE = \frac{1}{n}\sum_n (Y_i - \hat{Y}_i)^2$$

, where Y_i is the actual category of the ith item and $\hat{Y}_i$ is the predicted category by using our neural network.

The following codes compute the **mean squared error** for iris data. The parameter "data_vector" are the attributes we would like to take into account, it should be a dataframe of attributes. w_0, w_1, w_2 define the weights of neural network. The parameter "pattern_classes" is a dataframe of the category corresponding to the data_vectors. The **mean_square_error** makes use of the logistic non-linearity function in Problem 1.

```
# data vectors in dataframe, pattern classes in list
def mean_square_error(data_vectors, w0, w1, w2,
    pattern_classes):
     n = data_vectors.shape[0]
     data_vectors_list = data_vectors.values.tolist()
     pattern_classes_list = pattern_classes.tolist()
     temp_mse = 0
     for i in range(n):
         temp_mse = temp_mse + np.square(pattern_classes_list[
    i] - one_layer_network(w0, w1, w2, data_vectors_list[i
    ][0], data_vectors_list[i][1]))
     mse = temp_mse/n
     return mse
```

Listing 1: Mean Squared Error Calculation

(b)

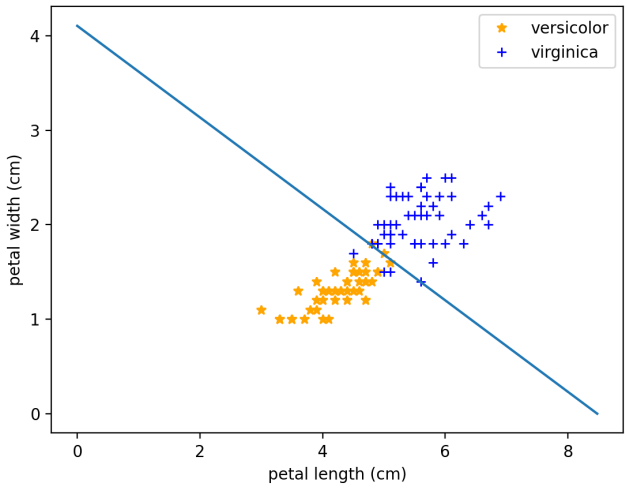
To plot the linear boundary, we find the x-intercept and y-intercept by using the following code:

```
plot([0,-w0/w1],[-w0/w2,0])
```

Listing 2: Plot Mean Squared Error

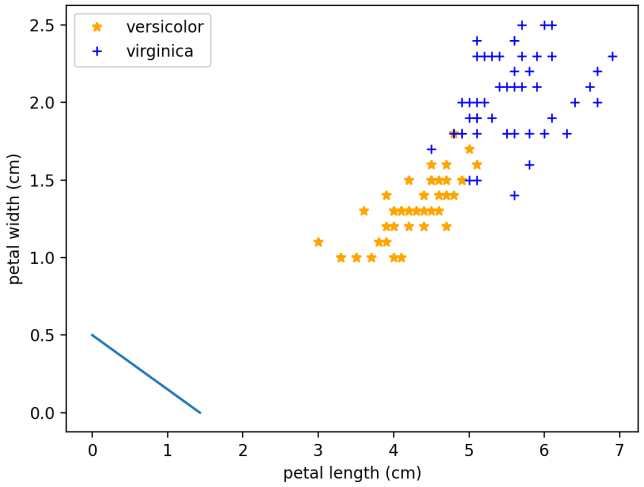
2.b.1

The first set of weights we've chosen is $[-3.9, 0.46, 0.95]$. The mean squared error we concluded is 0.13235245817007218. The following figure shows the linear boundary with our chosen set of weights. As we can tell from the graph, the linear boundary generally seperates the two categories of iris.



2.b.2

The second set of weights we've chosen is $[-1, 0.7, 2]$. The mean squared error we concluded is 0.48802189762271864. The following figure shows the linear boundary with our chosen set of weights. As we can tell from the graph, the linear boundary does not seperate these two categories at all.



(c)

To compute the gradient of the objective function, we need to take the derivative of the objective function. Assume $\mathbf{x}_i$ is the ith row with attribute petal length and petal width such that $x_i = [1, x_{i1}, x_{i2}]$(1 is placed here because w_0 requires 1 to be its coefficient). $\mathbf{w}$ is the set of weights for the attributes. The logistic function we get is:

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

Therefore, the mean squared function becomes:

$$MSE = \frac{1}{N} \sum_{i=1}^{n} (\sigma(w \cdot x_i) - y_i)^2$$

By taking the derivative of this objective function with respect to w_0, we have:

$$\frac{\partial MSE}{\partial w_0} = \frac{2}{N} \sum_{i=1}^{N} (\sigma(w \cdot x_i) - y_i) \frac{\partial \sigma}{\partial w_0}$$
$$\frac{\partial MSE}{\partial w_0} = \frac{2}{N} \sum_{i=1}^{N} (\sigma(w \cdot x_i) - y_i)(1 - \sigma(w \cdot x_i))(\sigma(w \cdot x_i)) \frac{\partial (w \cdot x)}{\partial w_0}$$
$$\frac{\partial MSE}{\partial w_0} = \frac{2}{N} \sum_{i=1}^{N} (\sigma(w \cdot x_i) - y_i)(1 - \sigma(w \cdot x_i))(\sigma(w \cdot x_i))$$

Similarly, by only changing the last derivative $\frac{\partial (w \cdot x)}{\partial w_0}$ to $\frac{\partial (w \cdot x)}{\partial w_1}$ and $\frac{\partial (w \cdot x)}{\partial w_2}$, we can compute the derivative of the objective function with respect to w_1 and w_2:

$$\frac{\partial MSE}{\partial w_1} = \frac{2}{N}\sum_{i=1}^{N}(\sigma(w\cdot x_i)-y_i)(1-\sigma(w\cdot x_i))(\sigma(w\cdot x_i))x_{i1}$$
$$\frac{\partial MSE}{\partial w_2} = \frac{2}{N}\sum_{i=1}^{N}(\sigma(w\cdot x_i)-y_i)(1-\sigma(w\cdot x_i))(\sigma(w\cdot x_i))x_{i2}$$

Therefore, the gradient of MSE is:

$$\nabla MSE = \begin{bmatrix} \frac{\partial MSE}{\partial w_0} \\ \frac{\partial MSE}{\partial w_1} \\ \frac{\partial MSE}{\partial w_2} \end{bmatrix} = \begin{bmatrix} \frac{2}{N}\sum_{i=1}^{N}(\sigma(w\cdot x_i)-y_i)(1-\sigma(w\cdot x_i))(\sigma(w\cdot x_i)) \\ \frac{2}{N}\sum_{i=1}^{N}(\sigma(w\cdot x_i)-y_i)(1-\sigma(w\cdot x_i))(\sigma(w\cdot x_i))x_{i1} \\ \frac{2}{N}\sum_{i=1}^{N}(\sigma(w\cdot x_i)-y_i)(1-\sigma(w\cdot x_i))(\sigma(w\cdot x_i))x_{i2} \end{bmatrix}$$

(d)

scalar form:

The scalar form of the gradient is computed in Problem 2 (c):

$$\nabla MSE = \begin{bmatrix} \frac{\partial MSE}{\partial w_0} \\ \frac{\partial MSE}{\partial w_1} \\ \frac{\partial MSE}{\partial w_2} \end{bmatrix} = \begin{bmatrix} \frac{2}{N}\sum_{i=1}^{N}(\sigma(w\cdot x_i)-y_i)(1-\sigma(w\cdot x_i))(\sigma(w\cdot x_i)) \\ \frac{2}{N}\sum_{i=1}^{N}(\sigma(w\cdot x_i)-y_i)(1-\sigma(w\cdot x_i))(\sigma(w\cdot x_i))x_{i1} \\ \frac{2}{N}\sum_{i=1}^{N}(\sigma(w\cdot x_i)-y_i)(1-\sigma(w\cdot x_i))(\sigma(w\cdot x_i))x_{i2} \end{bmatrix}$$

vector form:

Let $X = \begin{bmatrix} x_{11}, x_{12}, 1 \\ \dots \\ x_{n1}, x_{n2}, 1 \end{bmatrix}$, which is a matrix represents the attributes(1 corresponds for bias). Since the term $\frac{2}{N}\sum_{i=1}^{N}(\sigma(w\cdot x_i)-y_i)(1-\sigma(w\cdot x_i))(\sigma(w\cdot x_i))$ is independent of the dimension j and common for $\frac{\partial MSE}{\partial w_0}$, $\frac{\partial MSE}{\partial w_1}$, and $\frac{\partial MSE}{\partial w_2}$, we can first express this term as $z_i = (\sigma(w\cdot x_i)-y_i)(1-\sigma(w\cdot x_i))(\sigma(w\cdot x_i))$. The gradient of MSE becomes:

$$\nabla MSE = \frac{2}{m}\begin{bmatrix} \sum_{i=1}^{N} z_i \\ \sum_{i=1}^{N} z_i x_{i1} \\ \sum_{i=1}^{N} z_i x_{i2} \end{bmatrix} = \frac{2}{m}\begin{bmatrix} 1, \dots, 1 \\ x_{11}, \dots, x_{n1} \\ x_{12}, \dots, x_{n2} \end{bmatrix}\begin{bmatrix} z_1 \\ \dots \\ z_n \end{bmatrix} = \frac{2}{m}\mathbf{X^T}\begin{bmatrix} z_1 \\ \dots \\ z_n \end{bmatrix}$$

Then, we attempt to expand the vector $z_i = \begin{bmatrix} z_1 \\ \dots \\ z_n \end{bmatrix}$:

$$z_i = \begin{bmatrix} (\sigma(\mathbf{w}\cdot\mathbf{x_1}) - y_1)(1-\sigma(\mathbf{w}\cdot\mathbf{x_1}))(\sigma(\mathbf{w}\cdot\mathbf{x_1})) \\ \dots \\ (\sigma(\mathbf{w}\cdot\mathbf{x_n}) - y_n)(1-\sigma(\mathbf{w}\cdot\mathbf{x_n}))(\sigma(\mathbf{w}\cdot\mathbf{x_n})) \end{bmatrix} =$$
$$\begin{bmatrix} \sigma(\mathbf{w}\cdot\mathbf{x_1}) - y_1 \\ \dots \\ \sigma(\mathbf{w}\cdot\mathbf{x_n}) - y_n \end{bmatrix} \star \begin{bmatrix} 1-\sigma(\mathbf{w}\cdot\mathbf{x_1}) \\ \dots \\ 1-\sigma(\mathbf{w}\cdot\mathbf{x_n}) \end{bmatrix} \star \begin{bmatrix} \sigma(\mathbf{w}\cdot\mathbf{x_1}) \\ \dots \\ \sigma(\mathbf{w}\cdot\mathbf{x_n}) \end{bmatrix}$$

The $\star$ operation is defined as multiplying the ith row's elements. To simplify the expression, let $\sigma(\mathbf{Xw}) = \begin{bmatrix} \sigma(\mathbf{w^T x_1}) \\ \dots \\ \sigma(\mathbf{w^T x_n}) \end{bmatrix}$. Thus, z_i can be expressed as $z_i = (\sigma(\mathbf{Xw}) - \mathbf{y}) \star (1 - \sigma(\mathbf{Xw})) \star \sigma(\mathbf{Xw})$. The vector form of gradient can be expressed as:

$$\nabla MSE = \frac{2}{m}\mathbf{X^T}(\sigma(\mathbf{Xw}) - \mathbf{y}) \star (\mathbf{1} - \sigma(\mathbf{Xw})) \star \sigma(\mathbf{Xw})$$

(e)

The code we used to calculate the gradient is as following:

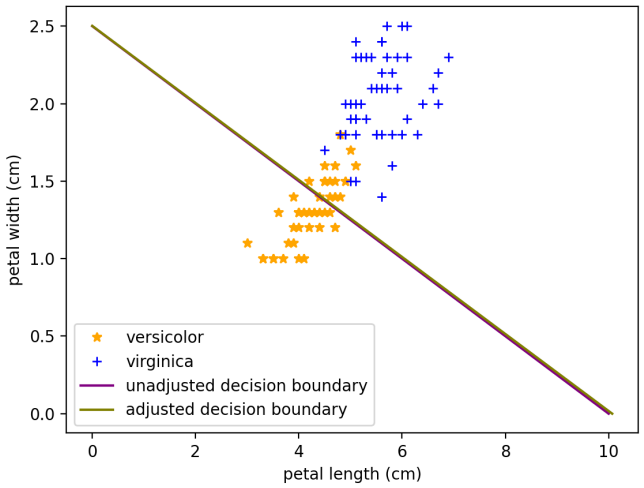
```
def summed_gradient(x, w0, w1, w2, y):
    sigmoid_list = []
    n = len(x)
    error_list = []
    coefficient_list = []
    x = x.values.tolist()
    y = y.tolist()
    for i in range(len(x)):
        sigmoid_list.append(one_layer_network(w0, w1, w2, x[i][0], x[i][1]))
        error_list.append(sigmoid_list[i] - y[i])
        coefficient_list.append(error_list[i] * sigmoid_list[i] * (1 - sigmoid_list[i]))
    # number of rows = number of rows in data vectors, number of columns = number of columns in data vector + 1 since 1 for bias coefficient
    temp_matrix = np.ones((len(x), len(x[0]) + 1))
    temp_matrix[:, 1:] = x
    sum_term = np.zeros((len(x), len(x[0]) + 1))
    for i in range(len(coefficient_list)):
        sum_term[i] = (2/n) * temp_matrix[i] * coefficient_list[i]
    return np.sum(sum_term, axis = 0)
```

Listing 3: Calculate Gradient

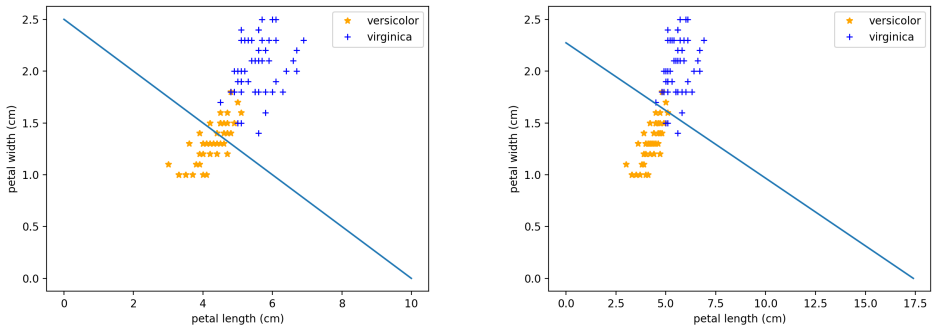
Let step size = 0.01. With the input weight $[w0, w1, w2] = [-5, 0.5, 2]$, we concluded the following table for the illustration of how the decision boundary changes for a small step:

	old data	adjusted data
weights	[-5, 0.5, 2]	[-5.00080742 0.49665344 1.9989812]
MSE	0.48802189762271864	0.4879972995169846
decision boundary	m = -0.35, b = 0.5	m = -0.3497940478, b = 0.500093299

As we can tell from the table, the mean squared error reduced a little bit as we made a small step. To visualize the change of decision boundary, please refer to the following graph:



Since the small step is kind of hard to tell, we run this process for 10000 times in order to enlarge the difference. The following two graphs show the decision boundary before the adjustment and after the adjustment:



The code we used for this illustration is:

```
def illustrate_summed_gradient(x, w0, w1, w2, y):
    num_of_iter = []
    for i in range(10000):
        num_of_iter.append(i+1)
        temp1, temp2, temp3 = summed_gradient(x, w0, w1, w2,
   y)
        w0 = w0 - 0.01 * temp1
        w1 = w1 - 0.01 * temp2
        w2 = w2 - 0.01 * temp3
    plot([0,-w0/w1],[-w0/w2,0])
```

Listing 4: Illustration of the Change of Gradient

Problem 3:

(a)

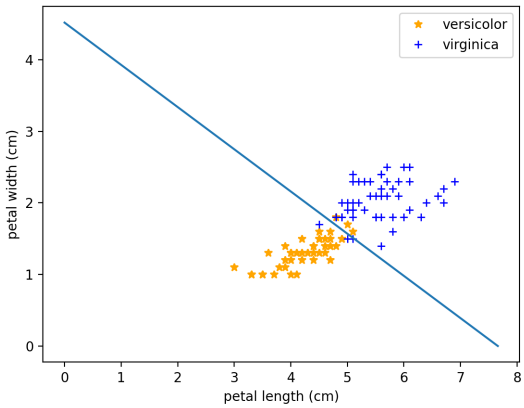
The code for gradient descent is as following:

```
def gradient_descent(a, b, x, w0, w1, w2, y):
    precision = 0.001 # mse we would like to reach
    max_iters = 20000 # max number of iterations
    iteration_counter = 0
    step_size = 0 # using armijo to update
    return_w = []
    current_w = [w0,w1,w2]
    current_mse = mean_square_error(x, w0, w1, w2, y)
    current_sum_g = summed_gradient(x, w0, w1, w2, y)
    improv_checker = 1 # check whether performed better
```

```
    # if current mse > the precision we defined and the number of iteration does not exceed the max iteration execute the gradient descent
    while mean_square_error(x, w0, w1, w2, y) > precision and iteration_counter < max_iters:
        if improv_checker > 0:
            return_w = current_w
        iteration_counter += 1
        temp0, temp1, temp2 = summed_gradient(x, w0, w1, w2, y)
        step_size = armijo_updating(a, b, x, y, w0, w1, w2)
        w0 = w0 -  step_size * temp0
        w1 = w1 -  step_size * temp1
        w2 = w2 - step_size * temp2
        current_w = [w0,w1,w2]
        next_mse = mean_square_error(x, w0, w1, w2, y)
        improv_checker = current_mse - next_mse
        current_mse = next_mse
        current_sum_g = summed_gradient(x, w0, w1, w2, y)
        print(w0, w1, w2, mean_square_error(x, w0, w1, w2, y))
        if improv_checker > 0:
            return_w = current_w
    plot([0,-w0/w1],[-w0/w2,0])
    print("MSE: ", mean_square_error(x, w0, w1, w2, y))
    return return_w
```

Listing 5: Gradient Descent

The basic idea is to specify a precision(in this case, mean squared error) and maximum number of iterations we would like to perform. We use the summed_gradient() function and the step size to update the weight in order to reduce the mse. The step size is a dynamic value obtained by using Armijo algorithm, which will be illustrated in section 3.d.
The following graph is a sample result using the gradient_descent() function on the example in section 2.b.2.:



(b)

Since we need to plot the objective function, we revised the code in section 3.a by adding a return element *mse_list*, which is a list storing the mean squared error after each iteration.

```
def plot_gradient_descent(a, b, x, w0, w1, w2, y):
    df = read_data()
    versicolor = df[df['species']=='versicolor']
    virginica = df[df['species']=='virginica']
    plt.plot(versicolor["petal_length"], versicolor["
   petal_width"], '*',label="versicolor", color = 'orange')
    plt.plot(virginica["petal_length"], virginica["
   petal_width"], '+',label="virginica", color = 'blue')
    # plot the decision boundary
    plt.xlabel("petal length (cm)")
    plt.ylabel("petal width (cm)")
    plt.plot([0, -w0/w1],[-w0/w2, 0], label="initial boundary
   ", color='purple')
    mid_w, mse_list = gradient_descent(1, 0.5, 10000, 0.001,
   x, w0, w1, w2, y)
    mid_w0, mid_w1, mid_w2 = mid_w[0], mid_w[1], mid_w[2]
    plt.plot([0, -mid_w0/mid_w1],[-mid_w0/mid_w2, 0], label="
   middle boundary", color='skyblue')
    fin_w, mse_list = gradient_descent(1, 0.5, 20000, 0.001,
   x, w0, w1, w2, y)
    fin_w0, fin_w1, fin_w2 = fin_w[0], fin_w[1], fin_w[2]
    plt.plot([0, -fin_w0/fin_w1],[-fin_w0/fin_w2, 0], label="
   final boundary", color='olive')
```

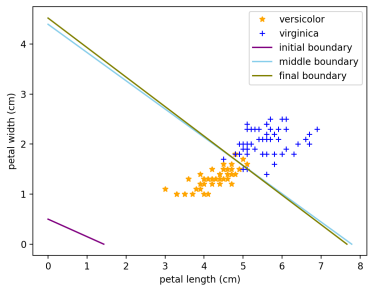
```
    # plot the change of objective function
    plt.plot(mse_list)
    plt.xlabel("Number of Iterations")
    plt.ylabel("Objective Function(MSE)")
    plt.legend()
    plt.show()
```

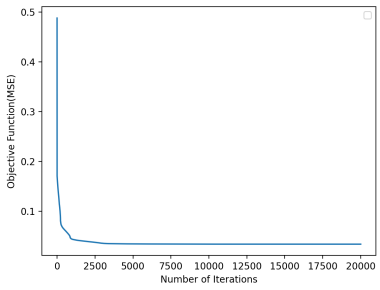
Listing 6: Visualize the Change of Decision Boundary and Objective Function

The demonstration of the correctness of this code is in section 3.c.

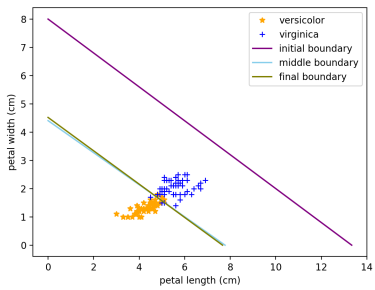
(c)

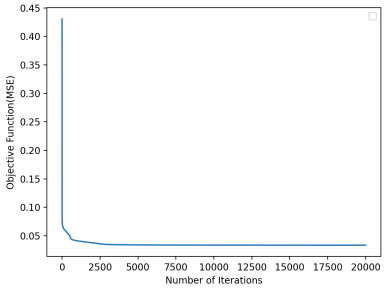
In this illustration, we select the initial weight to be $[-1, 0.7, 2]$, which is the same example we selected in section 2.b.2. We also select the parameter $a = 1$ and $b = 0.5$. The middle boundary for the objective function occurs when the number of iterations is 10000(since the total number of iterations we select is 20000).





To further illustrate the correctness of our code, we select another set of initial weight $[-8, 0.6, 1]$. Select $a = 1$ and $b = 0.5$ and run for 20000 iterations, we have:





(d)

The choice of gradient step size is important because if the step size is too large, the algorithm will not converge and jump around the minimum; if the step size is too small, the convergence will take place pretty slowly. Therefore, instead of using a constant step size, we decided to adjusted our step size while running the algorithm. To choose the step size, we used back-tracking inexact line search with Armijo rule(please refer to the article: Back-tracking with Armijo rule).
The basic idea is to start with a relatively large step size α_0 and subtract $\beta^k\alpha$ for $k = 1$ to n until a stopping condition(in this case, smaller than a predefined mse) is met. Choose the largest step size for α. The stopping step can be expressed as the following formula:

$$f(\mathbf{x} + \alpha\mathbf{d}) \leq f(\mathbf{x}) + \beta\alpha\nabla f(\mathbf{x})^T\mathbf{d}$$

, where f is the objective function, d is the descent direction, α and β are the parameters of Armijo algorithm. For the problem of iris dataset, we can rewrite the above function as following in order to update the step size:

$$f(\mathbf{w} - \alpha\nabla\mathbf{w}) \leq f(\mathbf{w}) - \beta\alpha\nabla f(\mathbf{x})^T f(\mathbf{x})$$

, where w is the weight. Since β is between 0 and 1, while the mean squared error converges to the minimum, the step size decreases.
The following code implements the Armijo algorithm:

```
1 def armijo_updating(a, b, x, y, w0, w1, w2):
2     step_size = a
3     gradient = summed_gradient(x, w0, w1, w2, y)
4     while mean_square_error(x, w0 - (step_size * gradient[0])
    , w1 - (step_size * gradient[1]), w2 - (step_size *
    gradient[2]), y) > mean_square_error(x, w0, w1, w2, y) -
    (0.5 * step_size * la.norm(gradient) ** 2):
5         step_size = step_size * b
6     return step_size
```

Listing 7: Armijo Updating Step Size

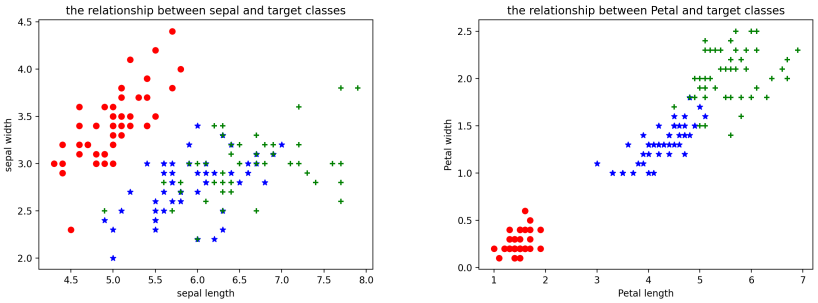
The step size does not assure us to find the global minimum since the step size could be too large for the final step(but by using the Armijo Updating algorithm, it performs much better than just select a constant stepping size). Note in the *gradient_descent* function, if the final iteration makes an improvement, then we will adopt it; otherwise, we won't. This ensures us to be as close to the global minimum of the objective function as possible.

(e)

The stopping criterion(in our code, it is represented as precision) we chose is 0.001 for precision(or tolerance of error) and 20000 for the number of iterations because it is a good balance between the time consumed to run the code and the precision of the decision boundary. As we illustrated in the graphs above, a precision of 0.001 is sufficient to classify the categories relatively accurate. If we select a much smaller precision or a much larger number of maximum iterations, it will take too much time to make just a little bit of improvement. As we can see from the graph of objective functions, after a certain threshold of number of iterations, the improvement we made is actually trivial.

Problem 4(Extra Credit):

The following two graphs visualize the classification of iris dataset with 4 attributes:

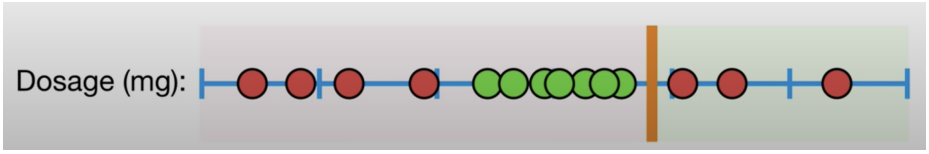


To categorize 3 classes by 4 attributes, we used **Support Vector Machine(SVM)**, which is a tool to find a hyperplane in a multi-dimensional space that distinctly classifying the data points.

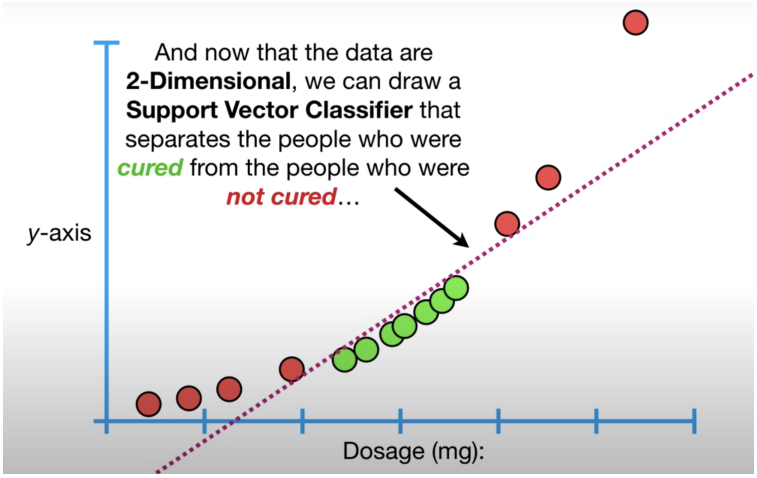
4.1 Brief Explanation of the Idea Behind SVM

The general idea is to start with data from a lower dimension and map our space to a higher dimension in order to find a hyperplane to classify the data points. Here is a simple example: the following graph is a number axis with 1-dimension, where the red points and the green points are two different classes we would like to classify and the orange line is the decision boundary

we compute to seperate the data points. However, in this case, it is hard to find a boundary which precisely seperates these two classes since no matter where we put the decision boundary, the number of misclassifications would be large.



In this case, we will add another dimension by squaring the coordinates as y, as illustrated in the following graph:



The transformation to a higher dimension makes it easier and more accurate for us to find a decision boundary.

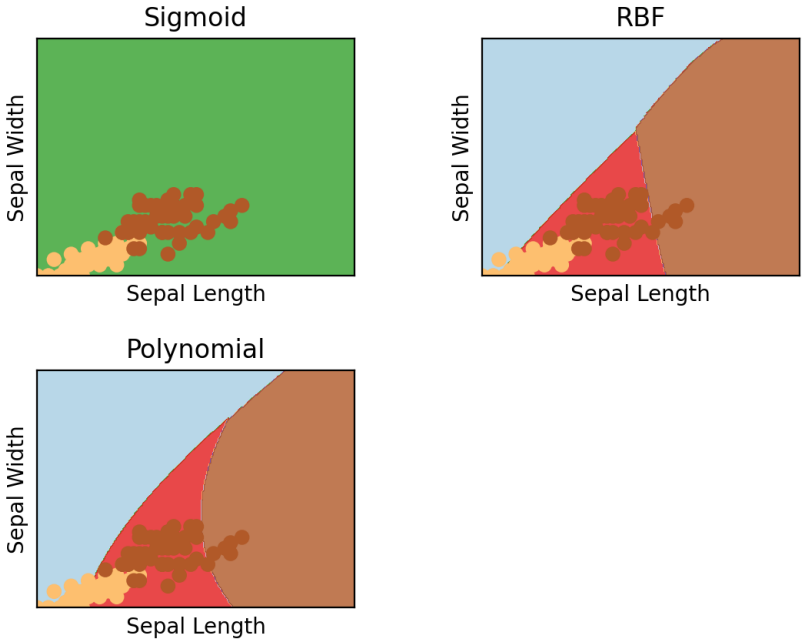
Note that in the example, we randomly choose a polynomial function to transform the data from 1-dimension to 2-dimension(by squaring their coordinates). There are actually more than one way to transform those data points. In SVM, we use a **kernel function** to specify how the data will be transformed to a higher dimension. From the link Kernel Explain, we found an excellent formal definition of kernel: suppose we have a mapping $\Phi : R^n \to R^m$ that brings our vectors in R^n to some feature space R^m. Then the dot product of x and y in this space is $\Phi(x)^T\Phi(y)$. A kernel is a function k that corresponds to this dot product, i.e. $k(\mathbf{x}, \mathbf{y}) = \Phi(\mathbf{x})^T\Phi(\mathbf{y})$. In the mini tutorial, we will briefly explore how the selection of different kernel function will affect the accuracy of classification.

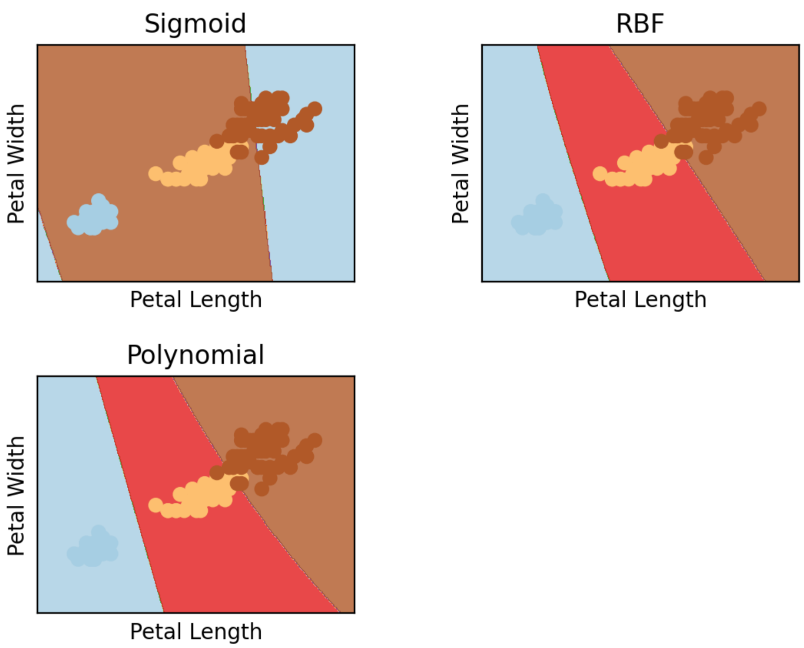
4.2 Code Implementation

kernel='sigmoid': $k(\mathbf{x}, \mathbf{y}) = tanh(\gamma \cdot \mathbf{x}^{\mathbf{T}}\mathbf{y} + \mathbf{r})$
kernel='rbf': $k(\mathbf{x}, \mathbf{y}) = e^{-\gamma \cdot \|\mathbf{x}-\mathbf{y}\|^2}$
kernel='poly': $k(\mathbf{x}, \mathbf{y}) = (\gamma \cdot \mathbf{x}^{\mathbf{T}}\mathbf{y} + \mathbf{r})$

4.3 Visualization of Output

The following two graphs are the output of putting iris dataset into SVM with kernel function equals to 'sigmoid', 'rbf', and 'poly'. As we can see from the data points and the decision boundaries, RBF and Polynomial classify classes much better than Sigmoid. By running an accuracy test, we found Sigmoid only correctly classifies 24.4% of sepal data and 11.1% of petal data. RBF and Polynomial correctly classify 80% of the sepal data and 97.8% of the petal data.





The output of test is as below:

```
Sigmoid Result(Sepal):  [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2]
RBF Result(Sepal):  [1 1 0 2 0 2 0 2 2 1 1 2 1 2 1 0 1 1 0 0 1 1 0 0 2 0 0 2 1 0 2 1 0 1 2 1 0
 1 1 1 2 0 2 0 0]
Polynomial Result(Sepal)):  [1 1 0 2 0 2 0 2 2 1 1 2 1 2 1 0 1 1 0 0 1 1 0 0 2 0 0 2 1 0 2 1 0 1 2 1 0
 1 1 1 2 0 2 0 0]
Accuracy of Sigmoid(Sepal):  0.24444444444444444
Accuracy of Sigmoid(Sepal):  0.8
Accuracy of Sigmoid(Sepal):  0.8
Sigmoid Result(Petal):  [2 2 2 0 2 0 2 2 2 2 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 2 2 2 0 2 2
 2 2 2 0 2 2 2 2]
RBF Result(Petal):  [2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0
 2 1 1 2 0 2 0 0]
Polynomial Result(Petal):  [2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0
 2 1 1 2 0 2 0 0]
Accuracy of Sigmoid(Petal):  0.1111111111111111
Accuracy of RBF(Petal):  0.9777777777777777
Accuracy of Polynomial(Petal):  0.9777777777777777
```

4.4 References

Support Vector Machines, Clearly Explained!!!