CSDS 391 P2 - Maryam Iqbal and Mohamed Salah
We submitted both a .py file and a Jupyter Notebook .ipynb for your convenience

Exercise 1: Linear decision boundaries

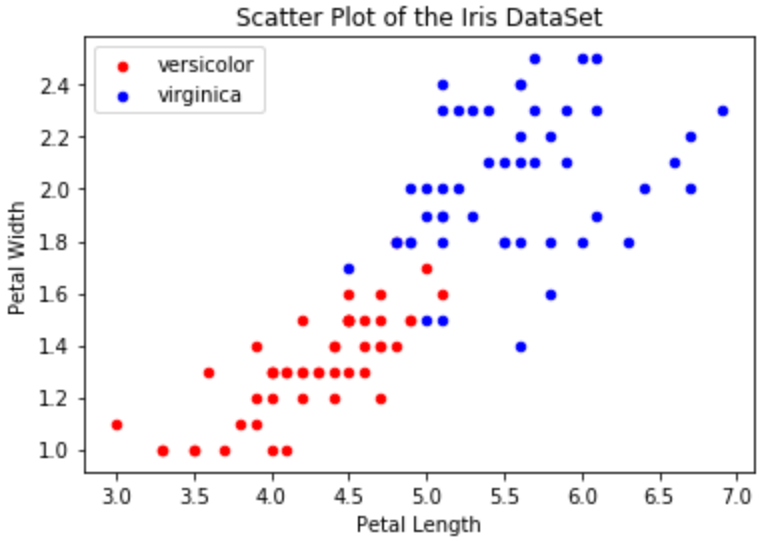
a) We first started by importing the necessary python libraries we will use throughout the project as well as loading and cleaning the iris data set as shown from the code below:

```
#import pandas, numpy and matplotlib libraries
import random as r
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sklearn as skl
import math
from mpl_toolkits.mplot3d import Axes3D

#import the iris dataset from and store it as dataSet
dataSet = pd.read_csv("irisdata.csv")

#remove the 1st iris class data
dataSet = dataSet[dataSet.species != "setosa"]

#add the column 'class' with values 0 for versicolor and 1 for virginica
irisClass = []
for item in dataSet.species:
   if item == "versicolor":
       irisClass.append(0)
   elif item == "virginica":
       irisClass.append(1)

dataSet['class'] = irisClass
```

In the code above, we cleaned the dataset by removing the data for the first iris class as this assignment does not require it. We also labeled versicolor and virginica with values of 0 and 1 respectively. Next step we took was to plot the dataset using the matplotlib library. The following code shows how the scatter plot of the iris dataset was achieved.

```
# Exercise 1, part a
df = pd.DataFrame(dict(x=dataSet['petal_length'], y= dataSet['petal_width'],
label=dataSet['species']))
```

```
colors = {"versicolor":'red', "virginica":'blue'}
fig, ax = plt.subplots()
grouped = df.groupby('label')
for key, group in grouped:
   group.plot(ax=ax, kind='scatter', x='x', y='y', label=key, color=colors[key])
plt.title("Scatter Plot of the Iris DataSet")
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.show()
```



b) In order to develop a function that indicates whether an iris belongs to the second or third class, we first developed a logistic regression function that computes the logistic regression given a list of weights consisting of (w_0, w_1, w_2) and (x1, x2) where x1 and x2 represent the value for petal length and petal width.

```
# Exercise 1, part b

# LogisticReg function calculates the logistic regression given a list of weights,
x_1 and x_2
def logisticReg(w,x_1,x_2):

   z = w[0] + w[1]*x_1 + w[2]*x_2
   logisticRegression = 1 / (1 + np.exp(-(z)))

   return logisticRegression
```

```
# neural Network function returns 1 if is the logistic regression > 0.5 and 0
otherwise
def neuralNetwork(logisticRegression):
   if logisticRegression > 0.5:
       output = 1
   elif logisticRegression <= 0.5:
       output = 0

   return output
```

The code works by first computing the value $z = w_0 + w_1x_1 + w_2x_2$ and then computing the logistic regression value:

$$(z) = \frac{1}{1+e^{-z}}$$

In order to indicate which class that iris belongs to, we then pass the output of our logisticReg function as the input to the neuralNetwork function.The function works by returning 1 (indicating that the iris belongs to the 3rd class) if the logistic regression value is bigger than 0.5 and returning 0 otherwise.

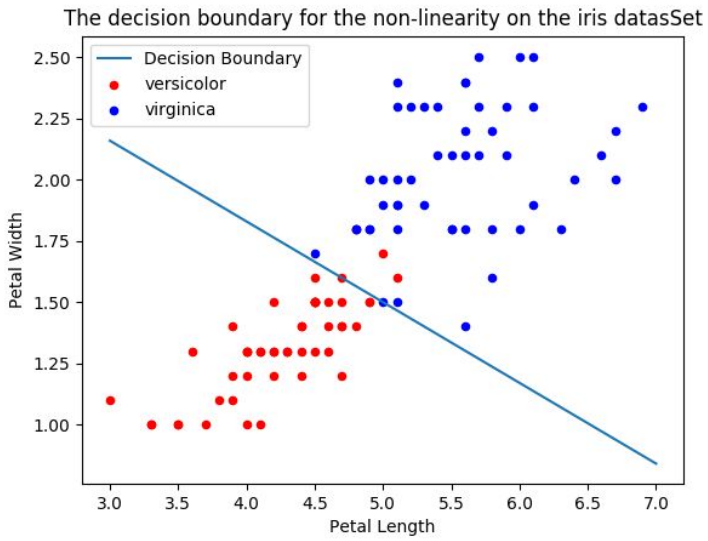
c) After experimenting with different sets of weights, we found that the following roughly separates the two classes: $w_0 = -3.15$, $w_1 = 0.33$, $w_2 = 1$

After plotting the data set like was done in part a, we developed a function to plot the decision boundary. Our decisionBoundary function takes a list of weights as input to compute the y_values for the x_values in the range (3,7) [the petal length range]. Each y_value is calculated using the following formula:

$y = (-w_0 - w_1x) \div w_2$

```
# Exercise 1, partC

# List of weights, we found roughly separates the two classes
w = [-3.15, 0.33, 1]

#plotting the scatter plot of the data set
df = pd.DataFrame(dict(x=dataSet['petal_length'], y= dataSet['petal_width'],
label=dataSet['species']))
colors = {"versicolor":'red', "virginica":'blue'}
fig, ax = plt.subplots()
grouped = df.groupby('label')
for key, group in grouped:
   group.plot(ax=ax, kind='scatter', x='x', y='y', label=key, color=colors[key])

# decision Boundary function that plots the line representing the decision boundary
given a list of weights
def descisonBoundary (w):
```

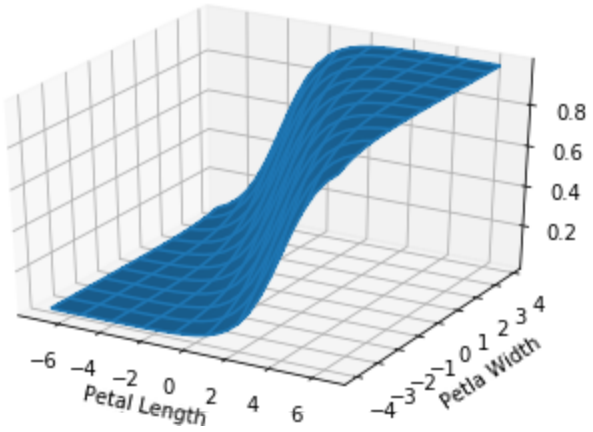
```
    x_values = np.array(range(3,8))
    y_values = []
    for x in x_values:
        y = (-w[0]-w[1]*x)/w[2]
        y_values.append(y)

    plt.plot(x_values, y_values, label='Decision Boundary')
    plt.legend()

descisonBoundary(w)
plt.title("The decision boundary for the non-linearity on the iris datasSet")
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.show()
```



d) We used the 3D functions (plot_surface and plot_wireframe) from the matplotlib 3D toolkit to plot the output of our neural network over the input space as shown in the following code and plot.

```
# Exercise 1, part d

# 3D plot showing the input of our simple neural network over the input space
fig = plt.figure()
ax = fig.gca(projection='3d')

#Creating the ranges for X_1 and X_2
X_1 = np.arange(-6,6,0.2)
```

```
X_2 = np.arange(-5,5,0.2)

#Using the np.meshgrid method to prepare the X_1 and X_2 arrays to be plotted in 3D
using the plot_surface
X, Y = np.meshgrid(X_1, X_2)

#Calculating the Z values
Z = logisticReg(w,X,Y)

#Plotting the 3D surface
surf = ax.plot_surface(X, Y, Z,linewidth=0, antialiased=False)

#Plotting the wireframe overlay
ax.plot_wireframe(X, Y, Z,rcount = 10, ccount = 10)

plt.title('The input of the neural network over the iris dataset input space')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
```



e) We tested the output of our simple neural network through 4 examples.

```
# Exercise 1, part e

test1 = neuralNetwork(logisticReg(w,4.2,1.3))
print(test1)

test2 = neuralNetwork(logisticReg(w,5.1,2.4))
print(test2)
```

```
test3 = neuralNetwork(logisticReg(w,5.0,1.5))
print(test3)

test4 = neuralNetwork(logisticReg(w,5.0,1.7))
print(test4)
```

Test 1 was an unambiguous example where the iris had a petal length of 4.2 and petal width of 1.3 and belonged to the versicolor (0) class which our neural network classified correctly. Test 2 was another unambiguous example but this time it belonged to the virginica (1) class with a petal length of 5.1 and a petal width of 2.4 which our neural network classified correctly. Tests 3 and 4 were right on the decision boundary with test 3 having a petal length of 5.0 and petal width of 1.5 and test 4 having a petal length of 5.0 and petal width of 1.7. Our neural network managed to classify the iris in test 3 correctly: virginica (1). However, it failed to classify the iris in test 4 correctly as our network classified it as virginica (1) while it in fact belongs to class 0, versicolor.

Exercise 2: Neural networks

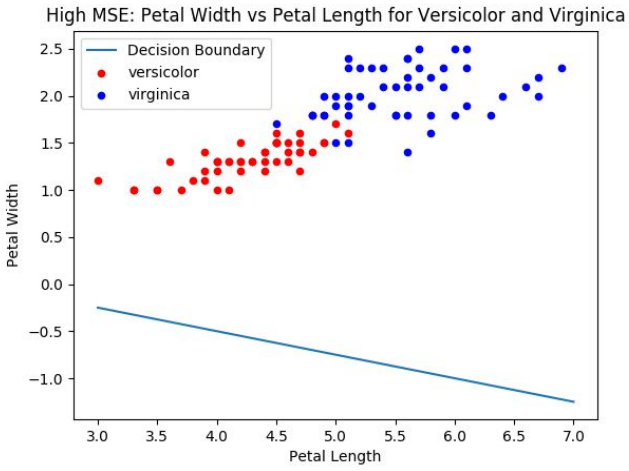
a) The code for finding the mean squared error calculations is given below. As per the question, the inputs are the data vectors i.e. petal length and petal width information taken from the iris data set points, and the parameters defining the neural network (w_0, w_1, w_2).

```
# Q2 Part A code
# Data sets being used
y_data = dataSet['class']
X_data = dataSet[['petal_length','petal_width']]

def mean_squared_error(w, X, y):

# Input Feature Values into a list
 petalLength = X['petal_length'].tolist()
 petalWidth = X['petal_width'].tolist()
 # Outputs of neural network for each example
 nrow = len(petalLength)
 outputs = [0]*nrow

#neural network logistic function output for each point
 for i in range(nrow):
   outputs[i] = logisticReg(w, petalLength[i], petalWidth[i])

 difference = np.subtract(outputs,y)
 # Compute the mean between the outputs and the labels (0,1)
```

```
 result = np.mean(np.power(difference,2))

 # Return result
 return result
```

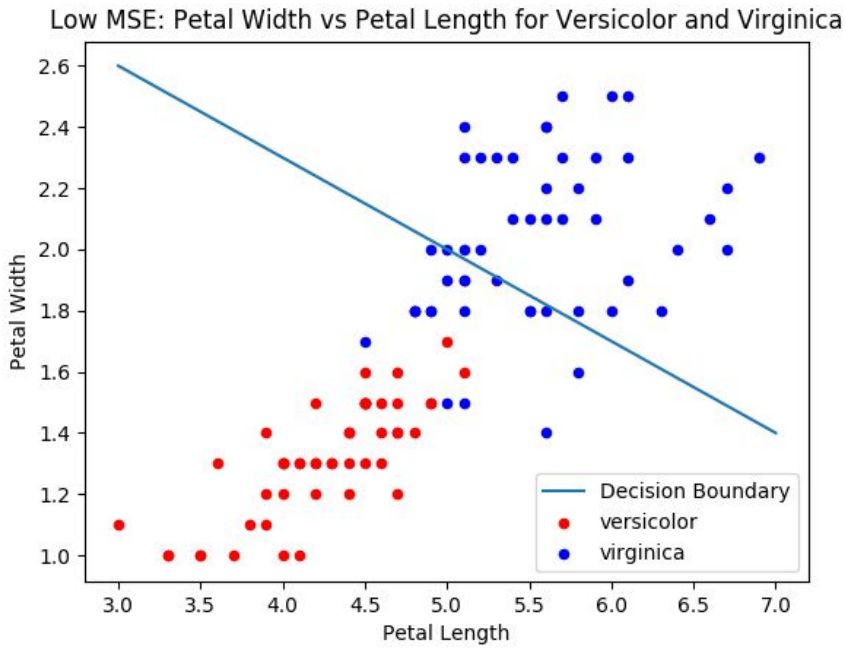
b) Having looked at the many values, we've chosen the two sets of values, one that reveals a high error and another that shows a low error. The high error values are [0.5,-0.25,-1] and the resulting error is 0.45726682017365766 where the line is not touching any data point. The code and plot to show this boundary is below.

```
# Q2 part B code

# High error values handpicked
wH = [-0.5,0.25,1]
print ('High Mean Square Error:', mean_squared_error(wH, X_data, y_data))

df = pd.DataFrame(dict(x=dataSet['petal_length'], y= dataSet['petal_width'],
label=dataSet['species']))
colors = {"versicolor":'red', "virginica":'blue'}
fig, ax = plt.subplots()
grouped = df.groupby('label')
for key, group in grouped:
   group.plot(ax=ax, kind='scatter', x='x', y='y', label=key, color=colors[key])

descisonBoundary(wH)
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.title('High MSE: Petal Width vs Petal Length for Versicolor and Virginica')
plt.show()
```



The low error values are [-3.5,0.3,1] and the resulting error is 0.1520260495513365 where the line is now near the centre of the plot with some values still below the boundary. The code and plot to show this boundary is below.

```
# Low error values
wL = [-3.5,0.3,1]
print ('Low Mean Square Error:', mean_squared_error(wL, X_data, y_data))

#plot graph
df = pd.DataFrame(dict(x=dataSet['petal_length'], y= dataSet['petal_width'],
label=dataSet['species']))
colors = {"versicolor":'red', "virginica":'blue'}
fig, ax = plt.subplots()
grouped = df.groupby('label')
for key, group in grouped:
   group.plot(ax=ax, kind='scatter', x='x', y='y', label=key, color=colors[key])

descisonBoundary(wL)

plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.title('Low MSE: Petal Width vs Petal Length for Versicolor and Virginica')
plt.show()
```



c) The number of points in data given in iris data can be denoted as *n* which we know is 100 in our case for virginica and versicolor. So we then represent the matrix as

$$X = \begin{bmatrix} 1 & x_{11} & x_{12} \\ \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} \end{bmatrix} = \begin{bmatrix} x_1^T \\ \vdots \\ x_n^T \end{bmatrix}$$

where we can say that $x_1^T = \begin{bmatrix} 1 & x_{i1} & x_{i2} \end{bmatrix}$ and x_{i1} is the petal length while x_{i2} is the petal width. Then the corresponding labels of all points are stated below as a single vector.

$$y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

Here y_i has the value of 0 or 1 depending on the species in the point *i.* The logistic function is given as below.

$$\sigma(z) = (1 + exp(-z))^{-1}$$

Hence we can write the coefficients in a single vector as $w^T = \begin{bmatrix} w_0 & w_1 & w_2 \end{bmatrix}$ and write the mean squared error (MSE) as

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

The first derivative of the logistic function above is $\sigma'(z) = \sigma(z)\,(1 - \sigma(z))$. Then the partial derivative of MSE equation with respect to each parameter $(w_0\ w_1\ w_2)$ is

$$\frac{\partial MSE}{\partial w_0} = \frac{2}{n}\sum_{i=1}^{n}[\sigma(w^T x_i) - y_i][\sigma(w^T x_i)][1 - \sigma(w^T x_i)]$$

$$\frac{\partial MSE}{\partial w_1} = \frac{2}{n}\sum_{i=1}^{n}[\sigma(w^T x_i) - y_i][\sigma(w^T x_i)][1 - \sigma(w^T x_i)]x_{i1}$$

$$\frac{\partial MSE}{\partial w_2} = \frac{2}{n}\sum_{i=1}^{n}[\sigma(w^T x_i) - y_i][\sigma(w^T x_i)][1 - \sigma(w^T x_i)]x_{i2}$$

The values x_{i1} & x_{i2} come from the step below for the MSE equations above while w_0 leaves a value of 1.

$$\frac{\partial(w^T x_i)}{\partial w_1} = \frac{\partial}{\partial w_1}(w_0 + w_1 x_{i1} + w_2 x_{i2}) = x_{i1} \qquad \frac{\partial(w^T x_i)}{\partial w_2} = \frac{\partial}{\partial w_2}(w_0 + w_1 x_{i1} + w_2 x_{i2}) = x_{i2}$$

$$\frac{\partial(w^T x_i)}{\partial w_0} = \frac{\partial}{\partial w_0}(w_0 + w_1 x_{i1} + w_2 x_{i2}) = 1$$

Finally to get the gradient of the objective function we combine the three partial derivatives MSE to get

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

d) The result above in part c is in the scalar form and can be converted to the vector form by manipulating some of the equations. Firstly we look at the partial derivatives in part c and denote a common factor as below.

$$z_i = [\sigma(w^T x_i) - y_i][\sigma(w^T x_i)][1 - \sigma(w^T x_i)]$$

Putting this value in the result gradient value of the objective function, we get

$$\nabla_w MSE = \frac{2}{n}\begin{bmatrix} \sum_{i=1}^{n} z_i \\ \sum_{i=1}^{n} z_i x_{i1} \\ \sum_{i=1}^{n} z_i x_{i2} \end{bmatrix} = \frac{2}{n}\begin{bmatrix} 1 & \cdots & 1 \\ x_{11} & \cdots & x_{n1} \\ x_{12} & \cdots & x_{n2} \end{bmatrix}\begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix} = \frac{2}{n} X^T \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix}$$

We can expand the vectors of all z_i as below.

$$\begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix} = \begin{bmatrix} [\sigma(w^T x_1) - y_1][\sigma(w^T x_1)][1 - \sigma(w^T x_1)] \\ \vdots \\ [\sigma(w^T x_n) - y_n][\sigma(w^T x_n)][1 - \sigma(w^T x_n)] \end{bmatrix}$$
$$= \begin{bmatrix} [\sigma(w^T x_1) - y_1] \\ \vdots \\ [\sigma(w^T x_n) - y_n] \end{bmatrix} * \begin{bmatrix} [\sigma(w^T x_1)] \\ \vdots \\ [\sigma(w^T x_n)] \end{bmatrix} * \begin{bmatrix} [1 - \sigma(w^T x_1)] \\ \vdots \\ [1 - \sigma(w^T x_n)] \end{bmatrix}$$

Where we can denote the different vectors being multiplied as

$$\sigma(Xw) = \begin{bmatrix} [\sigma(w^T x_1)] \\ \vdots \\ [\sigma(w^T x_n)] \end{bmatrix}$$

Which can then let us simplify the scalar form of part c into the vector for below.

$$\nabla_w MSE = \frac{2}{n} X^T [(\sigma(Xw) - y) * \sigma(Xw) * (1 - \sigma(Xw))]$$

Where 1 is an n-dimensional vector with all 1 components and *Xw* and *y* are both vectors.

e) The code to compute the summed gradient for an ensemble of patterns is written below.

```
# Q2 part E code

def gradsum(X, y, w):
 gw0 = 0       #gradient value for w0
 gw1 = 0       #gradient value for w1
 gw2 = 0       #gradient value for w2
 m = len(X)

 # Loop through all the input data
 for i in range(m):
   pL = X['petal_length'].tolist()
   x_i1 = pL[i]
   pW = X['petal_width'].tolist()
   x_i2 = pW[i]
   lab = y.tolist()
   y_i = lab[i]

   # Output of neural network for each input i
   s = logisticReg(w, x_i1, x_i2)

   # Add the value to the sum for each gradient component
   gw0 += ((2/m)*(s-y_i)*s*(1-s))
   gw1 += ((2/m)*(s-y_i)*s*(1-s)*x_i1)
   gw2 += ((2/m)*(s-y_i)*s*(1-s)*x_i2)

 # Make list of all three values
 result = []
 result.append(gw0)
 result.append(gw1)
 result.append(gw2)

 return result
```

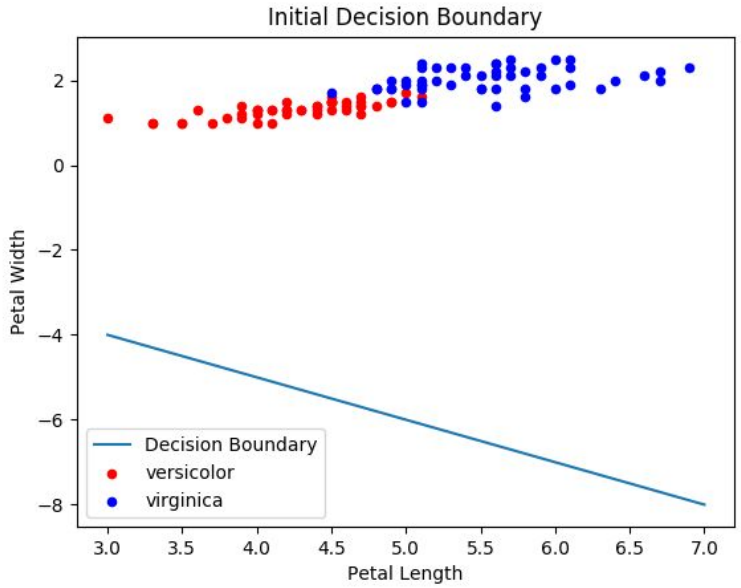
To test the above code we chose a step size of 0.01 and then show the change in decision boundary from original to a new one after 50000 iterations of the algorithm and plot both together. The boundaries were kept at original (1,1,1) values to compare the changes in values easily. The code for the graphs is indicated below. 50000 iterations were necessary to allow the decision boundary line to come near the centre of the scatter plot and have a low MSE.

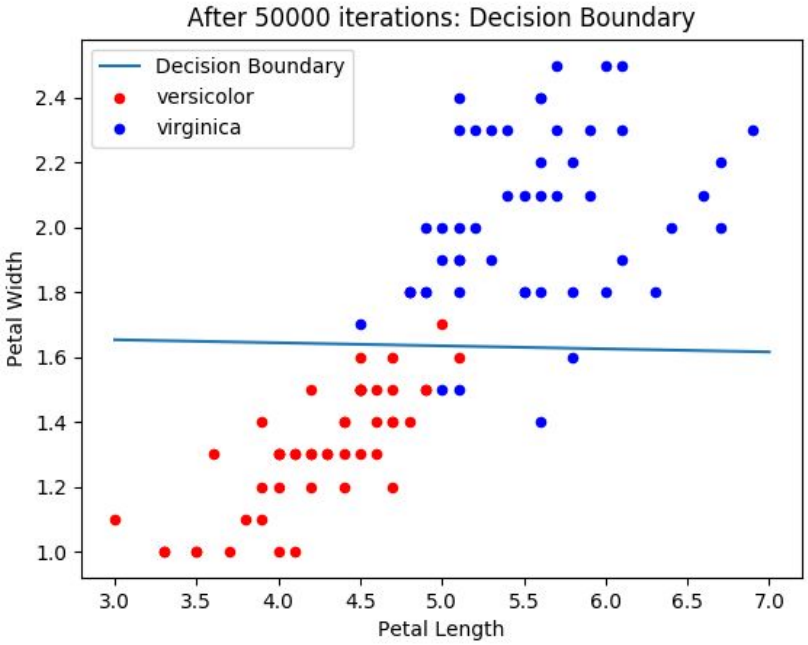
```
# Chosen original decision boundaries
ws = [1,1,1]
```

```
print(ws)
#Original MSE
print('MSE Original:', mean_squared_error(ws, X_data,y_data))

# iterate 50000 times to show change with small step size of 0.01
for _ in range(50000):
 # Step size is 0.01 here
 ws = np.subtract(ws,np.multiply(gradsum(X_data,y_data,ws),0.01))

# New decision boundaries and the MSE after iterations for a small step size
print(ws)
print('MSE New:', mean_squared_error(ws,X_data,y_data))

df = pd.DataFrame(dict(x=dataSet['petal_length'], y= dataSet['petal_width'],
label=dataSet['species']))
colors = {"versicolor":'red', "virginica":'blue'}
fig, ax = plt.subplots()
grouped = df.groupby('label')
for key, group in grouped:
   group.plot(ax=ax, kind='scatter', x='x', y='y', label=key, color=colors[key])


descisonBoundary(ws)
plt.title('Initial Decision Boundary')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.show()

# values taken from the result of the new decision boundaries as calculated above
wn = [-6.44802700/3.83480709,0.03598578/3.83480709, 1]

df = pd.DataFrame(dict(x=dataSet['petal_length'], y= dataSet['petal_width'],
label=dataSet['species']))
colors = {"versicolor":'red', "virginica":'blue'}
fig, ax = plt.subplots()
grouped = df.groupby('label')
for key, group in grouped:
   group.plot(ax=ax, kind='scatter', x='x', y='y', label=key, color=colors[key])


descisonBoundary(wn)
plt.title('After 50000 iterations: Decision Boundary')
```

```
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.show()
```

The results are then MSE of 0.4982996 for the decision boundary values of [1,1,1]. After 50000 iterations the new boundary values are [-6.44802700, 0.03598578, 3.83480709] with a new MSE of roughly 0.0793693. We can then compare the difference of values in $[w_0\ w_1\ w_2]$ .The before and after plots are.





Exercise 3: Learning a decision boundary through optimization

a & b) We combined both part 'a' and 'b' objectives by creating a combined function that implements gradient descent to optimize the decision boundary for the iris dataset as well as shows the progress in two plots depending on condition chosen: one showing the current decision boundary location overlaid on the data and the second showing the learning curve. The code is presented below.

```
#Q3 part a and b

#display curve is true or false string depending on whether the learning curve
#is being shown or not. If False, then the current decision boundary is shown
#one can choose the step size and the number of iterations to optimize the line
# Choose values 'TRUE' or 'FALSE' for display_curve to show the learning curve or
current plot

def optimize(w,step_size, no_iterations, display_curve):
 mse = [0]*no_iterations
 iteration = [0]*no_iterations
 preW = [0]*3
 mm = 0

 for i in range(no_iterations):
   # break this loop if too small a difference
   if abs(w[0] - preW[0]) < 0.00005 and abs(w[1] - preW[1]) < 0.00005 and abs(w[2] -
preW[2]) < 0.00005:
     print("Weight difference is less than 0.00005")
     mm = i
     break

   #create data for learning curve
   mse[i] = mean_squared_error(w,X_data, y_data)
   iteration[i] = i

   #reset both the previous w values and append current w values with gradsum values
   preW = w[:]
   w = np.subtract(w,np.multiply(gradsum(X_data,y_data,w),step_size))
  #in case more iterations occur but weight difference is too small, get rid of that
data
 if mm != 0:
   del mse[mm:]
   del iteration[mm:]
```

```
  #print new MSE values and the w values
 print(w)
 print('MSE:' , mean_squared_error(w, X_data, y_data))

 #code for the learning curve plot
 def learning_curve():
   plt.clf()
   plt.cla()
   plt.close()
     # Learning curve
   plt.figure(0)
   plt.plot(iteration,mse, '-b')
   plt.xlabel('Iterations')
   plt.ylabel('Mean Squared Error')
   plt.title("Learning curve with step " + str(step_size))
   plt.xlim(-1,no_iterations)
   plt.show()
  #code for the decision boundary line over the iris data
 def current_plot():

   wf = [w[0]/w[2], w[1]/w[2], 1]
   df = pd.DataFrame(dict(x=dataSet['petal_length'], y= dataSet['petal_width'],
label=dataSet['species']))
   colors = {"versicolor":'red', "virginica":'blue'}
   fig, ax = plt.subplots()
   grouped = df.groupby('label')
   for key, group in grouped:
       group.plot(ax=ax, kind='scatter', x='x', y='y', label=key, color=colors[key])

   descisonBoundary(wf)
   plt.title("Decision Boundary")
   plt.xlabel('Petal Length')
   plt.ylabel('Petal Width')
   plt.show()

 if display_curve == 'FALSE':
   current_plot()
 else:
   learning_curve()
```

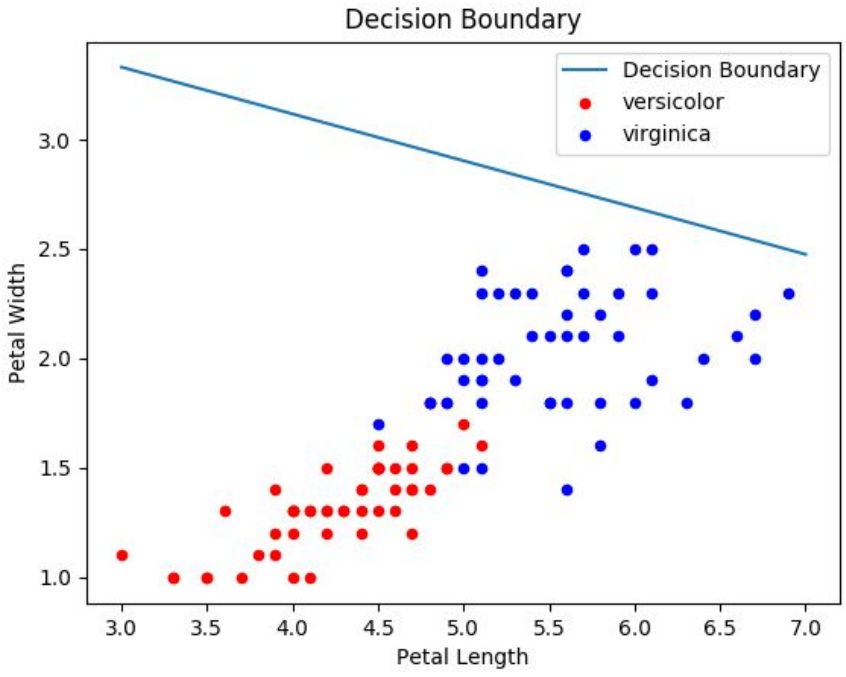
c) We tried random weights and chose a step size after trial and error of 0.005. After that we chose the initial, middle and final decision boundaries by increasing the amount of iterations each time. The example iterations are given below but one can choose any. After each iteration sequence, the MSE gets smaller and the line moves closer to the points. Our example random weight caused the weight difference to reduce to less than 0.00005 but this will not happen all the time. The code and the respective plots are shown below.

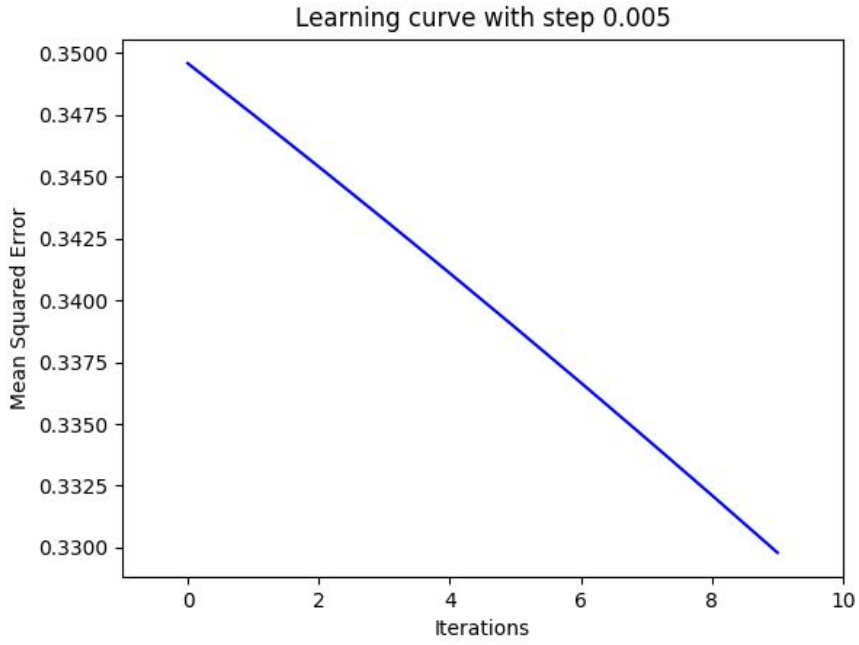
```
wcc = [r.uniform(-10,0),r.uniform(0,1),r.uniform(0,10)]

optimize(wcc,0.005,10, 'TRUE')
optimize(wcc,0.005,10, 'FALSE')

optimize(wcc,0.005,100, 'TRUE')
optimize(wcc,0.005,100, 'FALSE')

optimize(wcc,0.005,2000, 'TRUE')
optimize(wcc,0.005,2000, 'FALSE')
```

Values outputted from random generator are roughly [-8, 0.4, 2]

- *Initial locations of the decision boundary*

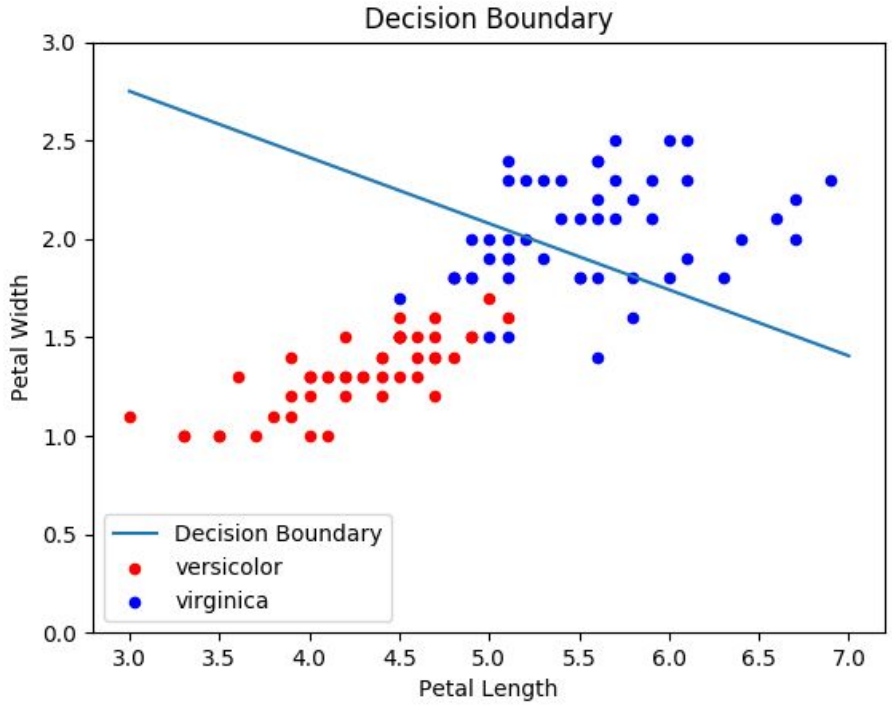
Near initial boundaries are [-7.99457208 0.43070378 2.01142569]
Initial MSE: 0.32744555546839654

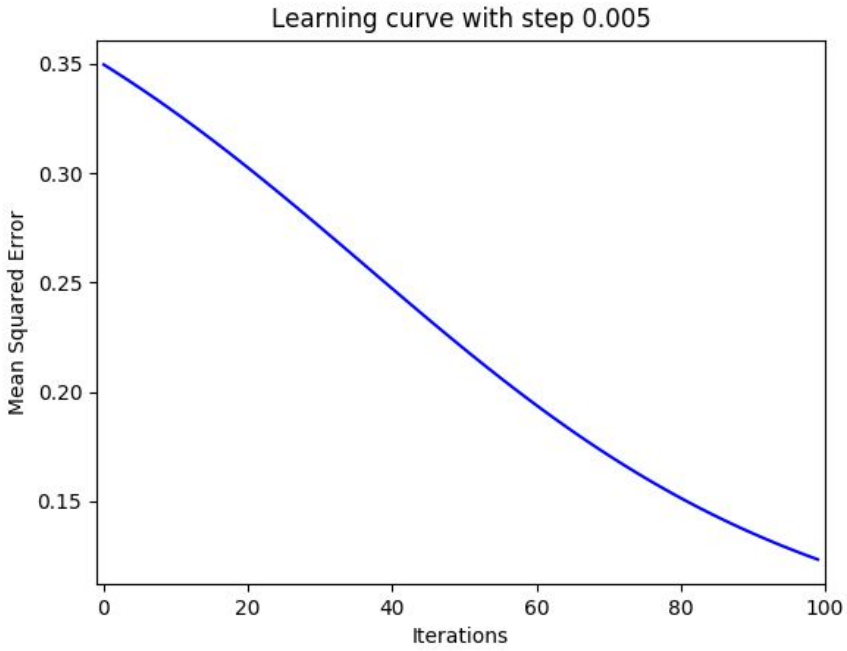




- *Middle locations of the decision boundary*

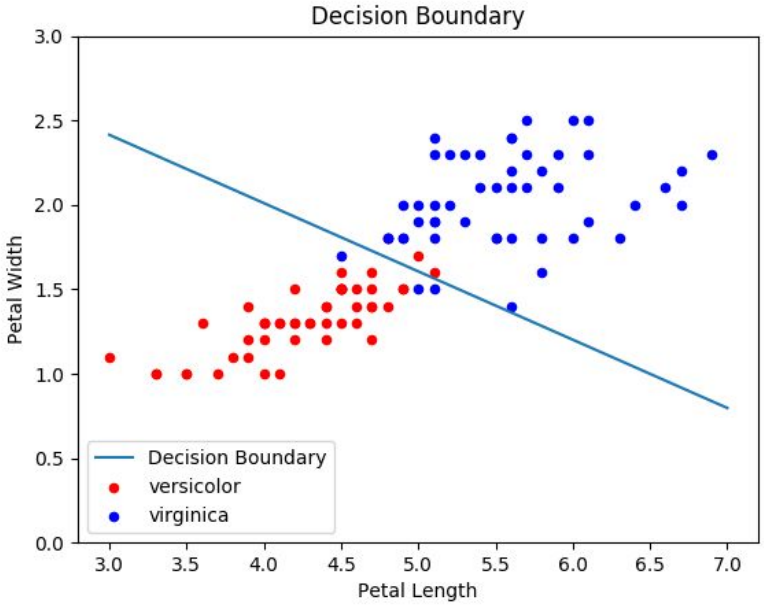
New boundaries are [-7.94443994 0.71030216 2.11395164]
Middle MSE: 0.12219317299804885

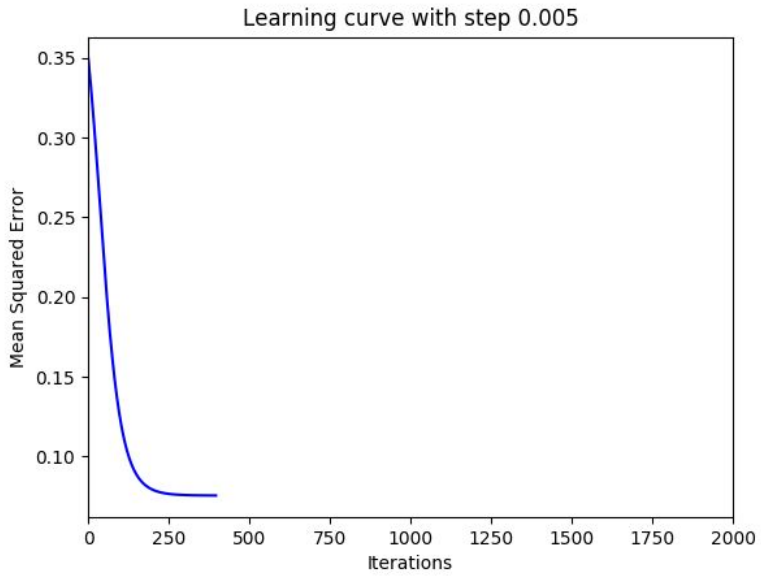




- *Final locations of the decision boundary*

Weight difference is less than 0.00005
New boundaries are [-7.91944151 0.88273692 2.18315529]
Final MSE: 0.0754687497940513





d) I experimented with various step sizes ranging from 0.00005 to 0.5 and decided on 0.005 since it produced good results based on trial and error. The tested values greater than 0.005 caused the gradient to jump around the minimum and not converge properly. On the other hand, values smaller than 0.005 caused the convergence to occur after a longer amount of time. Hence I wanted to find a value that converged after a reasonable time and didn't keep getting stuck on a local minimum, giving me the value of 0.005.

e) We chose the stopping criteria based on a tolerance threshold. The combined code in part 'a' and 'b' has a condition where if the change in the gradient is smaller than 0.00005, then this would stop the algorithm. I chose the value of 0.00005 after trial and error to find an effective value where the gradient magnitude starts becoming too small.

Exercise 4: Extra credit: Using a machine learning toolbox

For exercise 4 of this project, we used the Python machine learning toolbox: scikit-learn to generalize the neural network and classify all the three iris classes using the 4 data dimensions: Sepal Length, Sepal Width, Petal Length and Petal Width.

We first defined the two variables attributes and classes. Attributes contains the values for the 4 data dimensions and classes contains the values for the class that each iris belongs to as shown in the code below:

```
# Exercise 4: Extra Credits
#define the attributes matrix to be the first four columns, and the classes vector
to be the species column
dataSet_full = pd.read_csv("irisdata.csv")
```

```
attributes = dataSet_full.iloc[:,0:4].values
classes = dataSet_full.iloc[:,4:].values
```

From the scikit-learn model selection library, we used the train_test_split method in order to split our data set into two data sets: two thirds of the initial data-set for training and the other third for testing as shown in the code below:

```
#Using the teain_test_split method from the skleran library to split the data set to 25% test and 75% train
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(attributes, classes, test_size = 0.33)
Y_train = np.ravel(Y_train)
```

The last line of code shown above: Y_train = np.ravel(Y_train) just changes the shape of the Y_train from column vector to 1D array. Using the MLPClassifer method from the scikit-learn neural network toolbox, we were able to train a neural network with one hidden layer of size 4 (the number of data dimensions in the iris dataset) with maximum iterations of 50000 as shown in the code below:

```
from sklearn.neural_network import MLPClassifier
neural_classifer = MLPClassifier(hidden_layer_sizes=(8),  max_iter=50000)
neural_classifer.fit(X_train, Y_train)
Y_neural_predections = neural_classifer.predict(X_test)
```

**Disclaimer: because the neural network works by first randomly choosing the values for the weights and the bias initializer, everytime the code is run, it will produce a slightly different result.

We used our trained classifier to predict the labels for the testing data set we defined earlier and then produced a classification report using scikit-learn metrics toolbox as shown in the following code.

```
report_neural = skl.metrics.classification_report(Y_test, Y_neural_predections)
print(report_neural)
```

The following report was printed:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	16
versicolor	1.00	0.83	0.91	18
virginica	0.84	1.00	0.91	16
accuracy			0.94	50
macro avg	0.95	0.94	0.94	50
weighted avg	0.95	0.94	0.94	50

The 0.84 value of precision for the virginica class suggests that the neural network classified a few instances as “virginica” while they are in fact not. The 0.83 value of recall for the versicolor class on the other hand suggests that the neural network did not classify some of the “versicolor” irises as “versicolor”. Using the above classification report as a baseline, we tested different hidden layer sizes to try and improve our neural network. Below is the classification report for the hidden_layer_sizes=(8):

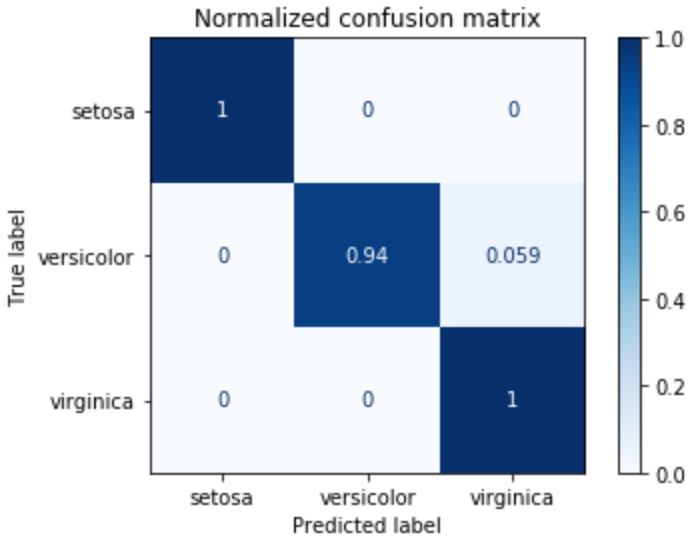
	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	21
versicolor	1.00	0.94	0.97	17
virginica	0.92	1.00	0.96	12
accuracy			0.98	50
macro avg	0.97	0.98	0.98	50
weighted avg	0.98	0.98	0.98	50

Both the precision value for the virginica class and the recall value for the versicolor class improved by increasing the hidden layer size to 8. We tested a hidden layer size of 16 as well to see if it will improve the performance of the neural network but as shown in the report below it didn’t.

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	16
versicolor	0.94	0.89	0.91	18
virginica	0.88	0.94	0.91	16
accuracy			0.94	50
macro avg	0.94	0.94	0.94	50
weighted avg	0.94	0.94	0.94	50

From these results, we selected 8 as the hidden layer size for our neural network. Using the scikit-learn metrics toolbox we also printed the confusion matrix for our neural network as below.

```
from sklearn.metrics import plot_confusion_matrix
disp = plot_confusion_matrix(neural_classifer, X_test, Y_test,
display_labels=["setosa", "versicolor", "virginica"],
                             cmap=plt.cm.Blues, normalize='true')
disp.ax_.set_title("Normalized confusion matrix")
plt.show()
```



We wanted to contrast the results of our neural network to another popular machine learning algorithm: the K Nearest Neighbours algorithm. We used the scikit-learn toolbox to implement a knn classifier as follows.

```
from sklearn.neighbors import KNeighborsClassifier
knn_classifer = KNeighborsClassifier(n_neighbors = 7)
knn_classifer.fit(X_train,Y_train)
Y_knn_predections = knn_classifer.predict(X_test)

report_knn = skl.metrics.classification_report(Y_test, Y_knn_predections)
print (report_knn)

from sklearn.metrics import plot_confusion_matrix
disp = plot_confusion_matrix(knn_classifer, X_test, Y_test,
display_labels=["setosa", "versicolor", "virginica"],
                             cmap=plt.cm.Blues, normalize='true')
disp.ax_.set_title("Normalized confusion matrix")
plt.show()
```

We initially selected k to be 3, and printed the following classification report like we did for the neural network earlier:

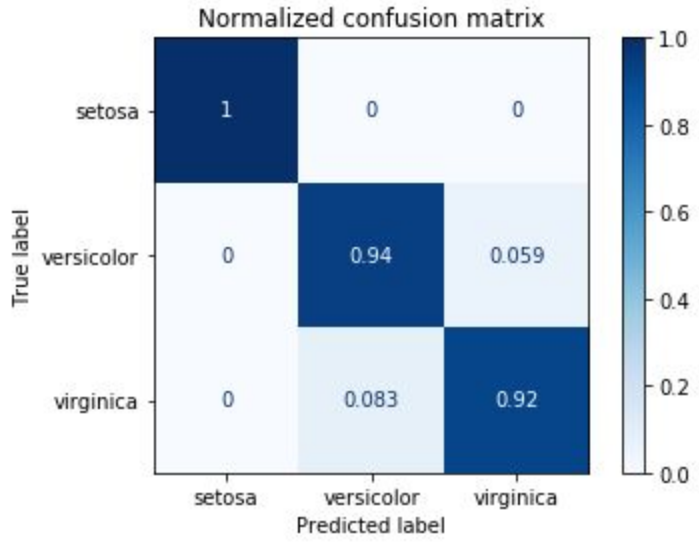
	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	16
versicolor	0.94	0.89	0.91	18
virginica	0.88	0.94	0.91	16
accuracy			0.94	50
macro avg	0.94	0.94	0.94	50
weighted avg	0.94	0.94	0.94	50

We then tried to improve our classifier by manually testing different k values so we tested 5 as an alternative to 3 and got the following classification report:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	21
versicolor	0.94	0.94	0.94	17
virginica	0.92	0.92	0.92	12
accuracy			0.96	50
macro avg	0.95	0.95	0.95	50
weighted avg	0.96	0.96	0.96	50

We settled on 5 as our final k value, because it improved the precision value for virginica and the recall value for versicolor

Below is the plotted confusion matrix for the knn (k=5) classifier obtained the same way as for the neural network above:



Contrasting the confusion matrix and classification report of both the neural network classifier and the knn classifier, we can tell that both algorithms performed very well on this data set but the neural network classifier did slightly better. They both managed to classify the “setosa” iris class with a 100% accuracy but the neural network classifier performed better than the knn classifier with regards to the “virginica” class and they both performed similarly while classifying the “versicolor” class as suggested by the confusion matrices.