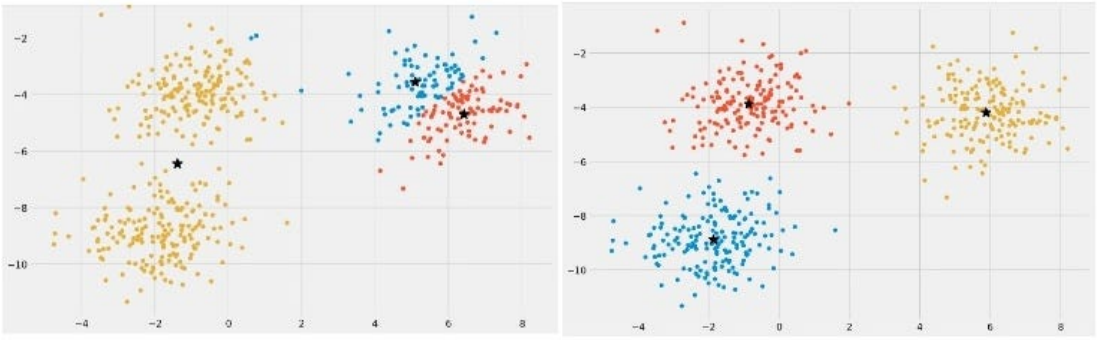
1.K-means++ algorithm:

(a) Explain the theory behind K-means++ in your words, including how it differs from the original K-means algorithm. Use graphical explanations wherever possible.

We know K-means is sensitive to initialization. How the initial centroids are chosen can greatly effect the K-mean cluster result. Below Left-plot is the formed cluster using bad initial centroids, and the right-plot is the ideal cluster.



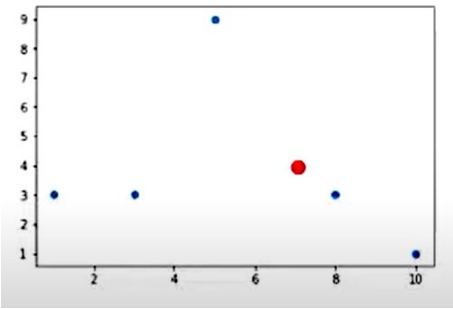
K-means++ is an improve version over the original K-means algorithm that selects the initial centroids in a smarter way to increase the chances of getting a better clustering result. In general, the way the centroids are selected is based on the maximum squared distance between the centroids and the rest points. The idea is to push the centroids as far as possible from one another.

Assume the number of clusters(K) is 3, K-means and K-means++ select the initial centroid in the following way:

	K-means	**K-means++**
Initial centroid 1	Select totally random	Select totally random
Initial centroid 2	Select totally random	1.Calculate the distance between the remaining points and centroid 1 2. Assign portability to each remaining points proportional to its distance to centroid 1 3. Select centroid 2 from the remaining points(each point having probability of being selected calculated in step2)
Initial centroid 3	Select totally random	1.Calculate the distance between the remaining points and their closest centorid(could be centroid 1 or2) 2.Assign portability to each remaining points proportional to its distance to centroid 1 or 2 based on step1 3.Select centroid 3 from the remaining points(each point having probability of being selected calculated in step2)

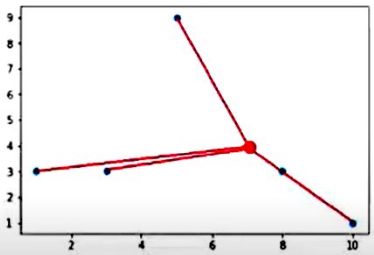
Example: Assume we have 6 points, and is trying to find 3 clusters using K-means++.
(example plots taken from https://www.youtube.com/watch?v=HatwtJSsj5Q)
Following the steps shown in table above:

Step 1: We total randomly pick one red point from the plot, which is (7,4)



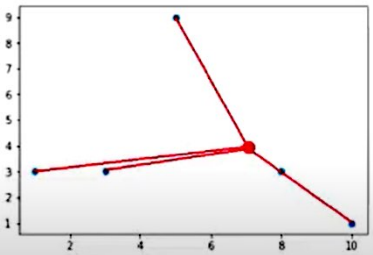
Step 2: We calculate the distance between the remaining point and red centroid 1

x	$\min(d(x, z_i)^2)$
(7,4)	-
(8,3)	2
(5,9)	29
(3,3)	17
(1,3)	37
(10,1)	18



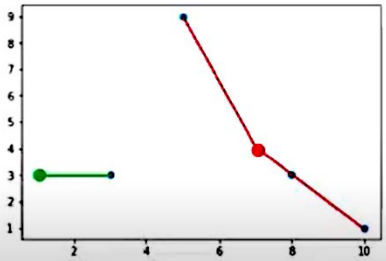
And assign probability according to the distance, the total distance/base is 2+29+17+37+18 = 103

x	prob
(7,4)	-
(8,3)	2/103
(5,9)	29/103
(3,3)	17/103
(1,3)	37/103
(10,1)	18/103



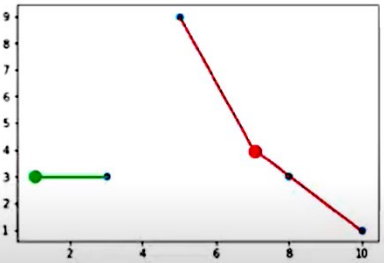
Step 3: Assume we randomly selected centroid (1,3) from the remaining 5 points, as it has the largest probability. We then calculate the distance between the remaining points and their closest centorid. For point (3,3) it is closest to centroid (1,3), and for the rest points, they are closest to centroid (7,4).

x	$\min(d(x, z_i)^2)$
(7,4)	-
(8,3)	2
(5,9)	29
(3,3)	4
(1,3)	-
(10,1)	18



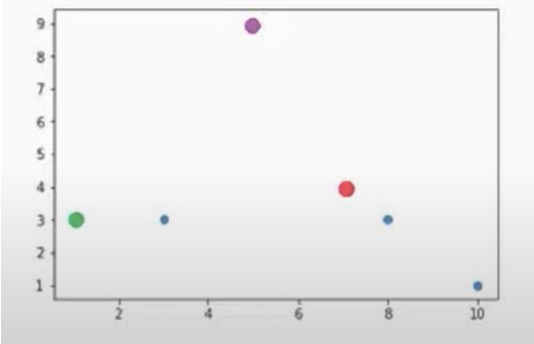
We then using the distance as probability of being selected, just like what was done in step 2.

x	prob
(7,4)	-
(8,3)	2/55
(5,9)	29/55
(3,3)	4/55
(1,3)	-
(10,1)	18/55



After we calculated the probability, we random select the last centroid from the remaining 4 points, assuming we chose point (5,9) for its highest probability. At this point, all three starting centroid were chosen.

We see in each step, in stead of chosen centroid totally randomly by K-means, the K-means++ method carefully calculates the distance between selected centroid and the rest points, trying to select each new centroids far spread out while still keeping some randomness.



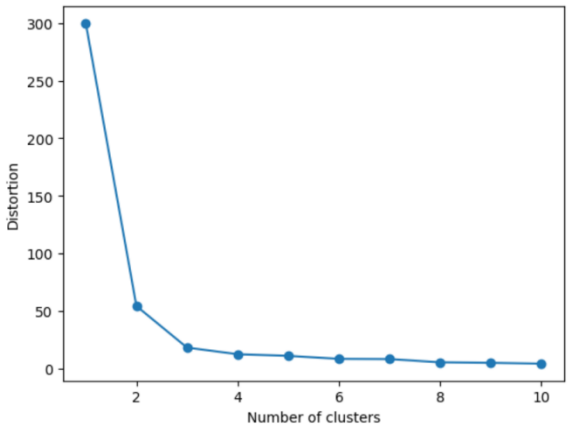
(b) What are the limitations in K-means++?

1. Both K-means and K-means++ depend on the K to be specified in advance. So it can be challenging to find the optimal K value.
2. The core concept of K-means++ rely on calculating distance between points/centroids, so outlier can distort the cluster result.
3. It is computational expensive. K-means++ needs multiple iterations to converge, and the complexity increases linearly with the number of data points and clusters.

(c) Implement the Elbow method using Python considering ”petal width” and ”petal length” features. Find the optimum number of clusters.

```
from sklearn.datasets import load_iris
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler

iris = load_iris()
iris_data = pd.DataFrame(data=iris.data, columns=iris.feature_names)

# We only use the "petal width" & "petal length" columns
iris_data_wl = iris_data.iloc[:,[2,3]]
# Scale the length and width
scaler = StandardScaler()
x_scaled = scaler.fit_transform(iris_data_wl)

distortions = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i,n_init='auto', random_state=0)
    kmeans.fit(x_scaled)
    distortions.append(kmeans.inertia_)

# Plot the Elbow curve
plt.plot(range(1, 11), distortions, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Distortion')
plt.show()
```

The elbow method in evaluating cluster result is to find the value of k at which the distortion starts to level off and form an elbow shape. Here the “distortion” is the squared distances between each sample in the dataset and its assigned centroid.

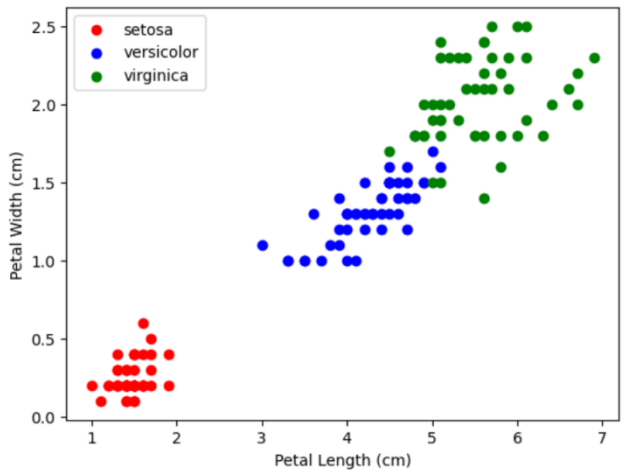


From plot above we can see that when the “number of clusters” rose from 1 cluster to 2 clusters, the distortion dropped from 300 to around 55, and when it rose from 2 clusters to 3 clusters, the distortion dropped from 55 to 20, still a lot. But from 3 clusters to 4 clusters, the difference is roughly 2 or 3, the

distortion drop is significantly leveled off at cluster number of 3. So 3 is the cluster potion we observed from the plot.

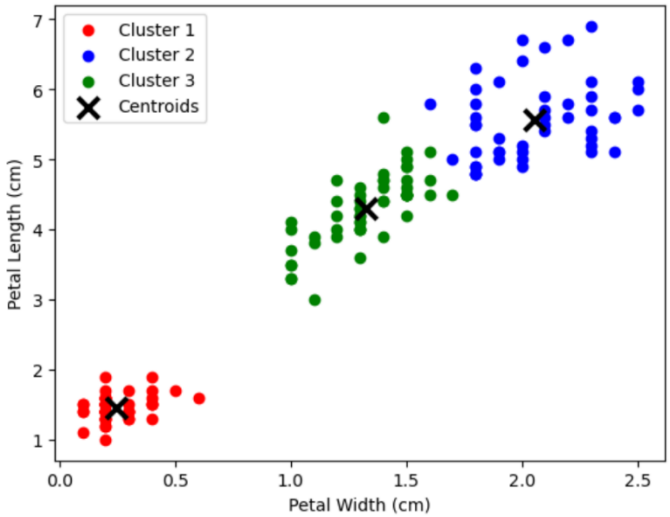
(d) Visualize the actual labels with respect to these two features.

```
import pandas as pd
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

# Our previous iris_data did not contain class, so let's add it
iris_data['class'] = iris.target

# Set up the plot
fig, ax = plt.subplots()

# Plot the data points for each class with a different color
for label, color in zip(range(3), ['red', 'blue', 'green']):
    mask = (iris_data['class'] == label)
    ax.scatter(iris_data.loc[mask, 'petal length (cm)'], iris_data.loc[mask, 'petal width (cm)'],
               c=color, label=iris.target_names[label])

# Add axis labels and Legend
ax.set_xlabel('Petal Length (cm)')
ax.set_ylabel('Petal Width (cm)')
ax.legend()

# Show the plot
plt.show()
```



(e) Write Python code to implement K-means++ clustering using the same features as above. Visualize the resulting clusters in a scatter plot. Mark the centroids.

```
kmeans = KMeans(n_clusters=3,n_init=20, max_iter=300, init='k-means++')
# Notice here the kmeans is fitted using the scaled petal width and length
kmeans.fit(x_scaled)
labels = kmeans.labels_
centroids = kmeans.cluster_centers_

# Since the kmeans is fitted on the scalled value, but we normally want to show how the cluster is formed
# under the regular scale of the petal width and length, so the centroids of the plot need to be inverse scaled to the
# original scale/value
origin_centroids = scaler.inverse_transform(centroids)

# Visualize the resulting clusters with centroids
fig, ax = plt.subplots()

for label, color in zip(range(3), ['red', 'blue', 'green']):
    mask = (labels == label)
    ax.scatter(iris_data_w1.loc[mask, 'petal width (cm)'], iris_data_w1.loc[mask, 'petal length (cm)'],
               c=color, label=f'Cluster {label+1}')

ax.scatter(origin_centroids[:, 0], origin_centroids[:, 1], s=150, marker='x', linewidths=3, color='black', label='Centroids')

ax.set_xlabel('Petal Width (cm)')
ax.set_ylabel('Petal Length (cm)')
ax.legend()

plt.show()
```


Cluster 1
Cluster 2
Cluster 3
Centroids
Petal Length (cm)
Petal Width (cm)
1
2
3
4
5
6
7
0.0
0.5
1.0
1.5
2.0
2.5

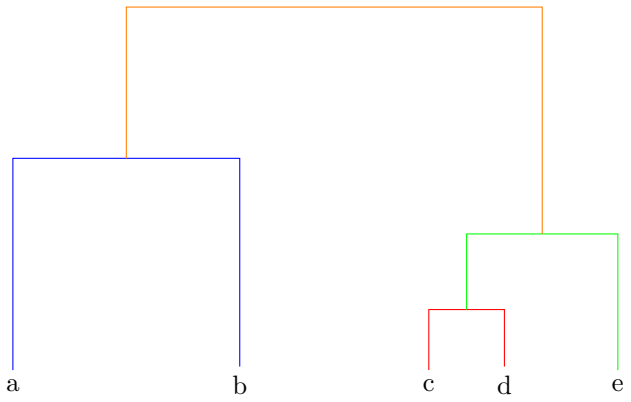
COMPSCI762 TUTORIAL WEEK 10

ANDREW WOOD

1. Question 3

1.1. **Part** (a)**.** There are two types of hierarchical clustering. The first is called *agglomerative nesting*, abbreviated *AGNES*, which is a bottom-up approach (we explain how it works in part (b)). The second is called *divisive clustering*, or *DIANA*, which is a top-down approach.

1.2. **Part** (b)**.** We explain how AGNES preforms hierarchical clustering. As previously mentioned, AGNES is a bottom-up approach to create our tree of clusters (which we can represent the output as a dendogram as shown below).



In the example above, using AGNES we would start with leaves a, b, c, d, e, and treat them each as individual clusters. We then use a distance metric (which we elaborate in part (c)) to combine clusters which have minimal distance. We repeat this recursively until we are left with a single cluster. So, in our example above, we would find that c and d have minimal distance, and so merge these two first. Thereafter, we find the distance between newly formed cluster c, d and e to be minimal and so merge them. Next, a and b are merged, and finally merge c, d, e and a, b.

1.3. **Part** (c)**.** Four different ways AGNES measures distance are *single linkage*, *complete linkage*, *centroid linkage*, and what we will call *average linkage* (also referred to as *group-average agglomerative clustering* or *GAAC*). To be clear, these are methods on how AGNES measures distance between *clusters*, not necessarily how AGNES measures distance between *instances* within clusters. To measure distance between points within clusters, we can choose any metric d (usually one takes euclidean distance).

Now, single linkage measures distance between two clusters X and Y by taking the minimal distance between $x \in X$ and $y \in Y$ with respect to our metric d. That is, $\min_{x \in X, y \in Y} d(x, y)$. Singe linkage can create issues, however, since there may be $x \in X$ which is an outlier of our cluster, yet is of minimal distance to some point $y \in Y$, even though X and Y should not otherwise be merged.

Complete linkage instead maximises the distance between points of two clusters (i.e., $\max_{x \in X, y \in Y} d(x, y)$). This means that the resulting clusters will have their points closer together, and hence have small diameter.

Centroid linkage measures distance by computing the average of the two clusters, then computing their distance; i.e., $\frac{1}{|X|} \sum_{x \in X} x - \frac{1}{|Y|} \sum_{y \in Y} y$. The resulting clusters are spherical. Centroid linkage may merge two clusters which have shorter distance than a previous cluster distance, which is called inversion.

Average linkage measures distance by taking the average over the distances between the instances (i.e., $\frac{1}{|X||Y|} \sum_{x \in X, y \in Y} d(x, y)$). The benefit of average linkage over single and complete linkage, is that it uses all instances rather than just using a single one (so does not have issues with noise/outliers). While centroid linkage also uses all the instances, it does not use the distance between instances (only distance between centroid/average of clusters).