Programming Assignment 2
Kevin Bruce
3/1/22

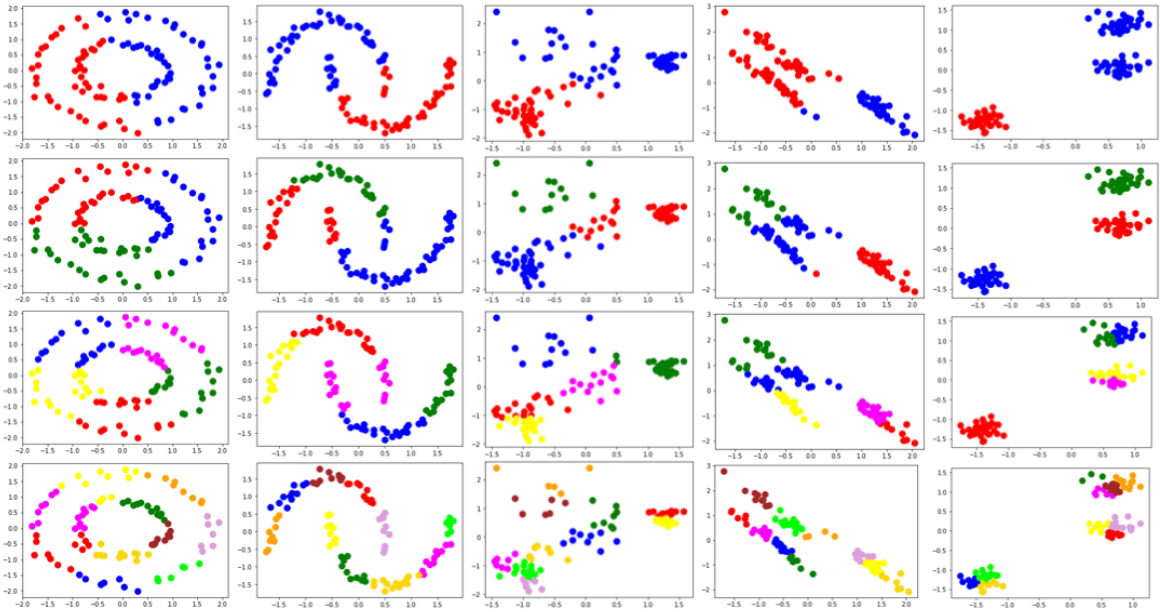
When creating the series of plots, I couldn't figure out how to correctly place each plot in the correct ordering of the table that was desired for 1(C), 4(B), and 4(C). Therefore, I created each plot separately and organized them by hand to get the desired table of plots.

1.(A). The data sets were imported correctly. See code below

(B). The data was standardized correctly and the k-means worked correctly. Instead of designing a function to do all of it at once, I just did it separately. My function kept messing up somewhere, so I decided to do it this way in order to have correct results. I had to organize the plots by hand.

(C). The following are the results of k-means clustering on the five datasets: Noisy Circles, Noisy Moons, Blobs with Varied Variances, Anisotropicly Distributed Data, and Blobs. Each row in the figure represents a different value of k=2,3,5,10.

Noisy Circles	Noisy Moons	Blobs with Varied Variances	Anisotropicly distributed data	Blobs



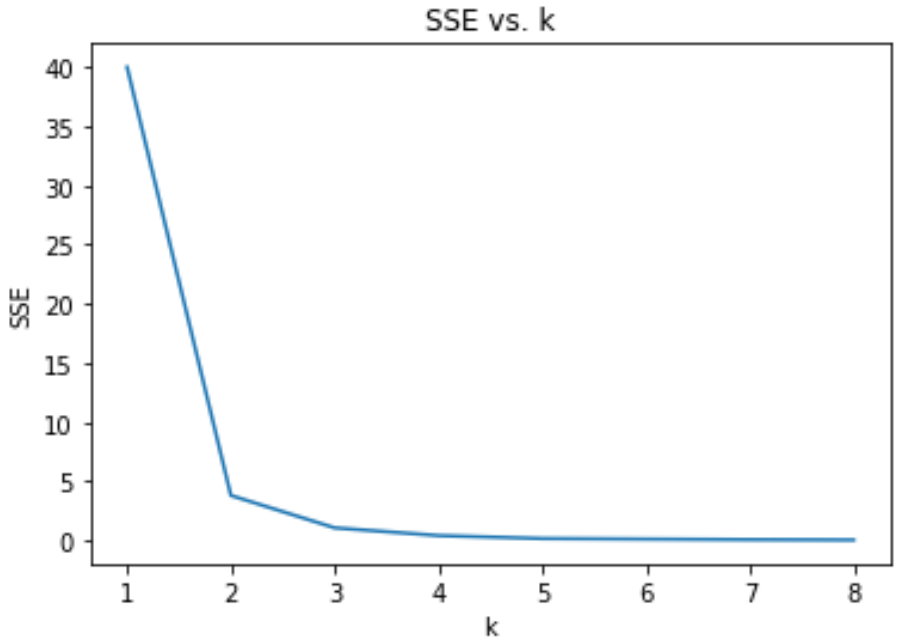
From this plot it is quite clear that k-means clustering fails to produce the correct clusters (assuming k is the correct number of groups) for the Noisy Moons, Noisy Circles, Anisotropicly Distributed Data, and the Blobs with Varied Variances. However, the Blobs with Varied Variances seems to be fairly close (when k=3.) K-means clustering appears to have correctly clustered the Blobs dataset.

(D). When repeating the k=2, 3 cases over the datasets, none of the data sets appear to be too sensitive to the choice of initialization. However, the cluster groups did change only slightly for all of the datasets, except for the Blobs dataset. The Blobs data set's clusters when k=2 and 3 did not change at all over the repeated computation of the k-means clustering. All of the other datasets changed slightly over repeated application of the k-means clustering when k=2 and 3.

2(A). I correctly used the make_blobs() function with the correct parameters. See code below.

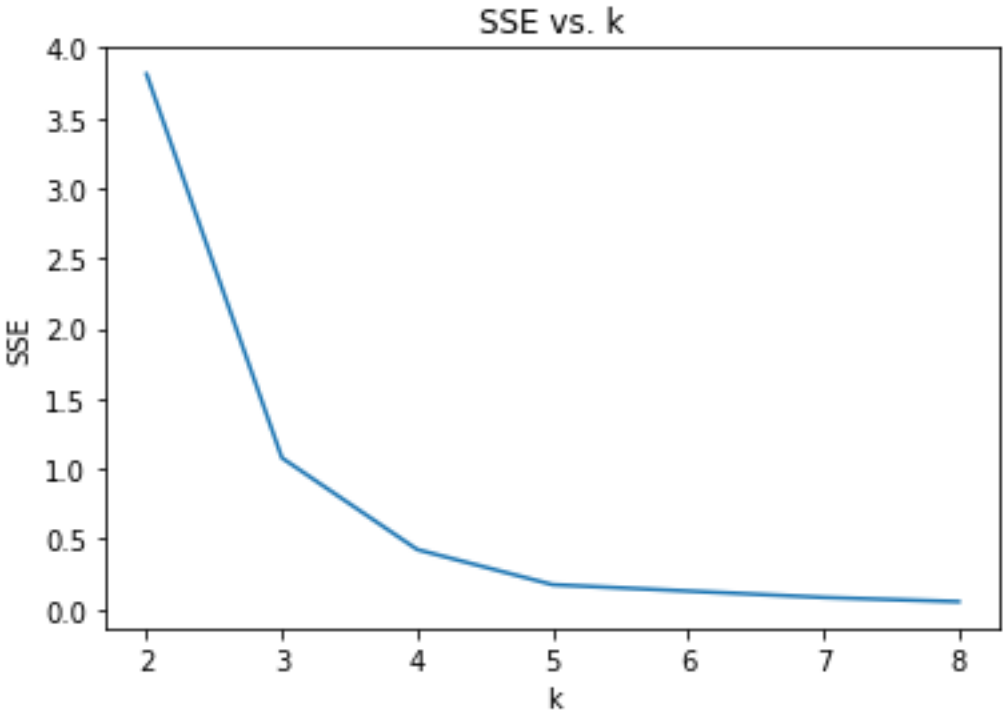
(B). Instead of modifying the entire function, I just created a for loop to compute the SSE and inertia for the various values of k=1, 2, 3, 4, 5, 6, 7, and 8. See the code below.

(C). After computing the sum of squares error over the values of k, we get the following plot of SSE vs k:



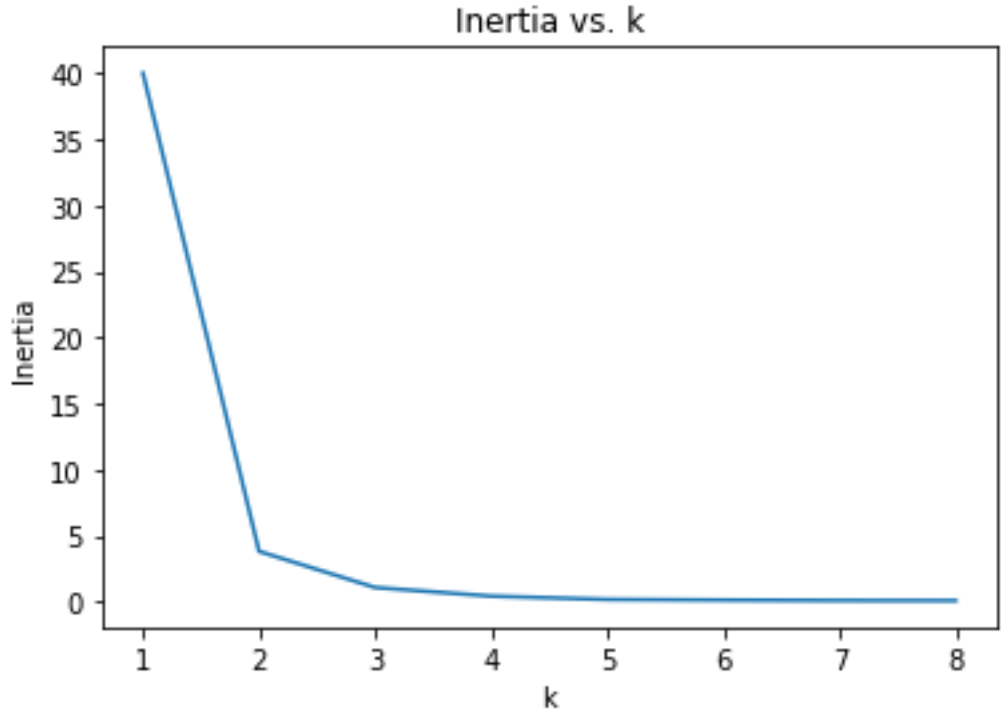
Since the SSE when k=1 is fairly large, lets remove it and see what the graph looks like.

Plot on next page!

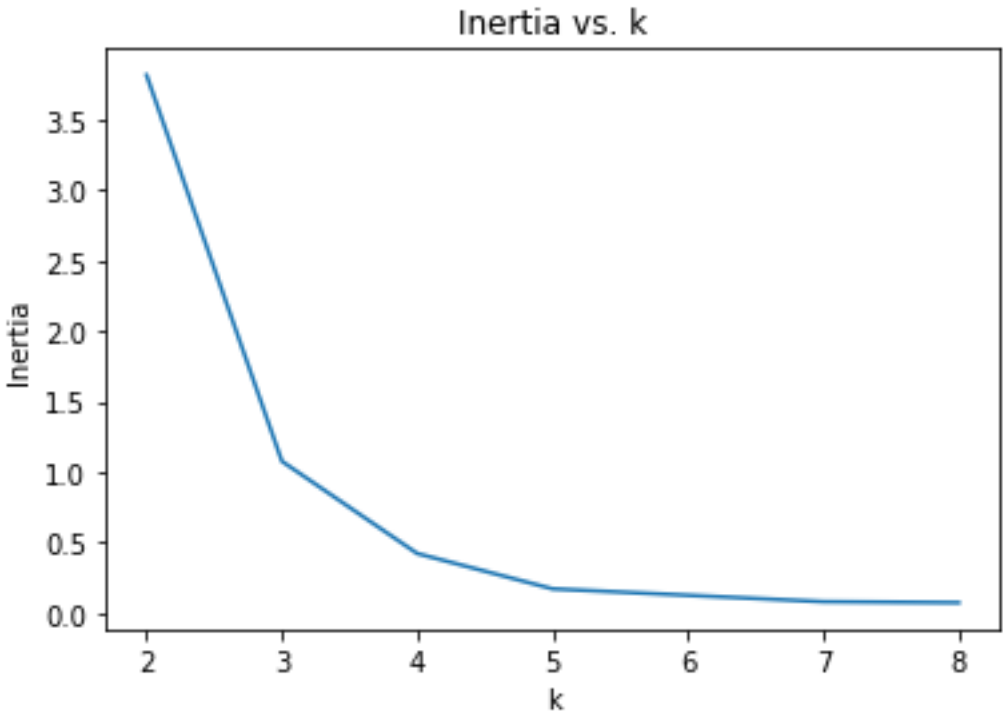


From this plot and using the elbow method, it is quite clear that five clusters are needed for this dataset. This was not obvious from the first plot as the k=1 case caused our graph to zoom out too much.

(C). Now consider using inertia instead of SSE:



Likewise with the plot of inertia, we will need to exclude the value when k=1. The graph becomes:



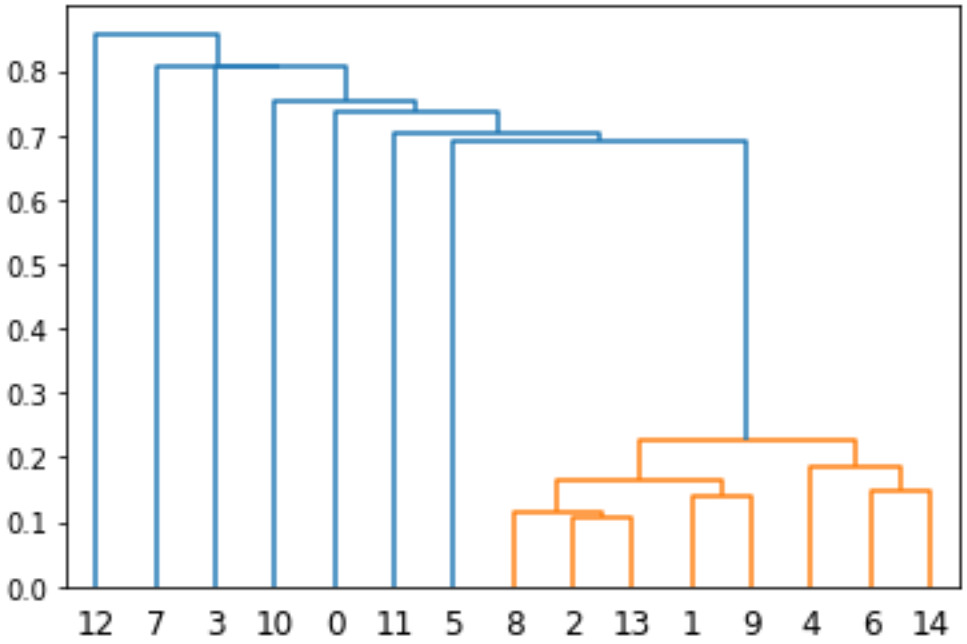
Therefore, we see that at k=5 (using the elbow-method) we have the ideal amount of clusters. Also, we see that using SSE or inertia, we get the same value of k using the elbow method.

3. (A). The data set was loaded in correctly. See code below. In the following problems, the values in the cluster are given by the index of them in the array.

(B). The linkage matrix was created and the dendrogram was plotted. The linkage matrix is:

	0	1	2	3
0	2	13	0.107777	2
1	8	15	0.115663	3
2	1	9	0.138799	2
3	6	14	0.148695	2
4	16	17	0.166462	5
5	4	18	0.183936	3
6	19	20	0.226643	8
7	5	21	0.692326	9
8	11	22	0.701342	10
9	0	23	0.734284	11
10	10	24	0.752084	12
11	3	25	0.805415	13
12	7	26	0.805427	14
13	12	27	0.85709	15

The dendrogram is:



(C). From the linkage matrix, Z, we see that {I={8,2,13}, J={1,9}} were merged at row 4 (starting from 0). This means that the fourth iteration (starting with the 0th iteration) merged these two clusters together. In other words, the fifth iteration (when 0th iteration is listed as 1st iteration) merged these two clusters together.

(D). The function I created that computes the dissimilarity is seen in the code below. When running this function on the same clusters as in 3(C) we get the same value as in the linkage matrix for this merger. In other words, the value computed by the function I created matches with linkage(X_train3)[4,2]. Both values were approximately 0.16646190977808645.

(E). All of the available clusters when {8, 2, 13} and {1, 9} were merged were:
{6,14}, {4}, {5}, {11}, {0}, {10}, {3}, {7}, and {12}.

(F). Based on the dendrogram, we will not get the correct clustering because not all of the inner entries are grouped in the yellow/orange section in the dendrogram. Considering the locations of each entry in a scatterplot and where these entries appear in the dendrogram, Entry 11 should be grouped with the yellow section as it is a point in the inner part of the data. Thus, we will not get the correct clustering, but the clustering is extremely close to being correct.

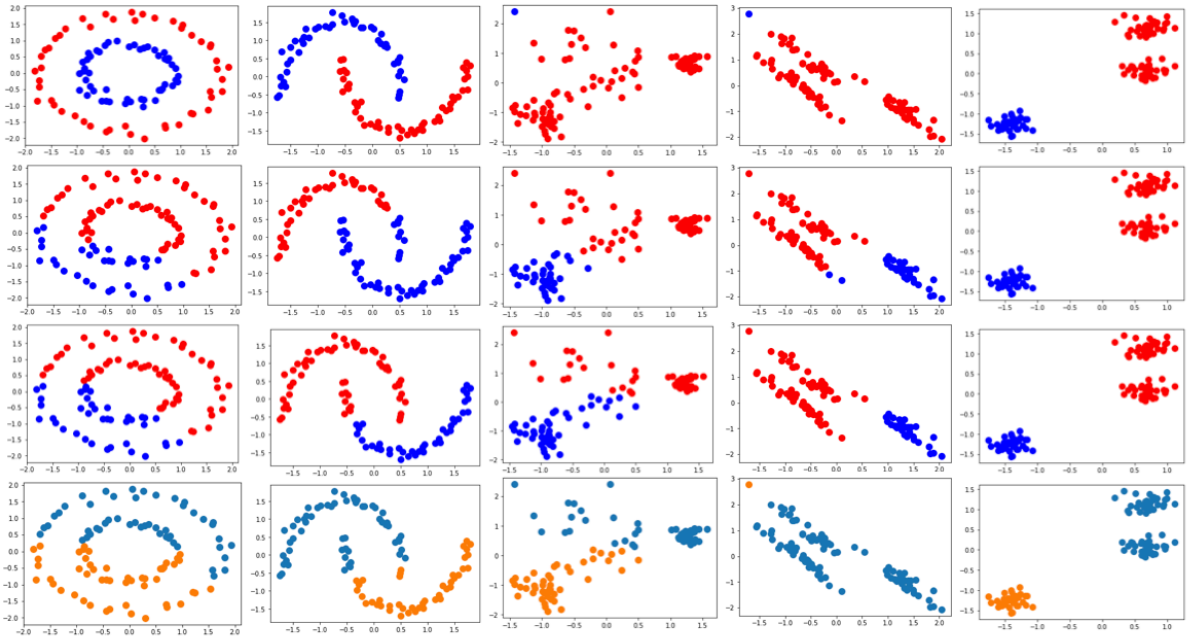
(G). It appears that my dendrogram is somewhat exhibiting this phenomenon. The dendrogram and linkage matrix shows that the orange/yellow group link together first. Thus, it is showing

that one cluster is continuously merging with available points. However, the other points are also being clustered together separately from the other cluster. However, these points are clustered together after the other cluster has essentially finished clustering together. Thus, my dendrogram is essentially showing this phenomenon.

4. (A). See code below for how I implemented agglomerative clustering. I did not create a separate function. I just did each case individually.

(B). Here are my results of agglomerative clustering on the five different datasets.

Noisy Circles	Noisy Moons	Blobs with Varied Variances	Anisotropicly Distributed Data	Blobs

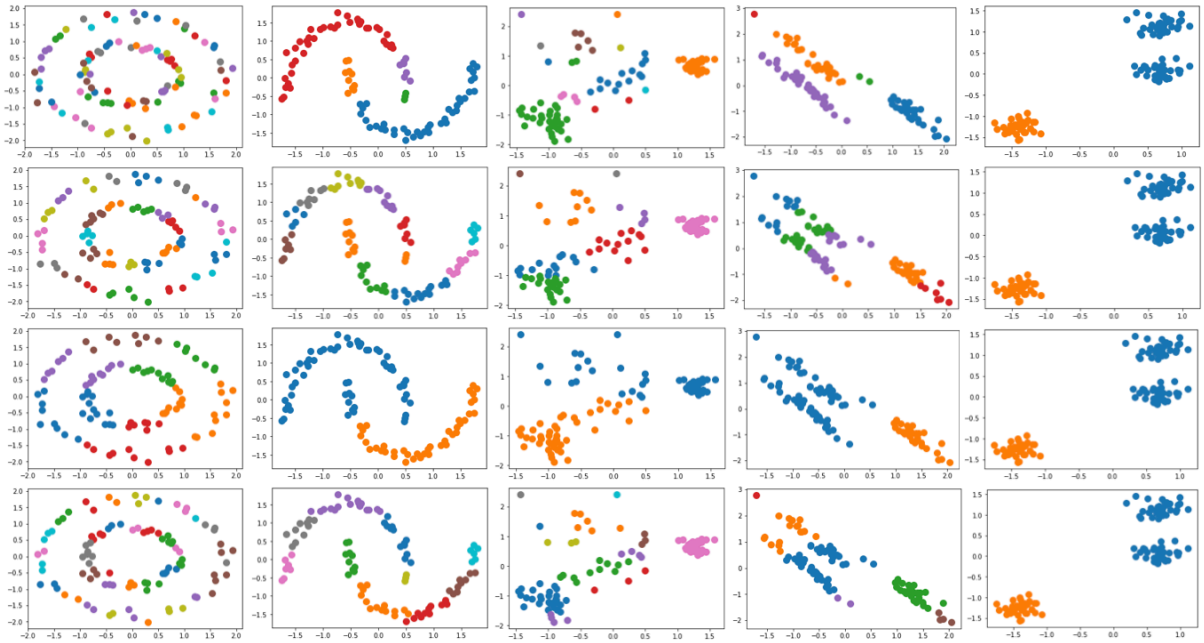


Each column is a different dataset. Each row is a different linkage type in the clustering. Row 1 is single; row 2 is complete; row 3 is ward; row 4 is average. For the noisy circles and noisy moons datasets, the single linkage type with 2 clusters seems to cluster the data set perfectly. These were not clustered correctly in number 1. The other linkage types did not cluster these datasets correctly though. The other three datasets will never be correctly clustered since there should be three different clusters for these sets. In this part we set the number of clusters to two. For the blobs dataset, we got the same exact results as we see in 1(C) when k=2 regardless of linkage type. For blobs with varied variances and anisotropicly distributed data, we see different clustering results than in k-means when using the single linkage type. Both of these place every

point except for one in one cluster. The rest of the linkage types produced results that were more similar to k-means when k=2, except when we use the complete linkage type.

(C). Now repeating these agglomerative clusters but using a distance threshold to stop the joining of clusters when distances get too large, we get the following results:

Noisy Circles	Noisy Moons	Blobs with varied variances	Anisotropicly Distributed Data	Blobs



The structure of this table is the same as in 4(B). Row 1 is single; row 2 is complete; row 3 is ward; row 4 is average. However, k is not constant. Our value of k depends on the distance threshold computed by looking at the maximum finite difference of successive cluster joins. This is computed using the numpy.diff() function on the third column of the linkage matrix for the respective linkage type. The number of clusters for noisy circles in each plot (going down the specific column) are 80, 32, 6, and 38 respectively. In other words, agglomerative clustering produced 80 clusters when the linkage type of single was used with the maximum difference being the largest finite difference of the third column in the linkage matrix. Since we have so many clusters, our max distance threshold must be too small so we are stopping the joining of clusters too soon. For noisy moons, the number of clusters were 5, 12, 2, and 10 respectively. For blobs with varied variances, the number of clusters were 14, 8, 2, and 15 respectively. For the anisotropicly distributed data, the number of clusters were 5, 5, 2, and 6 respectively. For the blobs, the number of clusters were two for all of the linkage types. We see that using this, we

never get the correct clustering for noisy circles because the distance thresholds were too small. Thus we stopped joining the cluster too early, resulting in a large amount of clusters. Recall that there should be two clusters. For noisy moons, we never correctly clustered the data. However, we were closer to the true amount of clusters; we even achieve this when using the ward linkage type, but the clustering is not correct. Likewise, with blobs of varied variances, anisotropicly distributed data, and the blobs datasets. In addition, no matter the linkage type and the specific cutoff using that linkage type, we get the same clustering results every time with the blobs dataset.

Code:

```
#Import libraries
import time
import warnings
import numpy as np
import matplotlib.pyplot as plt
from sklearn import cluster, datasets, mixture
from sklearn.datasets import make_blobs
from sklearn.neighbors import kneighbors_graph
from sklearn.preprocessing import StandardScaler
from itertools import cycle, islice
import scipy.io as io
from scipy.cluster.hierarchy import dendrogram, linkage
#import plotly.figure_factory as ff
import math
from sklearn.cluster import AgglomerativeClustering

#Number 1
#Set Seed
np.random.seed(0)

#Generate datasets
#Number of samples
n_samples=100

#Noisy Circles
noisy_circles = datasets.make_circles(n_samples=n_samples, factor=0.5, noise=0.05)

#Noisy Moons
noisy_moons = datasets.make_moons(n_samples=n_samples, noise=0.05)

#Blobs with varied variances
random_state = 170
```

```
varied = datasets.make_blobs(n_samples=n_samples, cluster_std=[1.0, 2.5, 0.5],
random_state=random_state)

#Anisotropicly distributed data
X, y = datasets.make_blobs(n_samples=n_samples, random_state=random_state)
transformation = [[0.6, -0.6], [-0.4, 0.8]]
X_aniso = np.dot(X, transformation)
aniso = (X_aniso, y)

#blobs
blobs = datasets.make_blobs(n_samples=n_samples, random_state=8)

#Place data into training arrays
xtrain=blobs[0]
ytrain=blobs[1]

#Standardize Data
scaler=StandardScaler().fit(xtrain)
Xtrain=scaler.transform(xtrain)

#Define colors for graphing
colors=['red', 'blue', 'green', 'yellow', 'magenta','orange','brown','plum','gold','lime',
'slategray','teal']

#Function to compute kmeans clustering
def my_kmeans(Xtrain,ytrain, k):
    #Run K-Means
    kmeans=cluster.KMeans(n_clusters=k, init='random')
    kmeans.fit(Xtrain)
    #Predictions
    y_kmeans=kmeans.predict(Xtrain)
    #Return y_kmeans
    return(y_kmeans)

#Set k=2,3,5,10
k=2
#Run Function
y_kmeans=my_kmeans(Xtrain, ytrain, k=k)
#Graph
plt.figure()
for i in range(0,k):
    plt.scatter(Xtrain[y_kmeans==i, 0], Xtrain[y_kmeans==i, 1], s=100, c=colors[i])
```

```
#Number 2
#Make Blobs
X, y = make_blobs(center_box=(-20,20), n_samples=20, centers=5, random_state=12)

#Standardize Data
scaler2=StandardScaler().fit(X)
X_train=scaler2.transform(X)

#Kmeans
kk=[1,2,3,4,5,6,7,8]


#Initialize vector to hold SSE and Inertias
SSE=[]
SSER=[]

#Run Kmeans
for j in kk:
   kmeans2=cluster.KMeans(n_clusters=j, init='random')
   kmeans2.fit(X_train)

   #Predictions
   y_kmeans2=kmeans2.predict(X_train)


   #Collect SSER (Real SSE) and plot
   cluster_centers = [X_train[kmeans2.labels_ == i].mean(axis=0) for i in range(j)]
   clusterwise_sse = [0]*j
   for point, label in zip(X_train, kmeans2.labels_):
      clusterwise_sse[label] += np.square(point - cluster_centers[label]).sum()
   SSER.append(np.sum(clusterwise_sse))

   #Inertia
   SSE.append(kmeans2.inertia_)

#Plot SSE
plt.plot(kk[1:len(kk)],SSER[1:len(SSER)])
plt.title("SSE vs. k")
plt.xlabel("k")
plt.ylabel("SSE")
```

```
#Plot Inertia
plt.plot(kk[1:len(kk)],SSE[1:len(SSE)])
plt.title("Inertia vs. k")
plt.xlabel("k")
plt.ylabel("Inertia")


#Number 3
#Import Dataset
data=io.loadmat('hierarchical_toy_data.mat')
X_train3=data['X']

#Compute linkage matrix
Z=linkage(X_train3)

#Compute dendrogram of linkage matrix
dendrogram(Z)

#Function to compute distance
def distance3(i,j,s,k,m):
    #Grab data points associated with the clusters
    x1=[X_train3[i,0], X_train3[i,1]]
    x2=[X_train3[j,0], X_train3[j,1]]
    x3=[X_train3[s,0], X_train3[s,1]]
    x4=[X_train3[k,0], X_train3[k,1]]
    x5=[X_train3[m,0], X_train3[m,1]]
    #Compute min dist
    #First three points are in same cluster, last two in another

distance=min(math.dist(x1,x4),math.dist(x1,x5),math.dist(x2,x4),math.dist(x2,x5),math.dist(x3,x
4),math.dist(x3,x5))
    #Output this distance
    return(distance)

#Compute Distance
distance3(8,2,13,1,9)



#Number 4
#Specify number of clusters
num_clust=2
```

```
#For 4C, compute cut-off distance using linkage matrix
method='average'
link4=linkage(Xtrain, method=method)
dist4=link4[:,2]
link4_fd=np.diff(dist4)
cut_off_dist=max(link4_fd)

#create heirarchical clustering model
#aggclust=AgglomerativeClustering(n_clusters=num_clust, linkage='average')

#For 4C
aggclust=AgglomerativeClustering(n_clusters=None, linkage=method,
distance_threshold=cut_off_dist)

#Fit and predict using the model
y_aggclust=aggclust.fit_predict(Xtrain)

#Output the graph of the clustering results
for i in range(0,len(np.unique(y_aggclust))):
    plt.scatter(Xtrain[y_aggclust==i, 0], Xtrain[y_aggclust==i, 1], s=100)

#Output number of clusters
len(np.unique(y_aggclust))
```