P3

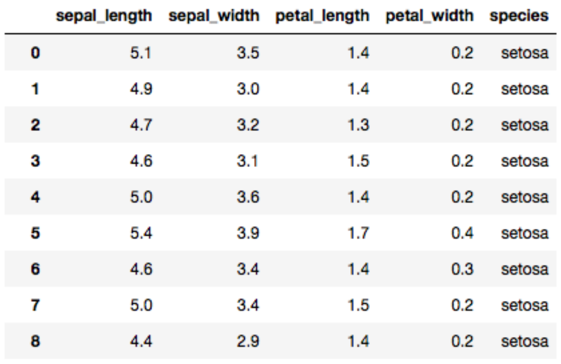
Adrian Guzman afg30

December 2019

1

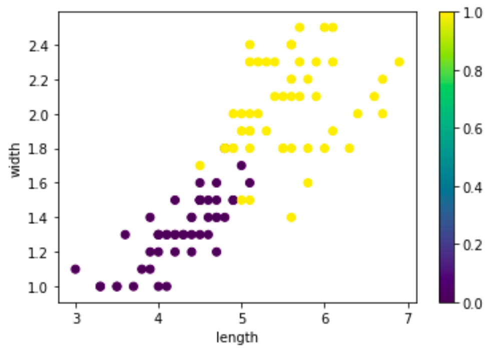
1.1

I used pandas and matplotlib to load and plot the data respectively



	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
5	5.4	3.9	1.7	0.4	setosa
6	4.6	3.4	1.4	0.3	setosa
7	5.0	3.4	1.5	0.2	setosa
8	4.4	2.9	1.4	0.2	setosa

I then plotted the versicolor and virginica below, with yellow representing virginica and purple representing versicolor



1.2

This function takes in the weight vector and petal feature vector and returns the logistic evaluation value using the function from the textbook.

```
def logistic_function(w, x):
    return 1 / (1+(np.e**(-1 * np.dot(w, x))))
```

1.3

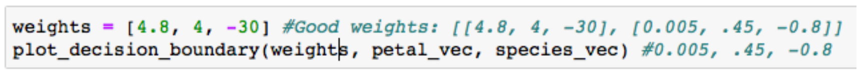
This function takes in weights and solves for $x_2 = mx_1 + b$ by setting the decision boundary line $w_1x_1 + w_2x_2 + w_3 = 0$

```
def find_decision_boundary(x1, w1, w2, w3):
    return ((-x1 * w1) - w3) / w2
```

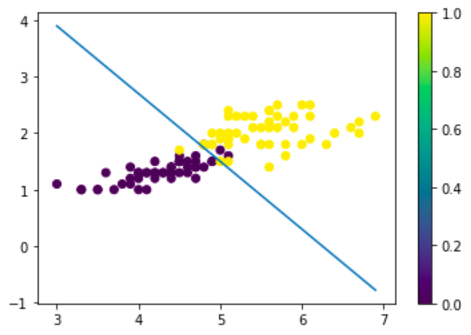
The following function takes in a weight vector, a vector of petal lengths/widths, and the class vector. It computes the decision boundary line from the given weights and plots the line of the scatterplot of the original data

```
def plot_decision_boundary(w, vec1, vec2):
    x1 = np.linspace(3,6.9,100)
    x2 = find_decision_boundary(x1, w[0], w[1], w[2])
    plt.plot(x1, x2)
    plt.scatter(vec1.values[:,0], vec1.values[:,1], c=vec2.values)
    plt.colorbar()
    plt.show()
```

Choosing weights so the decision boundary is a rough estimate we get the following plot:



```
weights = [4.8, 4, -30] #Good weights: [[4.8, 4, -30], [0.005, .45, -0.8]]
plot_decision_boundary(weights, petal_vec, species_vec) #0.005, .45, -0.8
```

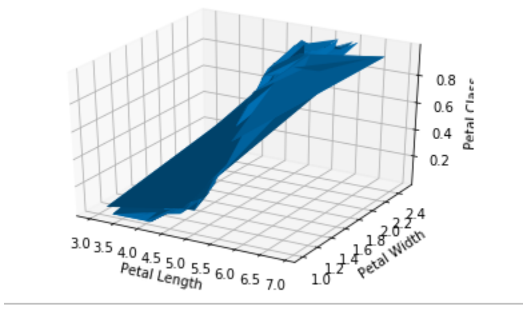


1.4

The following code applies the classifier to each petal and saves the value to be plotted

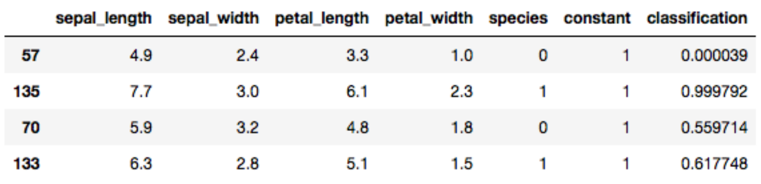
```
test_vec = []
for petal in data23.values:
    hyp = logistic_function(weights, [petal[2], petal[3], petal[5]])
    test_vec.append(hyp)
data23["classification"] = test_vec
data23
```

The plot looks like so:

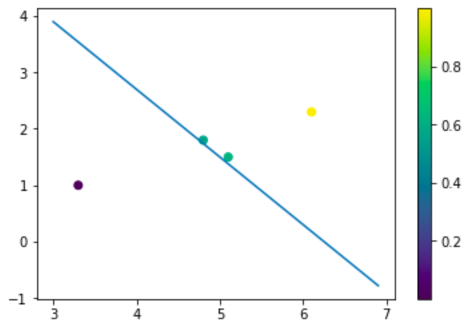


1.5

The classification column represents the class value assigned by the logistic classifier. From the graph, it is clear that when the species is far from the decision boundary, the species and classification are both very close to 0 or 1. When the petal species is close to the decision boundary, it is classified as around 0.5, because it is ambiguous. These results can be seen clearly in the graph, where blue dots have a value around 0.5.



	sepal_length	sepal_width	petal_length	petal_width	species	constant	classification
57	4.9	2.4	3.3	1.0	0	1	0.000039
135	7.7	3.0	6.1	2.3	1	1	0.999792
70	5.9	3.2	4.8	1.8	0	1	0.559714
133	6.3	2.8	5.1	1.5	1	1	0.617748



2

2.1

This function computes the MSE given feature vectors, weights, and classes

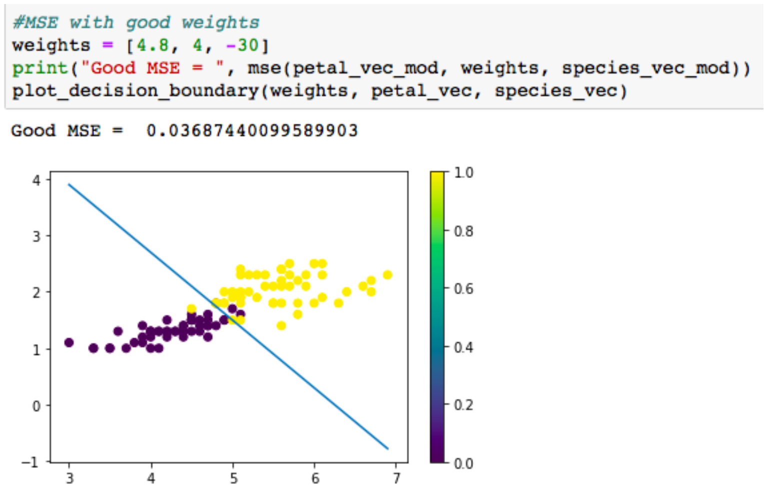
```
def mse(vectors, weights, classes):
    sse = 0
    for petal,p_class in zip(vectors, classes):
        sse += (logistic(weights, petal) - p_class)**2
    return sse/len(classes)
```

2.2

This code retrieves the vectors to be plotted

```
petal_vec_mod = data23[['petal_length', 'petal_width', 'constant']].values
species_vec_mod = data23['classification']
```

MSE with good weights



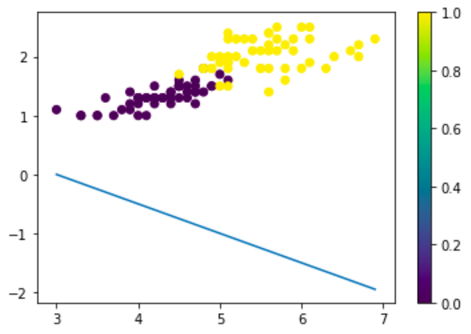
```
#MSE with good weights
weights = [4.8, 4, -30]
print("Good MSE = ", mse(petal_vec_mod, weights, species_vec_mod))
plot_decision_boundary(weights, petal_vec, species_vec)
```

```
Good MSE =  0.03687440099589903
```

MSE with bad weights

```
#MSE with bad weights
weights = [1, 2, -3]
print("Bad MSE = ", mse(petal_vec_mod, weights, species_vec_mod))
plot_decision_boundary(weights, petal_vec, species_vec)
```

```
Bad MSE =  0.4150367399404306
```



2.3

The objective function is:

$$MSE = \frac{1}{N}\sum_{1}^{N}(logistic - class)^2 \tag{1}$$

Plug in the logistic function

$$MSE = \frac{1}{N}\sum_{1}^{N}(\frac{1}{1+e^{-w\cdot x}} - c_n)^2 \tag{2}$$

Derive with respect to the weights

$$\frac{\partial MSE}{\partial w_i} = \frac{2}{N}\sum_{1}^{N}(\frac{1}{1+e^{-w\cdot x}} - c_n)x_{i,n}$$

Now plugging in the three weights to the above derivative we get the following equations:

$$\frac{\partial MSE}{\partial w_1} = \frac{2}{N}\sum_{1}^{N}(\frac{1}{1+e^{-w_1\cdot x_1}} - c_n)x_{1,n}$$

$$\frac{\partial MSE}{\partial w_2} = \frac{2}{N}\sum_{1}^{N}(\frac{1}{1+e^{-w_2\cdot x_2}} - c_n)x_{2,n}$$

$$\frac{\partial MSE}{\partial w_0} = \frac{2}{N}\sum_{1}^{N}(\frac{1}{1+e^{-w_0\cdot 1}} - c_n) * 1$$

And the gradient is the vector of all partial derivatives

$$\nabla MSE = \begin{bmatrix} \frac{\partial MSE}{\partial w_1} \\ \frac{\partial MSE}{\partial w_2} \\ \frac{\partial MSE}{\partial w_0} \end{bmatrix}$$

2.4

$$\frac{\partial MSE}{\partial w_1} = \frac{2}{N}\sum_{1}^{N}(\frac{1}{1+e^{-w_1 \cdot x_1}} - c_n)x_{1,n}$$

$$\frac{\partial MSE}{\partial w_2} = \frac{2}{N}\sum_{1}^{N}(\frac{1}{1+e^{-w_2 \cdot x_2}} - c_n)x_{2,n}$$

$$\frac{\partial MSE}{\partial w_0} = \frac{2}{N}\sum_{1}^{N}(\frac{1}{1+e^{-w_0 \cdot 1}} - c_n) * 1$$

$$\nabla MSE = \begin{bmatrix} \frac{\partial MSE}{\partial w_1} \\ \frac{\partial MSE}{\partial w_2} \\ \frac{\partial MSE}{\partial w_0} \end{bmatrix}$$

The above partial derivatives are expressed in scalar form, the below are expressed in vector form.
Here w is the vector of weights

$$\frac{\partial MSE}{\partial \mathbf{w}} = \frac{2}{N}\sum_{1}^{N}(\frac{1}{1+e^{-w \cdot x}} - c_n)x_n$$

$$\nabla MSE = \begin{bmatrix} \frac{\partial MSE}{\partial \mathbf{w}} \end{bmatrix}$$

2.5

This function calculates the gradient for a given input vector with given weights.

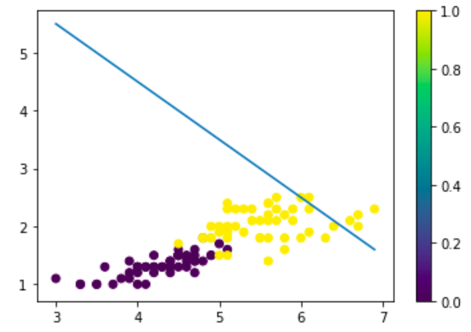
```
def gradient_function(w, weights, p_input_vec, p_output_vec, petal_vec,
    learning_p=0.05):
    #Accumulator for change in start weight
    weight_cumulator = 0

    #Iterates through each petal where
    #  z - feature value of current petal
    #  y - given class of current petal
    #  z - set of features for the current petal
    for x, y, z in zip(p_input_vec, p_output_vec, petal_vec):
        weight_cumulator += learning_p*(y-logistic_function(weights, z)) *
            logistic_function(weights, z)*(1-logistic_function(weights, z)) * x

    return weight_cumulator
```

To illustrate this working I will first plot a decision boundary with bad weight parameters.

```
weights = [4, 4, -34] #Good weights: [[4.8, 4, -30]]
plot_decision_boundary(weights, petal_vec, species_vec) #0.005, .45, -0.8
```



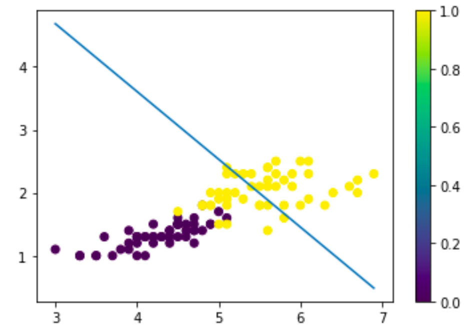
Now we use the gradient function above to adjust the weight parameters with a small step by using the following code:

```
weights[0] += gradient_function(weights[0], weights,
    p_input1_vec, p_output_vec, petal_vec_mod)
weights[1] += gradient_function(weights[1], weights,
    p_input2_vec, p_output_vec, petal_vec_mod)
weights[2] += gradient_function(weights[2], weights,
    p_input3_vec, p_output_vec, petal_vec_mod)
```

And now with the updated weights we plot the decision boundary again:

```
print("New Weights: ", weights)
plot_decision_boundary(weights, petal_vec, species_vec) #0.005, .45, -0.8
```

```
New Weights:  [4.607301216308777, 4.280355550457671, -33.867715756445456]
```



And we can see the decision boundary is slightly better than previously

3

3.1

The code for training the neural network can be found in the attached code file in the function train_network
The function can also be found at the bottom of the report

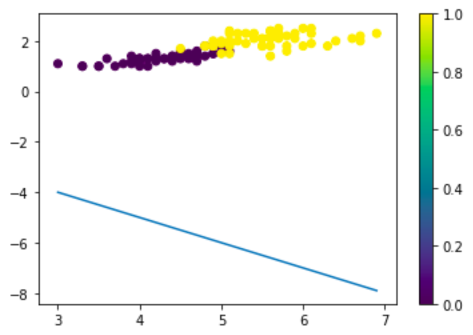
3.2

The code for plotting the progress of the neural network training can also be found in the train_network function
The function can also be found at the bottom of the report

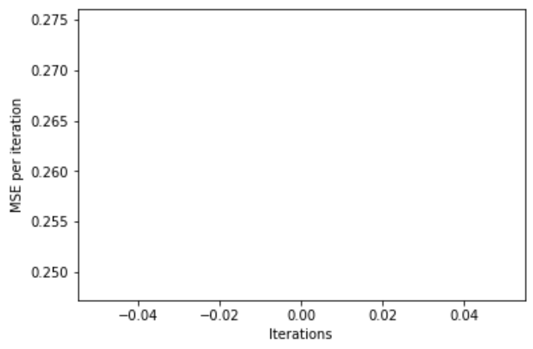
3.3

Initial Weighting

```
Initial Model weights:  [0.1, 0.1, 0.1]
```

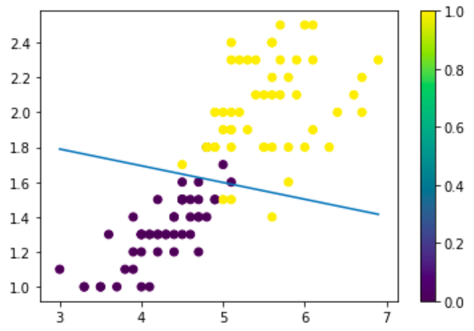


```
Model MSE over iterations:
```

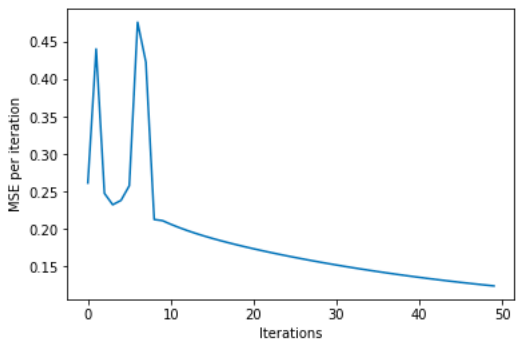


Middle Weighting

```
Middle Model weights:  [0.16771064827929086, 1.7486564456635567, -3.6320634971497503]
```

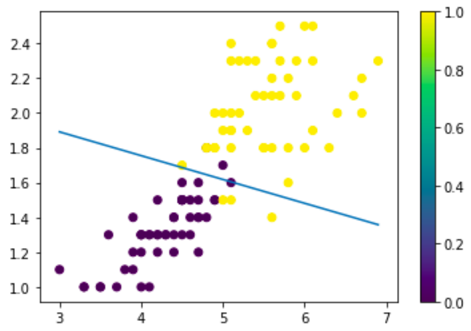


```
Model MSE over iterations:
```

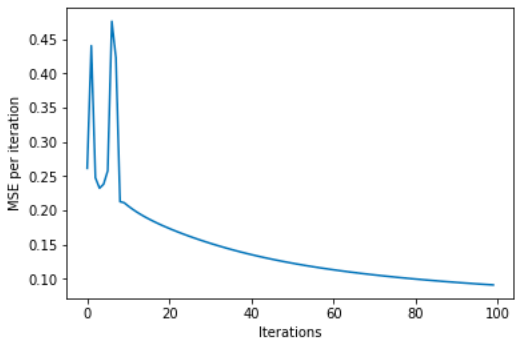


End Weighting

```
Model achieved weights:  [0.33929099213514413, 2.479167905189205, -5.70799981426987]
```



```
Model MSE over iterations:
```



You can see the decision boundary becoming more accurate as the network is learning better parameters. Also the MSE is decreasing as the number of iterations is increasing, meaning the error between the classification and the correct class is decreasing, meaning the model is becoming more accurate.

3.4

I chose the gradient step size to be 0.05. I experimented with some other values such as 0.001 and 0.8 but they would either converge too quickly and not reach optimal solutions or take too long to converge, so I settled on 0.05. This is also a number I am familiar with in my experience with confidence intervals.

3.5

There are two stopping criterion. The first is when the number of max_iterations is reached which I chose to default to 2000 if not specified because most models seemed to have converged by this point. The second stopping criterion is when the MSE is less than 0.05, which means that the model is accurate enough to be trusted, and has produced a good enough decision boundary.

4 Extra Credit

See function in code file

4.1

5 Functions for Reference

```
def train_network(data, weights_init, max_iter=2000, mse_threshold=0.05):
    #Trains a neural network using a gradient descent function

    #For plotting decision boundary
    petal_vec = data[['petal_length', 'petal_width']]
    species_vec = data['species']

    #For storing MSE per iteration
    cur_mse = 0
    mse_vec = []

    #For storing the weights to plot progress
    weights = weights_init
    weights_dict = {}

    p_input1_vec = data['petal_length']
    p_input2_vec = data['petal_width']
    p_input3_vec = data['constant']
    p_output_vec = data['species']
    petal_vec_mod = data[['petal_length', 'petal_width', 'constant']].values

    for i in range(max_iter):
        #Calculate and store MSE
        cur_mse = mse(data[['petal_length', 'petal_width', 'constant']].values,
            weights, p_output_vec)
        mse_vec.append(cur_mse)

        #If mse reaches the given threshold, exit
        if (cur_mse < mse_threshold):
            break

        #Increments each weight by computing gradient change for corresponding
            feature
        weights_dict[str(i)] = [weights[0], weights[1], weights[2]]
        weights[0] += gradient_function(weights[0], weights, p_input1_vec,
            p_output_vec, petal_vec_mod)
        weights[1] += gradient_function(weights[1], weights, p_input2_vec,
            p_output_vec, petal_vec_mod)
        weights[2] += gradient_function(weights[2], weights, p_input3_vec,
            p_output_vec, petal_vec_mod)

    #Plot initial decision boundary and MSE vector
    print("Initial Model weights: ", weights_dict['0'])
    plot_decision_boundary(weights_dict['0'], petal_vec, species_vec)

    print("Model MSE over iterations:")
    plt.plot(list(range(0,1)), mse_vec[0])
    plt.xlabel('Iterations')
    plt.ylabel('MSE per iteration')
    plt.show()

    #Plot decision boundary and MSE vector during middle of learning
    middle_index = int(len(mse_vec)/2)
    print("Middle Model weights: ", weights_dict[str(middle_index)])
    plot_decision_boundary(weights_dict[str(middle_index)], petal_vec,
        species_vec)

    print("Model MSE over iterations:")
    plt.plot(list(range(0,middle_index)), mse_vec[:middle_index])
    plt.xlabel('Iterations')
    plt.ylabel('MSE per iteration')
    plt.show()

    #Plot final decision boundary and MSE vector
    print("Model achieved weights: ", weights)
    plot_decision_boundary(weights, petal_vec, species_vec)

    print("Model MSE over iterations:")
    plt.plot(list(range(0,len(mse_vec))), mse_vec)
    plt.xlabel('Iterations')
    plt.ylabel('MSE per iteration')
    plt.show()
```