ICS 2207
Machine Learning Course Project

Name: Deborah Vella
ID: 366299M

Table of contents

Statement of completion

Item	Completed(Yes/No/Partial)
Data visualisation (Artifact 1)	Yes
k-Means (Artifact 2)	Yes
k-NN (Artifact 3)	Yes
Evaluation (Artifact 1)	Yes
Evaluation (Artifact 2)	Yes
Evaluation (Artifact 3)	Yes
Overall conclusions	Yes

Introduction

In this report I will be discussing the algorithms used to implement the artifacts. Furthermore, for each artifact I will be evaluating all results and plots generated. Some evaluations are done graph by graph while others are done on a set of graphs, whenever they all lead to same conclusions. By doing so, I will be able to assess if the program is working correctly and if the algorithms implemented are fine. All tasks were attempted and completed and if anything does not work as it should, it will also be stated in the report.

Each question has its own .cmd file to run it called:

1. runQuestion1.cmd
2. runQuestion2.cmd
3. runQuestion3.cmd

The Plagiarism Declaration Form is stored as a jpeg file inside the folder of the assignment.

Data Visualisation (Artifact 1)

Statement of completion

Attempted and completed

Overview

For this task, I created two .py files one called question1 and the other called methods. question1.py caters for the user inputs in choosing the number of features and which features s/he wants to see. According to the user's choice, the program executes the corresponding if statements passing the needed data to the functions found inside the methods.py file. The methods.py file mostly contains functions for outputting 1D, 2D and 3D plots.

Code explanation

question1.py

First the needed packages are imported, which are csv, methods and matplotlib. The code continues with the declaration of 12 arrays (4 arrays for each of the 3 classes), each will store a particular feature for their respective class.

```
import csv
import methods
import matplotlib.pyplot as plt
#arrays that will store all data respective to their class
setosa1 = []  #stores sepal length for Iris-Setosa
setosa2 = []  #stores sepal width for Iris-Setosa
setosa3= []   #stores petal length for Iris-Setosa
setosa4 = []  #stores petal width for Iris-Setosa

versicolor1 = []  #stores sepal length for Iris-Versicolor
versicolor2 = []  #stores sepal width for Iris-Versicolor
versicolor3= []   #stores petal length for Iris-Versicolor
versicolor4 = []  #stores petal width for Iris-Versicolor

virginica1 = []  #stores sepal length for Iris-Virginica
virginica2 = []  #stores sepal width for Iris-Virginica
virginica3= []   #stores petal length for Iris-Virginica
virginica4 = []  #stores petal width for Iris-Virginica
```

It then proceeds to read the iris.data text file and populating the arrays mentioned above. Given that each class has 50 rows, I could segment it with three if statements to be able to read and store the data according to each class.

```
 #opening file an populating the arrays above
with open('iris.data.txt', 'r') as csvfile:
       plots = csv.reader(csvfile, delimiter=',')
       i=0
       for row in plots:
           if(i < 50): #collect all features for iris-setosa
                    setosa1.append(float(row[0]))
                   setosa2.append(float(row[1]))
                   setosa3.append(float(row[2]))
                   setosa4.append(float(row[3]))
           elif(i<100): #collect all features for iris-versicolor
                    versicolor1.append(float(row[0]))
                   versicolor2.append(float(row[1]))
                   versicolor3.append(float(row[2]))
                   versicolor4.append(float(row[3]))
           elif(i<150): #collect all features for iris-virginica
                    virginica1.append(float(row[0]))
                   virginica2.append(float(row[1]))
                   virginica3.append(float(row[2]))
                   virginica4.append(float(row[3]))
           i += 1
```

After it collects the data, the user is prompted to enter the number of features s/he wants plotted. After the number is entered, a method called choose_one_feauture() is called which eventually prompts the user to choose the specific features s/he wants plotted. This is followed by if statements, which will execute based on the user's previous input. Each if statement calls the function that plots the scatter plot needed, passing the necessary arrays (each having a feature for a class).

```
choice = input('Choose number of features (e.g. 1,2,3): ')

if choice == '1': #caters for 1 feature
   methods.choose_one_feature() #displays choice of features
   feature = input()
   #according to the feature selected, the method one_scatter is called
passing the chosen feature as parameter
   if feature == '1':
       x = 'Sepal Length cm'
       methods.one_scatter(setosa2, versicolor2, virginica2, plt,x)
   elif feature == '2':
       x = 'Sepal Width cm'
       methods.one_scatter(setosa2,versicolor2,virginica2,plt,x)
   elif feature == '3':
```

```
        x = 'Petal Length cm'
        methods.one_scatter(setosa3,versicolor3,virginica3,plt,x)
    elif feature == '4':
        x = 'Petal Width cm'
        methods.one_scatter(setosa4,versicolor4,virginica4,plt,x)
    else:
        print("Wrong input!")
```

In case of two or three features are wanted, the program gets a predefined number according to the feature. This is done by calling the apply_nos_to_choice() function found in methods.py shown below.

```
def apply_nos_to_choice(feature):
        if feature == '1':
            return 50
        elif feature == '2':
            return 66
        elif feature == '3':
            return 40
        elif feature == '4':
            return 95
        else:
            print("Wrong input!")
            return 0
```

The returned numbers are then added all together and each different combination of the numbers get a unique answer. Therefore, based on the sum of these number, it can be known which features are wanted by the user, irrespective or the order they were entered in. According to the sum, the program will call the method that performs the scatter plot and passes the features denoted by the addition. The code below is in the case of 2 features. When 3 features are wanted, the same algorithm is implemented by using 3 numbers instead of 2.

```
elif choice == '2': #caters for 2 features
    methods.choose_one_feature()
    feature = input()
    no1 = methods.apply_nos_to_choice(feature)
    methods.choose_one_feature()
    feature2 = input()
    no2 = methods.apply_nos_to_choice(feature2)

    addition = no2 + no1

    if addition == 116: #It means that 1 and 2 where chosen irrispective of
order
        x='Sepal Length cm'
```

```
        y='Sepal Width cm'
        methods.two_scatter(setosa1,setosa2,versicolor1,versicolor2,virginica1,vi
        rginica2,x,y,plt)
    elif addition == 90: #It means that 1 and 3 where chosen irrispective of
order
        x = 'Sepal Length cm'
        y = 'Petal Length cm'
        methods.two_scatter(setosa1, setosa3, versicolor1, versicolor3,
virginica1, virginica3,x,y, plt)
    elif addition == 145: #It means that 1 and 4 where chosen irrispective of
order
        x = 'Sepal Length cm'
        y = 'Petal Width cm'
        methods.two_scatter(setosa1, setosa4, versicolor1, versicolor4,
virginica1, virginica4,x,y ,plt)
    elif addition == 106: #It means that 2 and 3 where chosen irrispective of
order
        x = 'Sepal Width cm'
        y = 'Petal Length cm'
        methods.two_scatter(setosa2, setosa3, versicolor2, versicolor3,
virginica2, virginica3, x, y, plt)
    elif addition == 161: #It means that 2 and 4 where chosen irrispective of
order
        x = 'Sepal Width cm'
        y = 'Petal Width cm'
        methods.two_scatter(setosa2, setosa4, versicolor2, versicolor4,
virginica2, virginica4, x, y, plt)
    elif addition == 135: #It means that 3 and 4 where chosen irrispective of
order
        x = 'Petal Length cm'
        y = 'Petal Width cm'
        methods.two_scatter(setosa3, setosa4, versicolor3, versicolor4,
virginica3, virginica4, x, y, plt)
    else:
        print("Wrong input!")
```

methods.py

This contains only functions which are executed when called from question1.py. It first imports the packages needed which are numpy and Axes 3D.

```
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
Axes3D = Axes3D
```

First method is choose_one_feature() which prompts the user to enter his desired feature.

```
def choose_one_feature():
   print("Press 1 for Sepal Length")
   print("Press 2 for Sepal Width")
   print("Press 3 for Petal length")
   print("Press 4 for Petal Width")
```

Second function is apply_nos_to_choice() which was already explained and shown in the question1.py section.

Third function is one_scatter() which accepts one feature from Iris-setosa, one from Iris-versicolor and one from Iris-virginica. These are then plotted on the x-axis having the y-axis values all set to zero.

```
def one_scatter(setosa,versicolor,virginica,plt,x):
   y = np.zeros(np.shape(setosa))  # Make all y values set to zero
   plt.xlim(0, 10)
   plt.ylim(0, 0.2)
   plt.plot(setosa, y, 'ro', marker='o', markersize=4, c='r',
label='Iris-setosa')
   plt.plot(versicolor, y, 'ro', marker='o', markersize=4, c='b',
label='Iris-versicolor')
   plt.plot(virginica, y, 'ro', marker='o', markersize=4, c='g',
label='Iris-virginica')
   plt.axis('auto')
   plt.xlabel(x)
   plt.ylabel('y')
   plt.legend(loc=2)
   plt.show()
```

Fourth function deals with the plotting of 2D scatter plots. This accepts two features from each class and plots them against each other.

```
Def two_scatter(setosa1,setosa2,versicolor1,versicolor2,virginica1,virginica2,
x,y,plt):
   plt.scatter(setosa1, setosa2, c='red',s=5, label='Iris-setosa')
   plt.scatter(versicolor1, versicolor2, s=5,c='blue', label='Iris-Versicolor')
   plt.scatter(virginica1, virginica2,s=5, c='green', label='Iris-Virginica')
```

```
    plt.xlabel(x)
    plt.ylabel(y)
    plt.legend(loc=1)
    plt.show()
```

Last but not least, there is the function that plots the features in 3D space. It is the same as the 2D function but this time it accepts one more feature from each class.

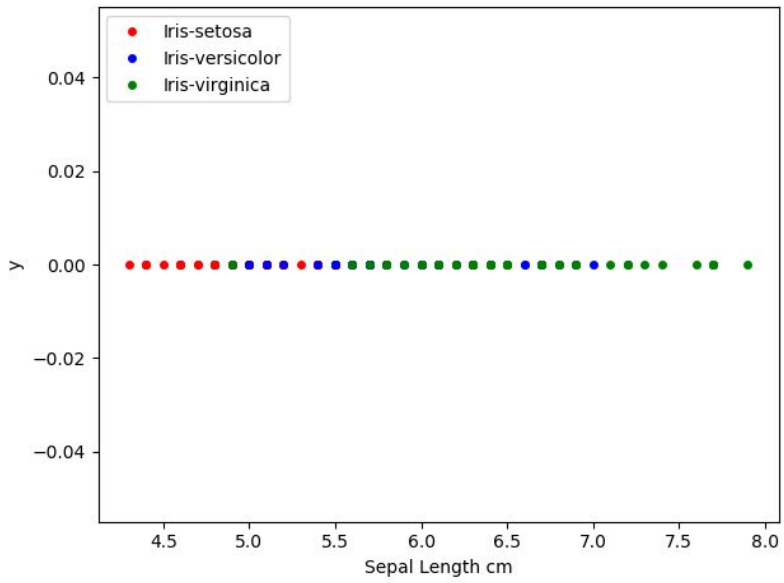
```
def three_scatter(setosa1,setosa2,setosa3,versicolor1,versicolor2,versicolor3,
virginica1,virginica2,virginica3,x,y,z,plt):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(setosa1, setosa2, setosa3, c='r', marker='o',label='Iris-setosa')
    ax.scatter(versicolor1,versicolor2,versicolor3, c='b', marker='o',
label='Iris-Versicolor')
    ax.scatter(virginica1,virginica2,virginica3,c='g', marker='o',
label='Iris-Virginica')
    ax.set_xlabel(x)
    ax.set_ylabel(y)
    ax.set_zlabel(z)
    plt.legend(loc=1)
    plt.show()
```

Evaluation

1-Dimensional

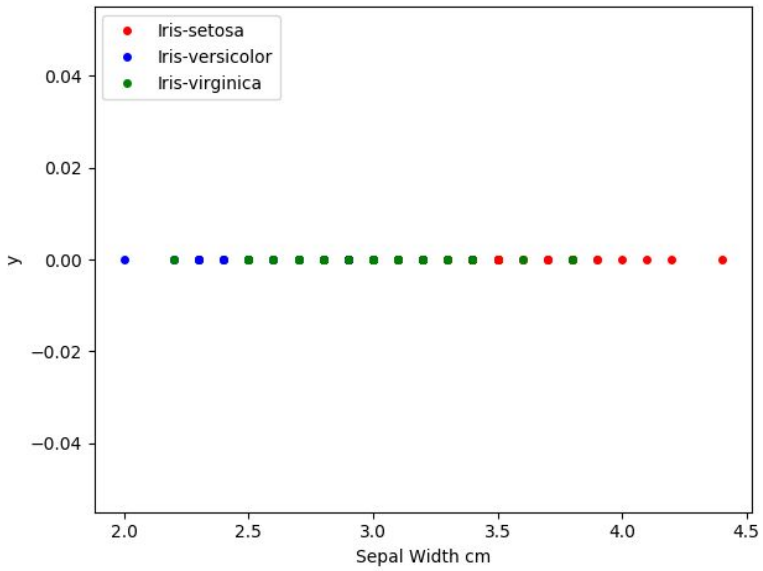
The below are all graphs of one feature. It should be kept in mind that all the graphs below may have some points being overlapped by points from a different class. Therefore, some information might be hidden by other information.

Sepal Length Feature



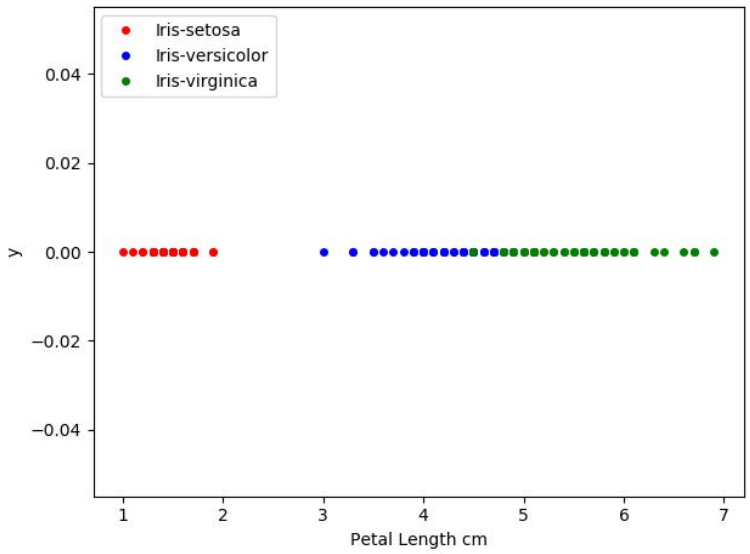
From this graph one can notice that Iris-virginica has a wider range of sepal length values, having most of them lie between 5.5cm and 8cm. Iris-versicolor and Iris-setosa have smaller ranges and smaller lengths. It can be concluded that Iris-virginica on the whole, has a longer sepal while the majority of Iris-setosa has the shortest sepal.

Sepal Width Feature



When comparing this graph to the Sepal Length graph, one may realize that here the points are more spaced out in a smaller range of x values. This may imply that there are more values which are the same or close to each other than there were in the first graph. Furthermore, this time, Iris-setosa is the one which has the highest values, followed by Iris-virginica, making Iris-versicolor the smallest from the 3.

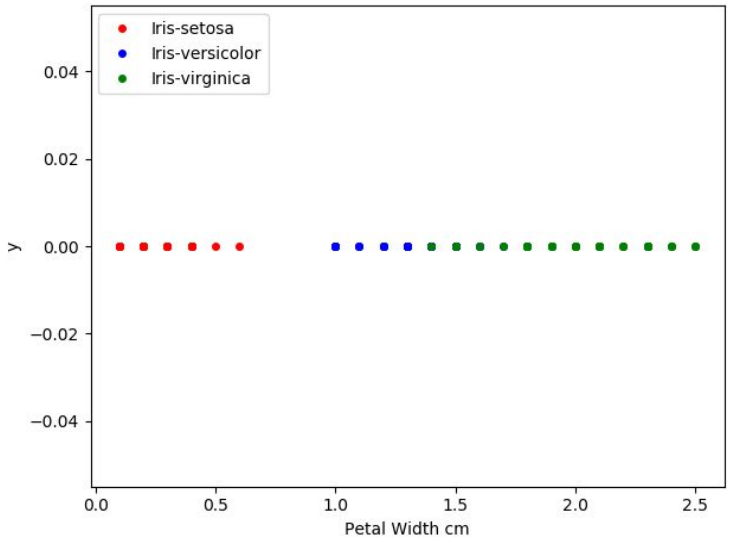
Petal Length Feature



It can be seen that the Iris-setosa points are plotted on their own between 1cm and 2cm, leaving another cm from the rest of the points. This shows that Iris-setosa has the smallest petal length and none of the other flowers overlap it. Iris-virginica has the longest petals, while Iris-versicolor has the second most long petals. The range of Iris-setosa is that of 1cm while the

others are plotted in a range of 2cm or a bit more. This means that Iris-setosa has less variety in its petal length while the other flower classes tend to have more difference in their length from one another. Another thing that may be noticed is that there is not a lot of space between one point and another showing that, the values are very close to each other.

Petal Width Feature



Like the sepal width, the points are spaced out rather than clammed together. On the other hand, Iris-setosa has the smallest dimensions unlike those in the sepal width plot. Once again, Iris-virginica has the highest dimensions, followed by Iris-versicolor.

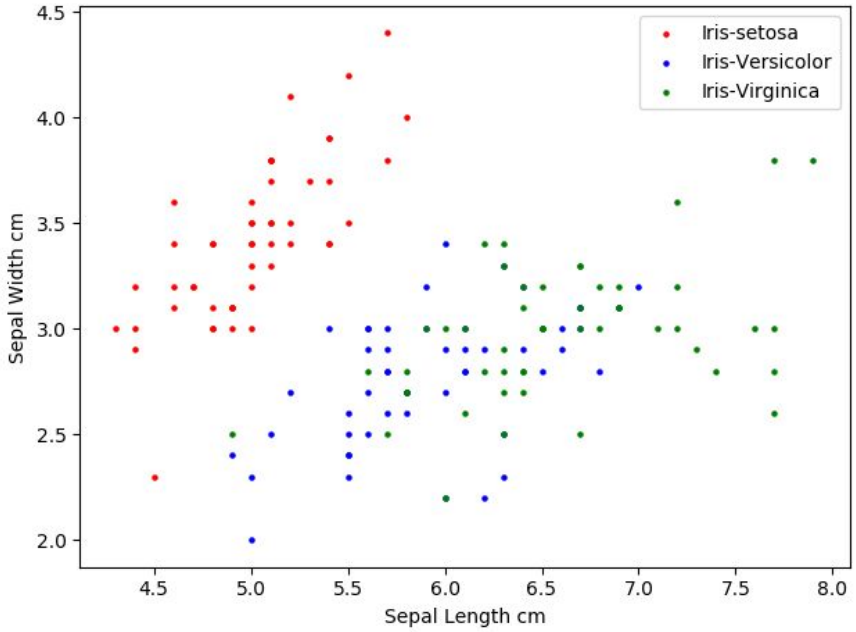
Overall Evaluation

After evaluating all 1D plots, it can be concluded that Iris-virginica is generally the largest flower of the 3. Iris-versicolor's dimensions always lie in between Iris-setosa's and Iris-virginica or are overlapped by one of them or both. This makes it the second largest flower, as a result, Iris-setosa is usually the smallest flower from them all. Iris-setosa's only dimension that is greater than the others' is that of sepal width

2 Dimensional

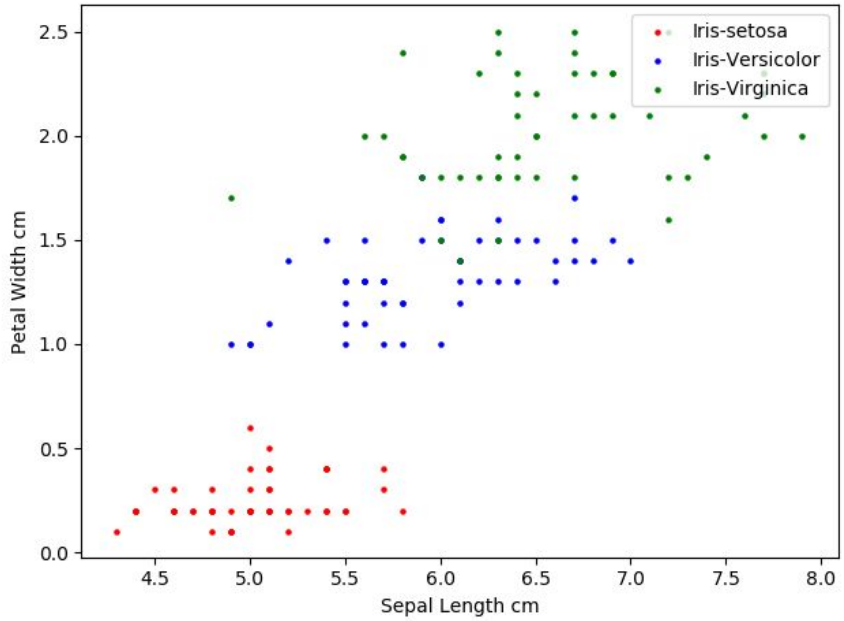
In these graphs there are less overlappings of points helping you visualize more the actual ranges of the values of each feature.

Sepal Width against Sepal Length



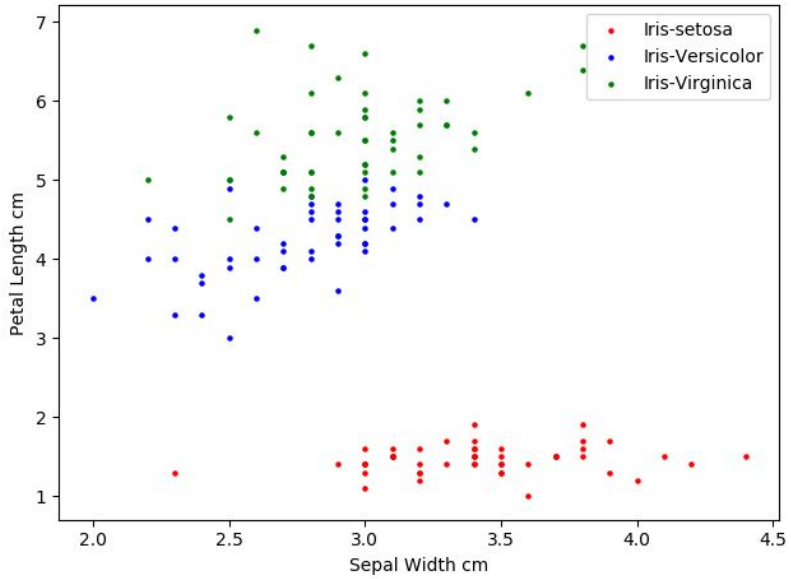
When analyzing the points that represent Iris-setosa, it can be noted that there is a relationship between the sepal length and the sepal width dimensions. This is because the points follow a pattern i.e. the longer the length, the wider the width. A regression line can be drawn on the class of Iris-setosa. In contrast, none of Iris-Versicolor's and Iris-Virginica's points follow any pattern. The points are scattered all around and no regression line can be drawn, which means that there is no relationship between the sepal length and the sepal width. But when comparing Iris-versicolor to Iris-virginica one may realize that Iris-versicolor's are less scattered.

Petal Width against Sepal Length



This plot shows that for all classes, no matter how long the sepal length is, the petal width is not affected. This is because the range of the petal width for all classes is slightly above 0.5cm which make the points follow the pattern of a horizontal line.

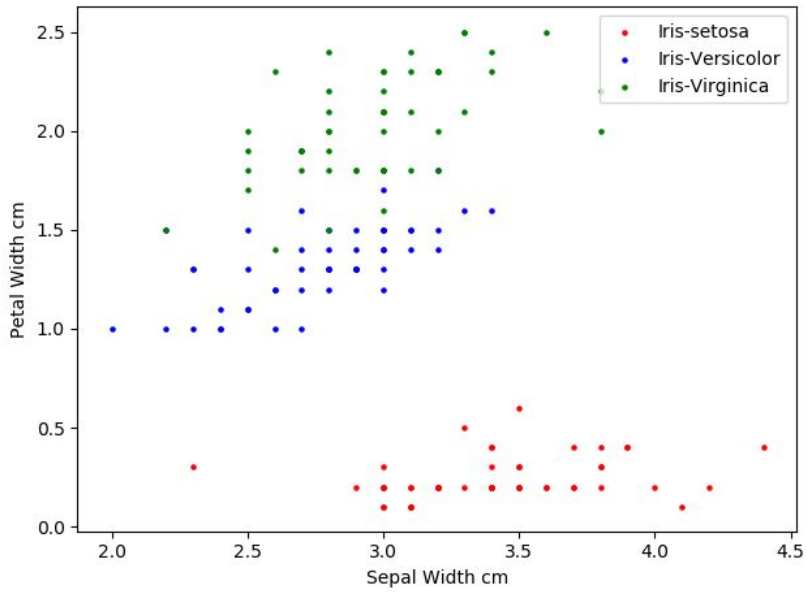
Petal Length against Sepal width



When looking at Iris-Versicolor plot, it can be seen that the smaller the sepal width is, the more scattered the points are, while the higher the sepal width is, the more close the points become. This shows that the dimensions of the petal length becomes more consistent when the sepal

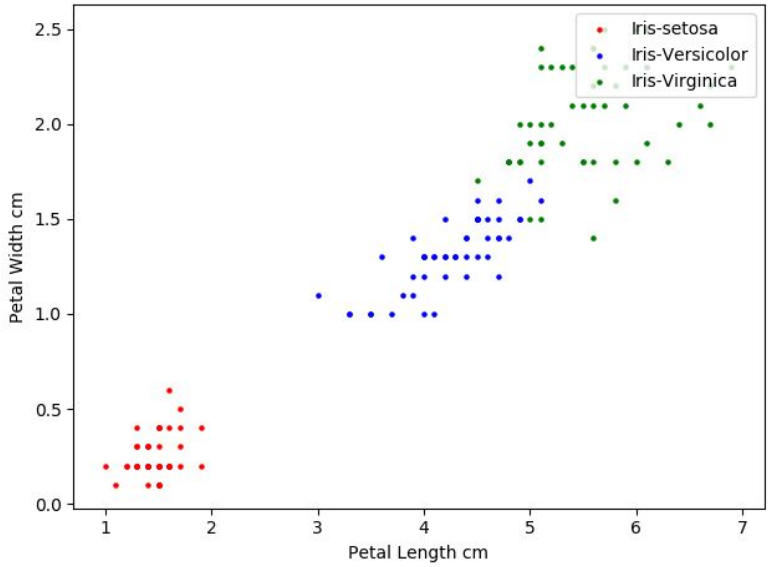
width is around 2.75cm and higher. The Iris-virginica class, is more scattered and once again no regression line can be drawn on it, as it has no type of relationship between the variables. Last but not least, Iris-setosa follows a horizontal line pattern having most of the point vary between 1cm and 2 cm on the Petal Length axis. Therefore, when increasing the sepal width, the Petal length is not affected.

Petal Width against Sepal Width



This time, Iris-virginica's points are following a particular pattern which results in a relationship between the two variables that is the wider the sepal width the wider the petal width. Iris-versicolor's variables also seems to have the same relationship Iris-virginica's have. Once again, Iris-setosa's class looks like the variable on the x-axis is not affecting the value on the y-axis. The points are following a horizontal pattern with a range of 0.5cm on the y-axis.

Petal Width against Petal Length



In this plot all of the classes follow the same pattern and a regression line can be drawn for all of them. The relationship clearly is that, when increasing the petal length, the petal width gets wider as well.

Overall Evaluation

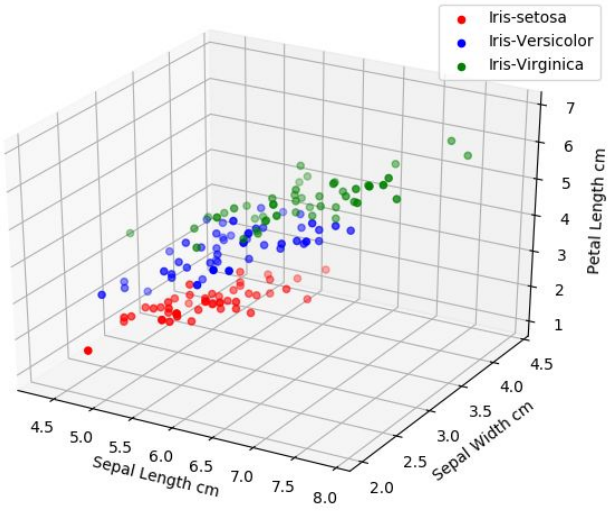
Most of Iris-setosa's plots follow a horizontal pattern therefore, having the x-axis's variables not affect the y-axis's variables.

Most of Iris-virginica's plots are scattered around resulting in no relationship between the variables.

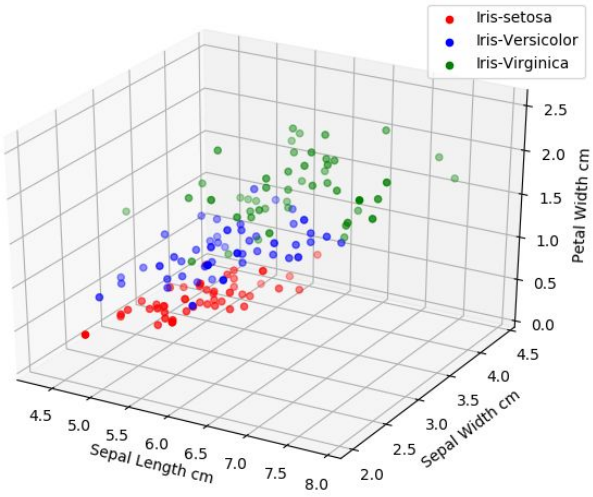
Some of Iris-versicolor's plots show that there is some kind of relationship while the others are more scattered and is not that clear if there is a relationship betweent the variables.

3 Dimensional

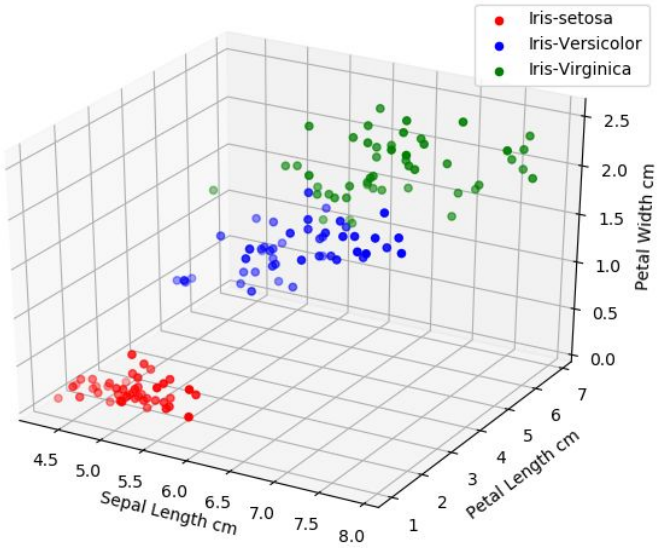
Sepal length, Sepal width and Petal length plotted against each other



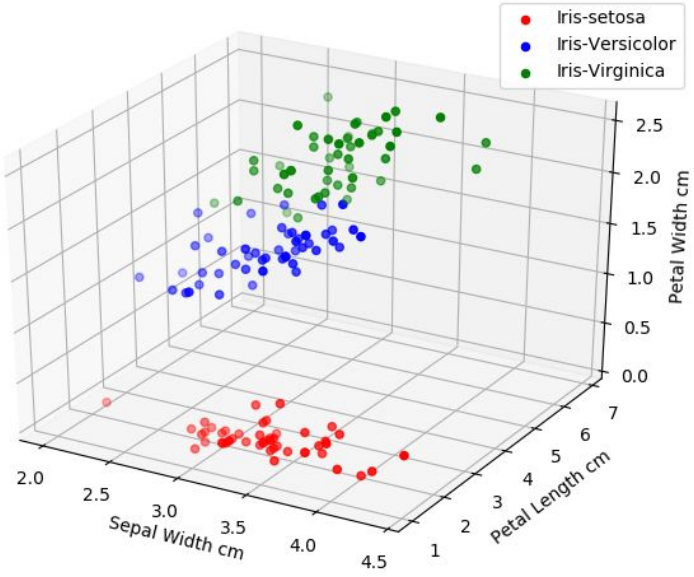
Sepal Length, Sepal width, Petal Width all plotted against each other



Sepal Length, Petal Width and Petal Length all plotted against each other



Sepal Width, Petal Length and Petal Width all plotted against each other



Overall Evaluation

With these 3D scatter plots, the points look more clear especially those which overlap another class's points. These graphs continue showing that Iris-setosa is relatively the smallest flower, while Iris-virginica has the largest dimensions.

K-Means (Artifact 2)

Statement of completion

Attempted and completed. Sometimes a centroid is randomly positioned in a way that it can't find points which are closest to it than to other centroids, therefore the program does not output any results. This happens on rare occasions though, so it mostly runs fine.

Overview of algorithm

I made use of this video to understand more the algorithm of the k-means and eventually implement it in my program
https://www.youtube.com/watch?v=_aWzGGNrcic

Algorithm implemented:

1. K is set to 3 because we are clustering 3 classes
2. Pick random numbers for the x and y (and z if 3D) values for each of the three centroids. Zip the x and y values into one numpy array.
3. Set old centroids equal to zero to start with.
4. Calculate distance between current centroids and old centroids
5. While the distances do not converge
    5.1. Calculate the distance between each point and the centroids.
    5.2. Get nearest centroid and add the point to a cluster
    5.3. Update old centroids to store the current ones
    5.4. Calculate new centroids by computing the mean of their cluster.
    5.5. Calculate distance between current centroids and old centroids
6. Plot features colored according to the cluster they're in.

Code Explanation

question2.py

Main function

Read text file and store each feature type in a different array.

```
def main():
  #each column will contain all date of one particular feature
  column1 = [] #contains Sepal Length
  column2 = [] #contains Sepal Width
  column3 = [] # contains Petal Length
  column4 = [] # Contains Petal Width

  #read text file and populate arrays above
  with open('iris.data.txt', 'r') as csvfile:
    plots = csv.reader(csvfile, delimiter=',')
    for row in plots:
      column1.append(float(row[0]))
      column2.append(float(row[1]))
      column3.append(float(row[2]))
      column4.append(float(row[3]))
```

Prompt user for input just like in question 1. According to the input, execute the correct if statement. In each if statement ask user to choose the feature and the call k-means() function accordingly. The below is for the case of choosing 2 features only, because the rest follow the same pattern.

```
elif choice == '2':  # caters for 2 features

   methods.choose_one_feature()
   feature = input()
   no1 = methods.apply_nos_to_choice(feature)
   methods.choose_one_feature()
   feature2 = input()
   no2 = methods.apply_nos_to_choice(feature2)

   addition = no2 + no1

   if addition == 116:  # It means that 1 and 2 where chosen irrispective of
order
       data = np.array(list(zip(column1, column2))) #join the 2 arrays into one
2D array
       k_means(data, 2, 0)  #pass 2Darray of data, number of features, pass 0
cause 4th feature not chosen
```

```
   elif addition == 90:  # It means that 1 and 3 where chosen irrispective of
order
       data = np.array(list(zip(column1, column3)))#join the 2 arrays into one
2D array
       k_means(data, 2, 0) #pass 2Darray of data, number of features, pass 0
cause 4th feature not chosen
   elif addition == 145:  # It means that 1 and 4 where chosen irrispective of
order
       data = np.array(list(zip(column1, column4)))#join the 2 arrays into one
2D array
       k_means(data, 2, 4)  #pass 2Darray of data, number of features, pass 4
cause 4th feature is chosen
   elif addition == 106:  # It means that 2 and 3 where chosen irrispective of
order
       data = np.array(list(zip(column2, column3)))#join the 2 arrays into one
2D array
       k_means(data, 2, 0)  #pass 2Darray of data, number of features, pass 0
to later choose the right random centroids
   elif addition == 161:  # It means that 2 and 4 where chosen irrispective of
order
       data = np.array(list(zip(column2, column4)))#join the 2 arrays into one
2D array
       k_means(data, 2, 4)  #pass 2Darray of data, number of features, pass 4
cause 4th feature is chosen
   elif addition == 135:  # It means that 3 and 4 where chosen irrispective of
order
       data = np.array(list(zip(column3, column4)))#join the 2 arrays into one
2D array
       k_means(data, 2, 4)  #pass 2Darray of data, number of features, pass 4
cause 4th feature is chosen
```

dist() function

This function calculates the euclidean distance by using linalg.norm from the numpy library.

```
def dist(p, q, ax=1):
   return np.linalg.norm(p - q, axis=ax) #calculates euclidean distance
```

K_means() function

The first parameter is the dataset on which the clustering is to be performed. The second parameter denotes the number of features being passed. The last parameter contains either the number 4 or 0. This will later be used to find random floats in different ranges because I realised that when the 4th feature is chosen, one or more centroids could never be positioned in a way that they find a cluster of points.

```
def k_means(data, num_features, number):
```

This function starts by setting k equal to 3 and proceeds to find the centroids randomly.

```
k = 3

centroids_x = []
centroids_y = []
x = 0
if number == 4: #For when the fourth feature is chosen.  This requires
different ranges from which random numbers can be picked
   #else, one of the centroid will get a position from where it can never get
a cluster of its own
   while x < 3:
       if num_features == 3: # pick random numbers for centroids for 3D space
           centroids_x = np.random.uniform(low=4.1, high=6.1, size=(3,))#x
values
           centroids_y = np.random.uniform(low=0.0, high=2.0, size=(3,))#y
values
           centroids_z = np.random.uniform(low=0.0, high=2.5, size=(3,))#z
values
       elif num_features == 1: # pick random numbers for centroids for 1D
space
           centroids_x = np.random.uniform(low=0.1, high=6.1, size=(3,)) #x
values
           centroids_y = np.zeros(len(centroids_x))#y values all zeros
       else: ## pick random numbers for centroids for 2D space
           centroids_x = np.random.uniform(low=4.1, high=6.1, size=(3,))#x
values
           centroids_y = np.random.uniform(low=0.0, high=2.0, size=(3,))#y
values
       x += 1
else:#if feature 4 is never chosen execute the below with different ranges for
random centroids
   while x < 3:
       if num_features == 3:
           centroids_x = np.random.uniform(low=4.1, high=6.1, size=(3,))
           centroids_y = np.random.uniform(low=0.0, high=1.0, size=(3,))
           centroids_z = np.random.uniform(low=2.1, high=4.5, size=(3,))
       elif num_features == 1:
```

```
            centroids_x = np.random.uniform(low=0.1, high=6.1, size=(3,))
            centroids_y = np.zeros(len(centroids_x))
        else:
            centroids_x = np.random.uniform(low=4.1, high=6.1, size=(3,))
            centroids_y = np.random.uniform(low=2.1, high=4.5, size=(3,))

        x += 1
```

When the centroids' x,y (and z if needed) coordinates are randomly generated, they are joined into one array. The arrays of old centroids and clusters are then declared and initialized to an array of zeros. The distance between new and old centers is calculated.

```
#join the arrays into one array of centroids x,y,z positions
if num_features == 3:
   centroids_new = np.array(list(zip(centroids_x, centroids_y, centroids_z)),
dtype=np.float32)
else: #join the arrays into one array of centroids x,y positions
   centroids_new = np.array(list(zip(centroids_x, centroids_y)),
dtype=np.float32)


# To store the value of centroids when it updates
centroids_old = np.zeros(centroids_new.shape) #set to zeros for the first time
# Cluster labels(0, 1, 2)
clusters = np.zeros(len(data))
# calculate distances between old centres and new centers.
converge = dist(centroids_new, centroids_old, None)
```

If the distance did not converge start the clustering algorithm. Start by assigning all data to a cluster. And copy the current centers into the old centers

```
while converge != 0:
   # Assigning each value to its closest cluster
   for i in range(len(data)):
       distances = dist(data[i], centroids_new) #calculate distance between all
data and each centroid
       cluster = np.argmin(distances) #returns 0,1 or 2 according to which is
the smallest distance
       clusters[i] = cluster #stores values of 0,1 and 2 denoting each cluster
# update the old centroids with the current centroids
centroids_old = deepcopy(centroids_new)
```

Then loop through all clusters, and at each iteration, get the list of points found in the group. If the list of points in the cluster is not empty, calculate their mean, but if empty, generate again a random centroid. Then recalculate the distance and check if it converges in the while loop condition. If it does not, run the algorithm again using the new centroids.

```
#for each cluster
for i in range(k):
   # get the points of the cluster i
   ith_cluster_points = [data[j] for j in range(len(clusters)) if clusters[j]
== i]

   if ith_cluster_points:  # if list is not empty
       centroids_new[i] = np.mean(ith_cluster_points, axis=0) #calculate mean
of that cluster for new centroids
   else:#if empty
       #find another random position for the centroid to eventually have
points to collect in its cluster
       if (number == 4):
           centroids_new[i] = np.random.uniform(low=0.0, high=2.0)
       else:
           centroids_new[i] = np.random.uniform(low=3.1, high=6.1)
#calculate the distance between new centroids and old ones
converge = dist(centroids_new, centroids_old, None)
```

When the program exits the while loop, it should be ready to plot the data and the centroids. An if case is provided for all 3 different dimensional spaces.

```
#to be used when plotting the clusters
colors = ['r', 'g', 'b']

if num_features == 1: #plot for 1D
   plt.figure()
   for i in range(k): #plot a cluster at a time
       points = np.array([data[j] for j in range(len(clusters)) if clusters[j]
== i])
       y = np.zeros(np.shape(points))  # Make all y values the same
       plt.xlim(0, 10)
       plt.ylim(0, 0.2)
       plt.plot(points, y, 'ro', marker='o', markersize=4,c=colors[i])#plot
point in i'th cluster
       plt.axis('tight')
   y = [0, 0, 0]
   plt.plot(centroids_new[:, 0], y, 'ro', marker='*', c='#050505')#plot
centroids
elif num_features == 2: #plot for 2D
   for i in range(k):#plot a cluster at a time
       points = np.array([data[j] for j in range(len(clusters)) if clusters[j]
== i])
```

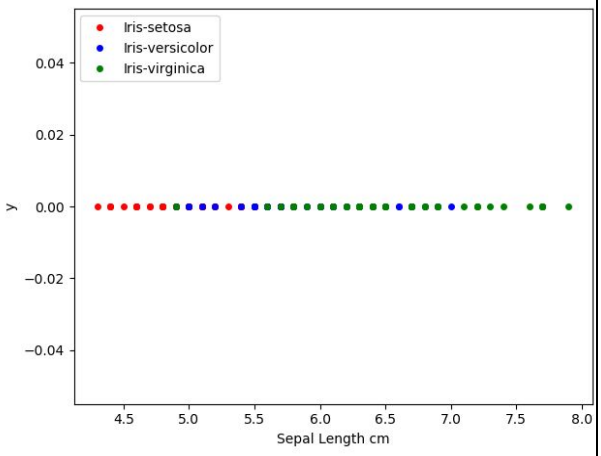
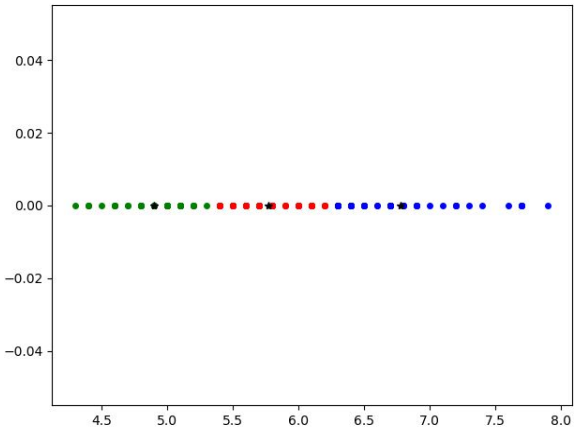
```
        plt.scatter(points[:, 0], points[:, 1], s=20, c=colors[i])#plot point in
i'th cluster
        plt.scatter(centroids_new[:, 0], centroids_new[:, 1], marker='*', s=100,
c='#050505')
elif num_features == 3:
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    for i in range(k):#plot a cluster at a time
        points = np.array([data[j] for j in range(len(clusters)) if clusters[j]
== i])
        ax.scatter(points[:, 0], points[:, 1], points[:, 2], c=colors[i])#plot
point in i'th cluster
        ax.scatter(centroids_new[:, 0], centroids_new[:, 1], centroids_new[:,
2], marker='*', s=180, c='#050505') #plot centroids
plt.show()
```

Evaluation

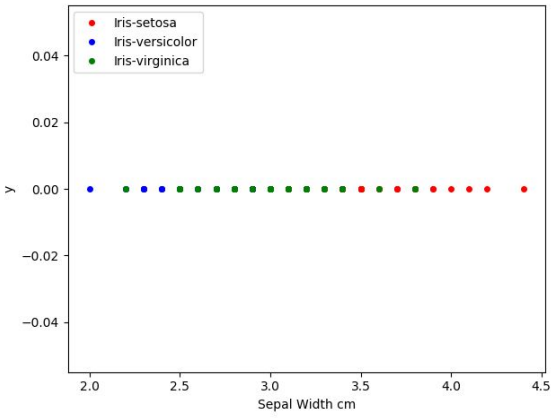
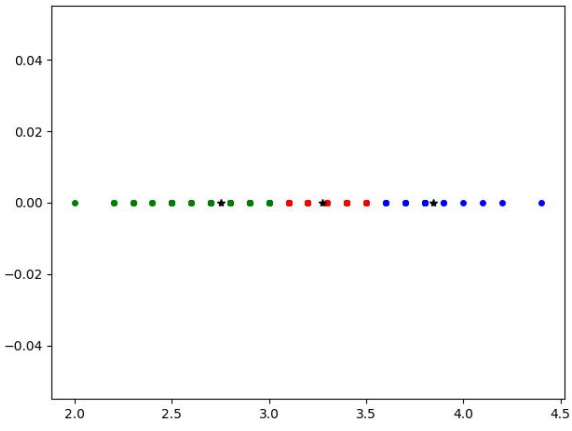
In this section I will be comparing the clustering done by the k-means algorithm, to the plots generated for question 1. This way I can see if the clustering is working on the whole or not. Each centroid is represented by a black star. Each cluster is represented in a different color. These plots will be evaluated in one paragraph as same conclusion can be drawn for all.

1 Dimensional

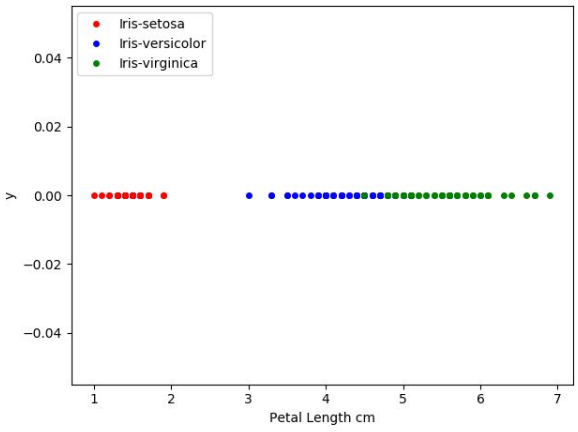
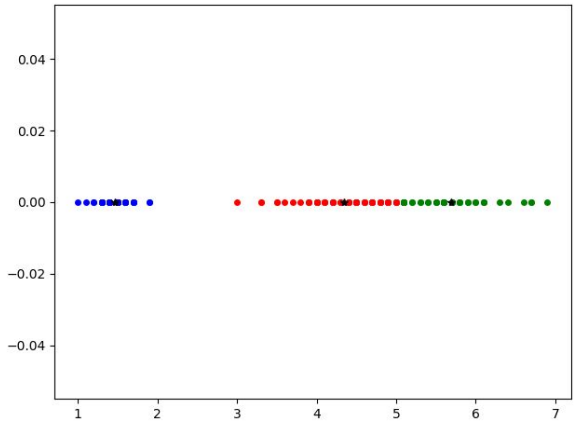
Sepal Length Feature

Question 1 result	K-means result
	

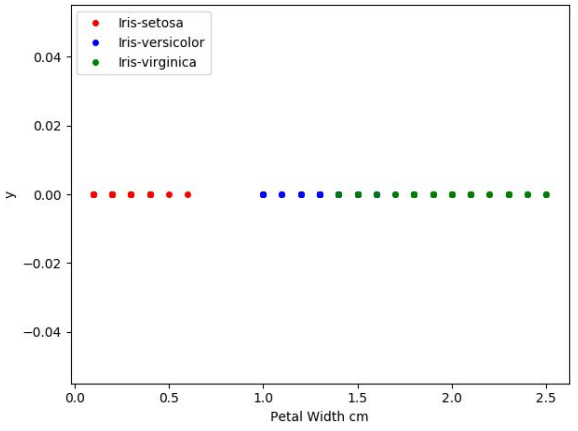
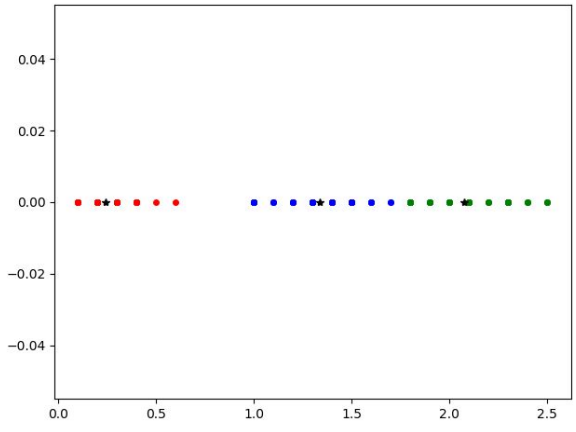
Sepal Width Feature

Question 1 result	K-means result
	

Petal Length Feature

Question 1 result	K-means result
	

Petal Width Feature

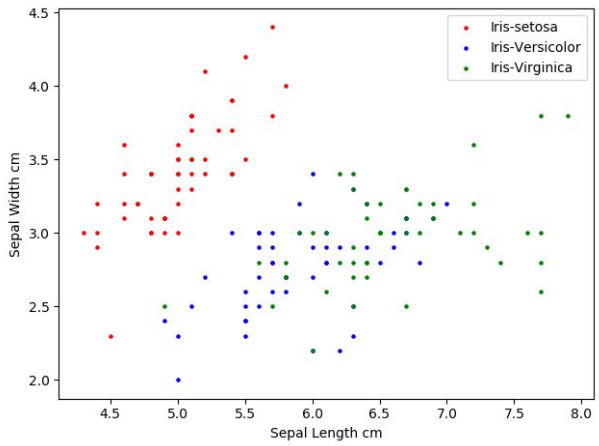
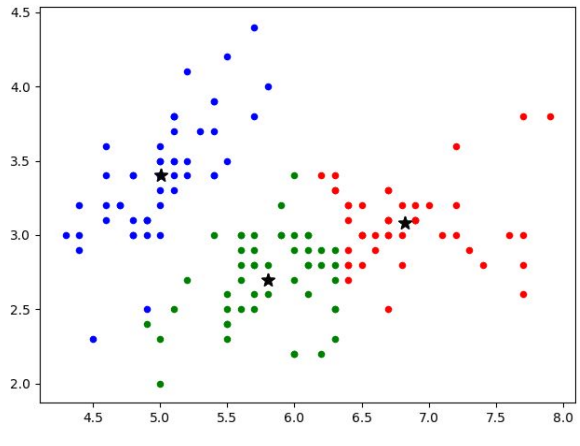
Question 1 result	K-means result
	

Overall Evaluation

When comparing the plots from question 1 to the k-means ones, I noticed that when a cluster overlaps another one, k-means cannot measure under which cluster it should go. Also, the results for question 1 show points of same class being scattered on a wider range. K-means tries to find sub-groups which are similar to each other, therefore, each cluster contains values which are close to each other. Hence, each cluster is spread out on a smaller range. In the evaluation of question 1 it was discussed that some points of a class may be hidden by the overlapping of points from other class. K-means does not allow overlapping of groups and therefore can give you an idea of how spread out the classes may actually be.

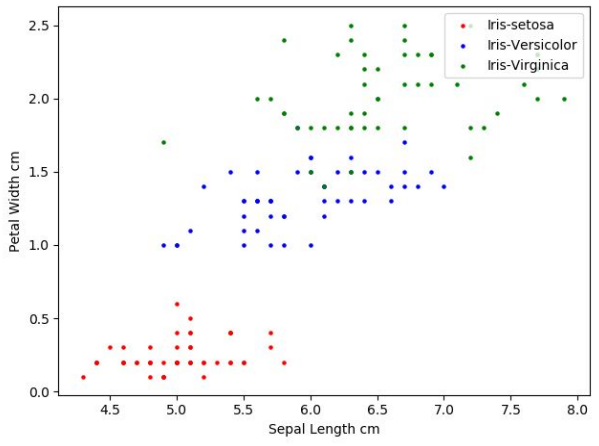
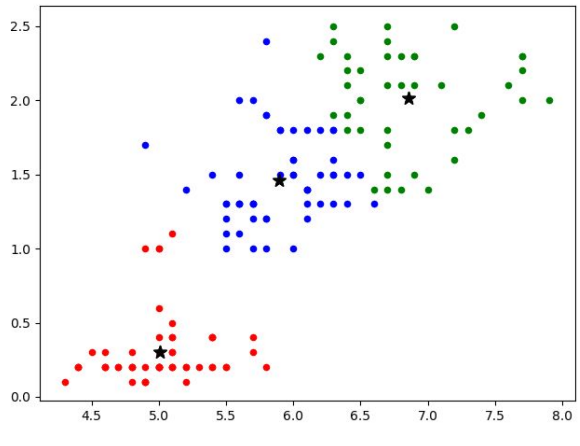
2 Dimensional

Sepal Width against Sepal Length

Question 1 result	K-means result
	

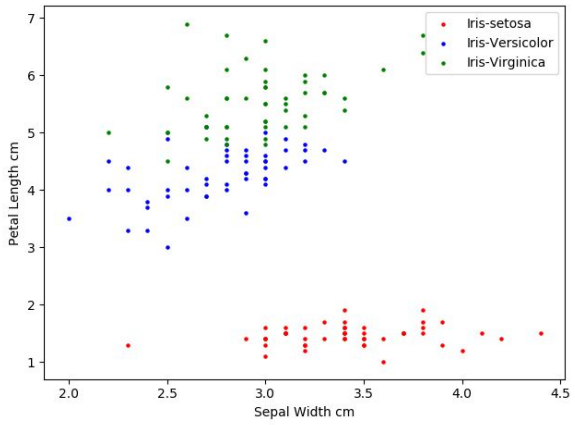
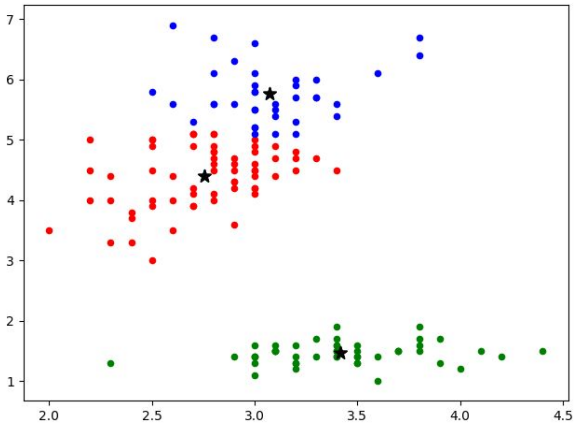
The blue cluster is the same as the Iris-setosa class. The remaining 2 classes overlap each other therefore, the clusters could not be exactly the same as the classes in question 1.

Petal Width and Sepal Length

Question 1 result	K-means result
	

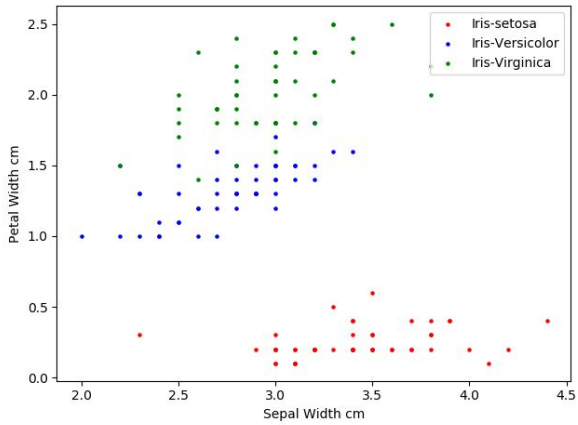
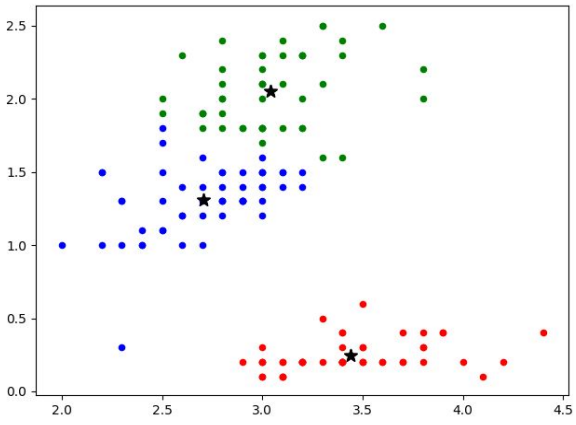
Here the classes in question 1 look very similar to the clusters generated by k-means. If the centroids were slightly lower than they are, the clusters would have been more accurate.

Petal Length against Sepal width

Question 1 result	K-means result
	

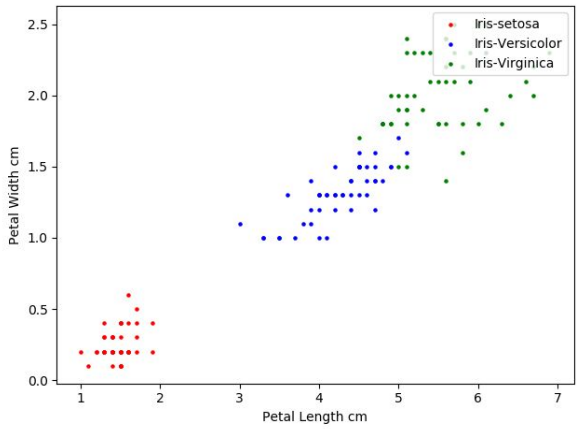
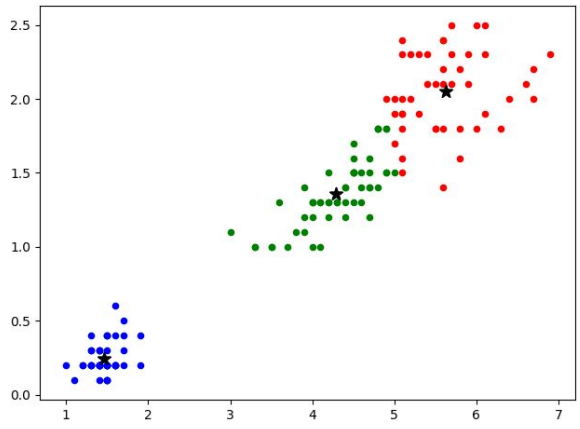
The clustering for these features is almost the same as the plotting in question 1 except for a few points here and there. The k-means for this example made a pretty good job in grouping the points together.

Petal Width against Sepal Width

Question 1 result	K-means result
	

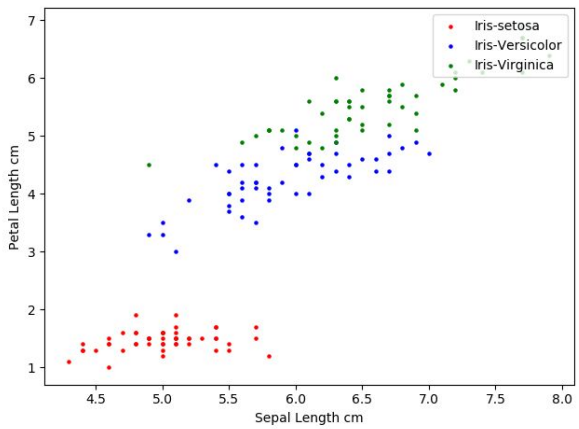
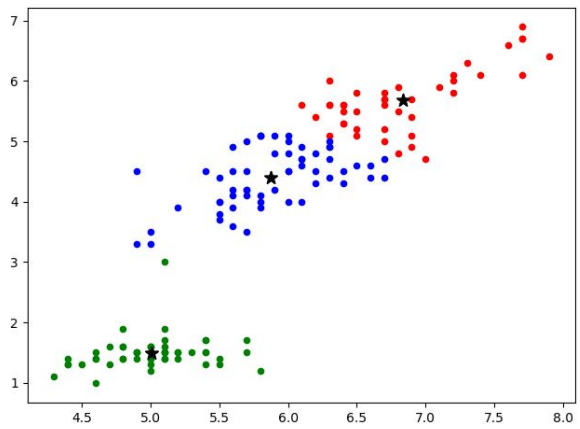
In this case, the clusters are also very similar to the plotting of different classes in question 1. Some of the points could be in more than one cluster and therefore there could be a bit of inaccuracy because of that.

Petal Width against Petal Length

Question 1 result	K-means result
	

The points are clustered really well here as well. There is less overlapping of classes in question 1's result, which is why the groups in the k-means result could be more accurate.

Sepal Length and Petal Length

Question 1 result	K-means result
	

This is a bit less accurate, as the points of Iris-virginica spread out over Iris-versicolor's points. Hence, the grouping of the red cluster missed out on some points as they are in the region of the blue cluster's centroid.

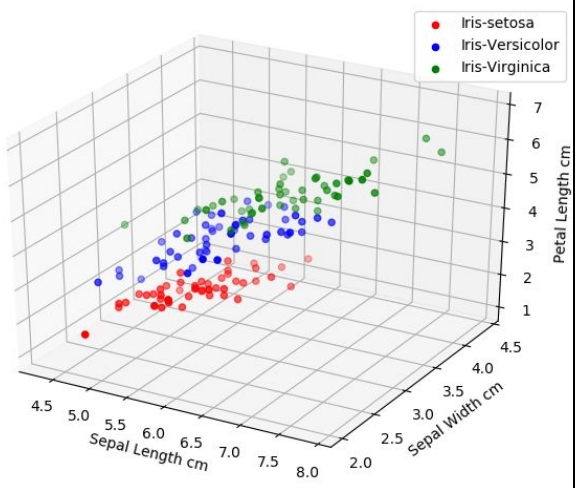
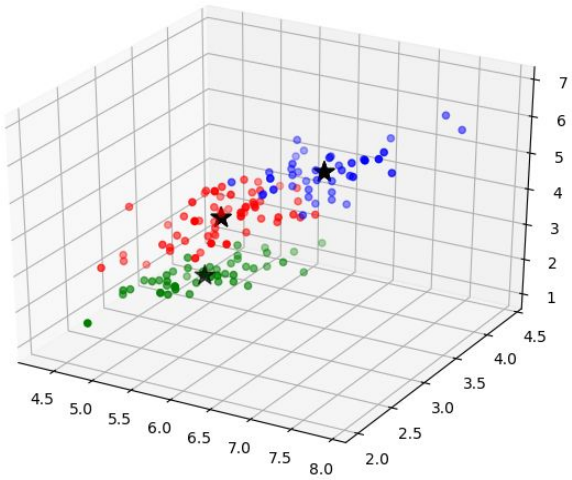
Overall Evaluation

Overall the k-means performed a good clustering of points in 2 Dimensional space. It outputted more accurate results when the classes do not overlap a lot each other, therefore, in such cases the grouping is very close to the plotting in question 1. Iris-setosa class in all cases is plotted at a distance away from the other classes and does not intertwine with them. As a result, the points of this class are grouped very well in each example.

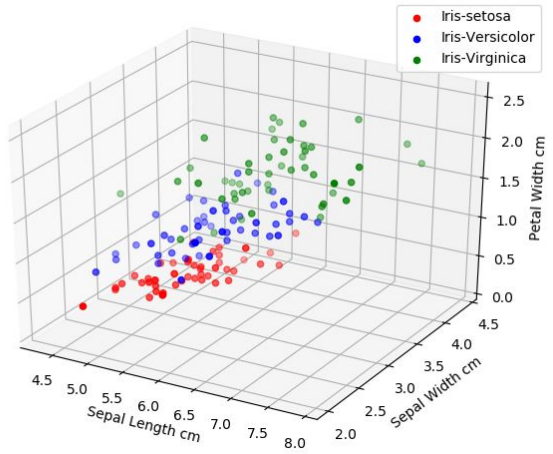
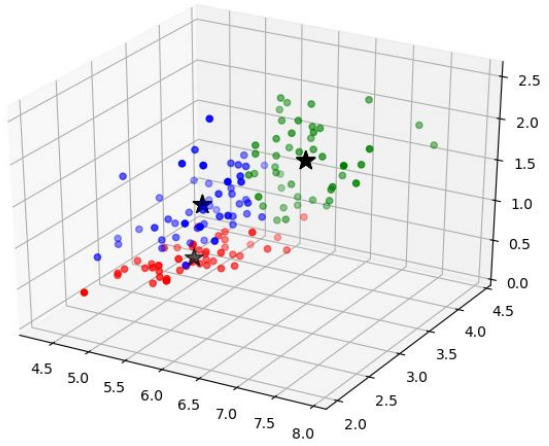
3 Dimensional

These plots will be evaluated all in one paragraph as the same conclusions could be drawn for all of them.

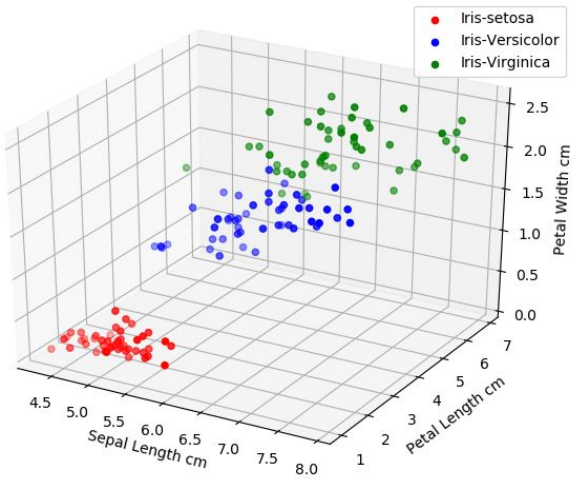
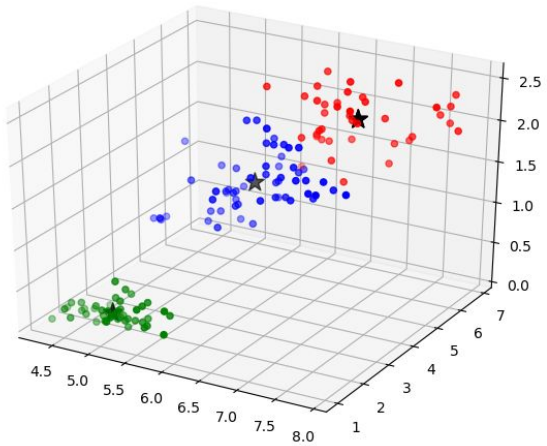
Sepal length, Sepal width and Petal length plotted against each other

Question 1 result	K-means result
	

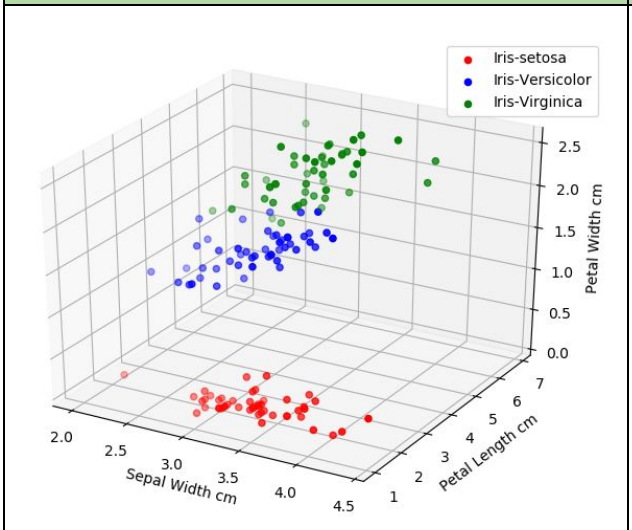
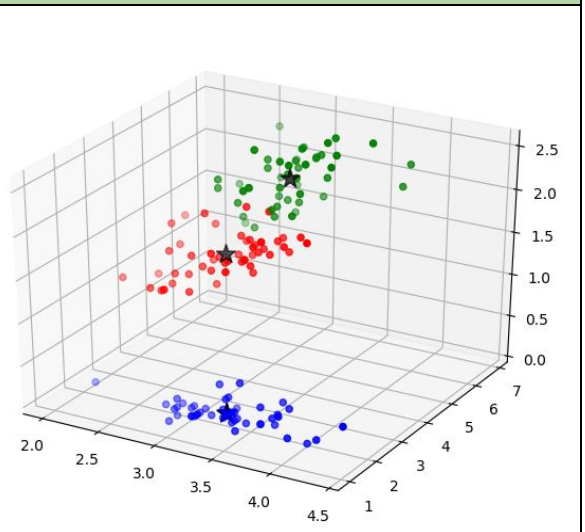
Sepal Length, Sepal width, Petal Width all plotted against each other

Question 1 result	K-means result
	

Sepal Length, Petal Width and Petal Length all plotted against each other

Question 1 result	K-means result
	

Sepal Width, Petal Length and Petal Width all plotted against each other

Question 1 result	K-means result
	

Overall Evaluation

Just like in the 2 Dimensional outputs, the k-means performed a good grouping of points in 3D. Once again, it outputted more accurate results when the classes do not overlap a lot each other, which means that, in such cases the clustering is very similar to the plotting in question 1. The points of Iris-setosa are also the best to be clustered as they rarely crossover other classes' points.

K-NN (Artifact 3)

Statement of completion

Attempted and completed.

Overview of algorithm

Breaking it Down – Pseudo Code of KNN

We can implement a KNN model by following the below steps:

1. Load the data
2. Initialise the value of k
3. For getting the predicted class, iterate from 1 to total number of training data points
    1. Calculate the distance between test data and each row of training data. Here we will use Euclidean distance as our distance metric since it's the most popular method. The other metrics that can be used are Chebyshev, cosine, etc.
    2. Sort the calculated distances in ascending order based on distance values
    3. Get top k rows from the sorted array
    4. Get the most frequent class of these rows
    5. Return the predicted class

I mostly followed the algorithm shown above taken from the website below.
https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/

Algorithm implemented:

1. Get a % of the dataset as training data, and the other % as test data. I set the split equal to 0.8 so that it gets a good amount of training data, and eventually the K-nn on the testing data will develop more accurate results.
2. Set k equal to the square root of the number of instances in the training data
3. Loop through all test data
    a. Calculate distance from the test instance to each training data instance
    b. Sort the distances in ascending order
    c. Get the first k distances and store them as nearest neighbours
    d. Get the most frequent class from the nearest neighbours list
    e. Print results

Code explanation

question3.py

main() function

Start by declaring an array for the training set and and array for the testing set and set the split to 0.8. Read from text file and store each line of data in one of the 2 arrays.

```
def main():
    trainingData = []
   testingData = []
   split = 0.8 #set to 0.8 to generate accurate results

   #read text file
   with open('iris.data.txt', 'r') as csvfile:
       lines = csv.reader(csvfile, delimiter=',')
       dataset = list(lines) #stored in an array, each element contains a line
from the txt file
       #randomly splitting the dataset into training data and test data
       for x in range(len(dataset) - 1):
           for y in range(4):
                dataset[x][y] = float(dataset[x][y])
           if random.random() < split:
               trainingData.append(dataset[x])
           else:
               testingData.append(dataset[x])
```

Tell the user the number of training instances and testing instance which are going to be used.

```
#output amount of training set and test set for the user to know
print('Number of data for training set: ' + repr(len(trainingData)))
print('Number of data to test: ' + repr(len(testingData)))
```

And now it is ready to start with the K-NN algorithm. It starts by looping through all the test data and at each iteration perform the following: get the nearest neighbours and get the most frequent class of these neighbors. Then output results and the test instance.

```
#output amount of training set and test set for the user to know
print('Number of data for training set: ' + repr(len(trainingData)))
print('Number of data to test: ' + repr(len(testingData)))

for x in range(len(testingData)): # for each testing data
   neighbors = nearest_neighbors(testingData[x], trainingData) #get the nearest
neighbors
   result = getResult(neighbors) #get the class the test set should belong to
```

```
    #print results and test data to compare the actual class with the predicted
one
    print('Test data: ')
    print(testingData[x])
    print('K-NN result=' + result)
input("Press enter to exit ;)")
```

calculate_distance() function

This method calculated the euclidean distance by implementing this mathematical formula:

$$= \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$

$$= \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}.$$

```
def calculate_distance(p, q, length): #calculates euclidean distance
    distance = 0
    for i in range(length):#for all test data
        distance += np.square(p[i] - q[i]) #calculate this part of equation:
(p-q)^2 + (b-a)^2+....
    return np.sqrt(distance) #square root the total of distances
```

nearest_neighbors() function

Start by finding the value of k by applying the square root on the length of training data set. Then, for each training data instance, calculate the distance between it and the test data instance. When all distances are found, sort the array storing all distances, in ascending order. Loop through the first k elements of the array and store them in the neighbors array. These are the nearest neighbors to the test instance.

```
def nearest_neighbors(testData, trainingData):
    k = math.sqrt(len(trainingData)) #square root of the amount of training
data
    k = int(k)#in case the answer is float change it to integer

    distances = []
    length = len(testData)-1 # length of testData minus one as it starts from 0

    #for every training data instance
    for data in range(len(trainingData)):
        #calculate the distance between the testing instance and the training
data
```

```
        distance = calculate_distance(testData, trainingData[data], length)
        #populate the array with the distances and their corresponding training
data
        distances.append((trainingData[data], distance))

    #sort the array distances in ascending order
    distances.sort(key=operator.itemgetter(1))

    nearest_neighbors = []
    #store the first k distances which are the smallest
    for kvalue in range(k):
        nearest_neighbors.append(distances[kvalue][0])

    return nearest_neighbors
```

getResult() function

Loop through array of the nearest neighbours and store their class in another array and then check which is the class that is mostly listed in the array.

```
def getResult(neighbors):
    neighborClasses = []
    #for all first k distances (loops through all neighbours)
    for x in range(len(neighbors)):
        flowerClass = neighbors[x][-1] #gets the class of each neighbor i.e
Iris-setosa, iris-virginica or Iris-versicolor
        neighborClasses.append(flowerClass) #add each flower class to the array
of all found classes
    result = max(neighborClasses, key = neighborClasses.count) #find most common
class in the array
    return result
```

Evaluation

First the number of training data used and testing instances used are printed. Then for each test instance, the data being tested is printed, showing also to which class it really belongs to. Right after it the result generated by the K-NN is outputted. The K-NN result can be compared with the actual class to check if the algorithm works good.

Output after first code execution:

```
Number of data for training set: 124
Number of data to test: 25
Test data:
[4.7, 3.2, 1.3, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.0, 3.4, 1.5, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.7, 4.4, 1.5, 0.4, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.4, 3.9, 1.3, 0.4, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.1, 3.7, 1.5, 0.4, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.2, 4.1, 1.5, 0.1, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.5, 3.5, 1.3, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[6.0, 2.2, 4.0, 1.0, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.7, 3.1, 4.4, 1.4, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.2, 2.2, 4.5, 1.5, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.9, 3.2, 4.8, 1.8, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.6, 2.7, 4.2, 1.3, 'Iris-versicolor']
K-NN result=Iris-versicolor
```

Test data:
[5.7, 3.0, 4.2, 1.2, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.8, 2.7, 5.1, 1.9, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[7.6, 3.0, 6.6, 2.1, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.5, 3.2, 5.1, 2.0, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.8, 3.0, 5.5, 2.1, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.5, 3.0, 5.5, 1.8, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.9, 3.2, 5.7, 2.3, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[5.6, 2.8, 4.9, 2.0, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.4, 2.8, 5.6, 2.1, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[7.9, 3.8, 6.4, 2.0, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.0, 3.0, 4.8, 1.8, 'Iris-virginica']
K-NN result=Iris-versicolor
Test data:
[6.9, 3.1, 5.1, 2.3, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.8, 3.2, 5.9, 2.3, 'Iris-virginica']
K-NN result=Iris-virginica

From all 25 test data, only one of them gave an inaccurate result which is marked in red.

Output after second code execution:

```
Number of data for training set: 121
Number of data to test: 28
Test data:
[4.7, 3.2, 1.3, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[4.6, 3.4, 1.4, 0.3, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.0, 3.4, 1.5, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[4.4, 2.9, 1.4, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[4.8, 3.4, 1.6, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.4, 3.9, 1.3, 0.4, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.0, 3.0, 1.6, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.4, 3.4, 1.5, 0.4, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.1, 3.4, 1.5, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[7.0, 3.2, 4.7, 1.4, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.5, 2.8, 4.6, 1.5, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.1, 2.9, 4.7, 1.4, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.6, 2.9, 3.6, 1.3, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.1, 2.8, 4.0, 1.3, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.1, 2.8, 4.7, 1.2, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
```

[5.7, 2.6, 3.5, 1.0, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.0, 2.7, 5.1, 1.6, 'Iris-versicolor']
K-NN result=Iris-virginica
Test data:
[6.7, 3.1, 4.7, 1.5, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.5, 2.5, 4.0, 1.3, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.5, 2.6, 4.4, 1.2, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.1, 2.5, 3.0, 1.1, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[7.7, 2.6, 6.9, 2.3, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.2, 2.8, 4.8, 1.8, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.4, 2.8, 5.6, 2.1, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[7.9, 3.8, 6.4, 2.0, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.1, 2.6, 5.6, 1.4, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.7, 3.1, 5.6, 2.4, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.9, 3.1, 5.1, 2.3, 'Iris-virginica']
K-NN result=Iris-virginica

Here we have a bit less training data, but the results show that only one of them generated a wrong result, just like the first time.

The below is an example of the output in the case when the split in the code is changed to 0.6. This gets less training instances and more test instances.

```
Number of data for training set: 78
Number of data to test: 71
Test data:
[5.1, 3.5, 1.4, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[4.6, 3.1, 1.5, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.0, 3.6, 1.4, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[4.6, 3.4, 1.4, 0.3, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[4.4, 2.9, 1.4, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[4.8, 3.4, 1.6, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[4.8, 3.0, 1.4, 0.1, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[4.3, 3.0, 1.1, 0.1, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.8, 4.0, 1.2, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.4, 3.9, 1.3, 0.4, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[4.8, 3.4, 1.9, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.2, 3.5, 1.5, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.2, 3.4, 1.4, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[4.7, 3.2, 1.6, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[4.8, 3.1, 1.6, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
```

Test data:
[4.9, 3.1, 1.5, 0.1, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.0, 3.2, 1.2, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.5, 3.5, 1.3, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.1, 3.8, 1.6, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[4.6, 3.2, 1.4, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[5.0, 3.3, 1.4, 0.2, 'Iris-setosa']
K-NN result=Iris-setosa
Test data:
[6.9, 3.1, 4.9, 1.5, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.5, 2.3, 4.0, 1.3, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.7, 2.8, 4.5, 1.3, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.3, 3.3, 4.7, 1.6, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.0, 2.0, 3.5, 1.0, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.0, 2.2, 4.0, 1.0, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.6, 3.0, 4.5, 1.5, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.8, 2.7, 4.1, 1.0, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.2, 2.2, 4.5, 1.5, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.1, 2.8, 4.0, 1.3, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.3, 2.5, 4.9, 1.5, 'Iris-versicolor']

K-NN result=Iris-virginica
Test data:
[6.1, 2.8, 4.7, 1.2, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.8, 2.8, 4.8, 1.4, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.7, 3.0, 5.0, 1.7, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.5, 2.4, 3.7, 1.0, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.0, 2.7, 5.1, 1.6, 'Iris-versicolor']
K-NN result=Iris-virginica
Test data:
[6.0, 3.4, 4.5, 1.6, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.3, 2.3, 4.4, 1.3, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.5, 2.5, 4.0, 1.3, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.5, 2.6, 4.4, 1.2, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.1, 3.0, 4.6, 1.4, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.8, 2.6, 4.0, 1.2, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.6, 2.7, 4.2, 1.3, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.7, 3.0, 4.2, 1.2, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.7, 2.9, 4.2, 1.3, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[5.1, 2.5, 3.0, 1.1, 'Iris-versicolor']
K-NN result=Iris-versicolor
Test data:
[6.3, 3.3, 6.0, 2.5, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:

[5.8, 2.7, 5.1, 1.9, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[7.1, 3.0, 5.9, 2.1, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[5.7, 2.5, 5.0, 2.0, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[5.8, 2.8, 5.1, 2.4, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.4, 3.2, 5.3, 2.3, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.5, 3.0, 5.5, 1.8, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[7.7, 3.8, 6.7, 2.2, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[7.7, 2.6, 6.9, 2.3, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.9, 3.2, 5.7, 2.3, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[5.6, 2.8, 4.9, 2.0, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.3, 2.7, 4.9, 1.8, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[7.2, 3.2, 6.0, 1.8, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.1, 3.0, 4.9, 1.8, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[7.2, 3.0, 5.8, 1.6, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[7.9, 3.8, 6.4, 2.0, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.4, 2.8, 5.6, 2.2, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.3, 2.8, 5.1, 1.5, 'Iris-virginica']
K-NN result=Iris-virginica

Test data:
[6.0, 3.0, 4.8, 1.8, 'Iris-virginica']
K-NN result=Iris-versicolor
Test data:
[6.7, 3.1, 5.6, 2.4, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.9, 3.1, 5.1, 2.3, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.8, 3.2, 5.9, 2.3, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.7, 3.0, 5.2, 2.3, 'Iris-virginica']
K-NN result=Iris-virginica
Test data:
[6.5, 3.0, 5.2, 2.0, 'Iris-virginica']
K-NN result=Iris-virginica

In this case, a smaller split was assigned and as can be seen, the results are more inaccurate (marked in red) than they were with a 0.8 split. This is because less data is provided to train.

Overall evaluation.

From the output, it can be concluded that the k-nn algorithm works for most of the data, except for a few here and there. The k is calculated by performing the square root of the number of training data, because with each execution the % of training data is different. Therefore, the K has to adapt according to the amount of data being trained to eventually generate better results. It was also noted that a split of 0.8 gives better responses than smaller splits.

Overall evaluation and conclusion

Artifact 1

Artifact 1 works fine. The plots are generated as they should be for all different dimensions. One weakness is that some points cannot be seen as other points overlap them and hide them, so it sometimes can mislead you while interpreting them. In every dimensional space, the graphs show consistency between each other, having Iris-setosa being the smallest, followed by Iris-versicolor which is always placed in between and then Iris-virginica which is relatively the biggest flower.

Artifact 2

Overall the k-means performed good clustering of points. It was noted that it performed best when the classes do not overlap a lot each other. Iris-setosa class in all cases is plotted at a distance away from the other classes and does not intertwine with them. As a result, the points of this class are grouped very well in each example.

Artifact 3

The k-nn algorithm returns accurate results for most of the test data, except for a few here and there. The k is calculated by performing the square root of the number of training data, because with each execution the % of training data is different. Therefore, the K has to adapt according to the amount of data being trained to eventually generate better results. It was also noted that a split of 0.8 gives better responses than smaller splits.

These 3 artifacts are a great start for implementing machine learning techniques and help you understand more where these can be used and how.