COLLEGE OF BASIC AND APPLIED SCIENCES
SCHOOL OF PHYSICAL AND MATHEMATICAL SCIENCES
BSc COMPUTER SCIENCE

DCIT 414

Report, Coursework II***: "K-means clustering and Hierarchical Clustering."***

BY

IBRAHIM ANYARS SAFIANU – **10826754**

Table of Contents

INTRODUCTION

The aim of this experiment is to demonstrate knowledge of clustering, an unsupervised learning machine learning approach, using both K-means clustering and Hierarchical clustering. Clustering helps us group datapoints that are similar, without the help of a labelled class. The dataset used is the wine dataset acquired from the University of California Machine Learning repository.

Clustering can be used in data exploration to find patterns in data, segment dataset into different groups, detect outliers in a machine learning model, or extract or select preferred features for machine learning model. These can make it possible to group customers based on their purchase pattern, demography and behavior, which will help business tailor their marketing strategies in an attempt to increase customer satisfaction, recommend personalized products, etc., Clustering has application in image processing, datamining, social media analysis, genetic analysis and traffic analysis.

This project will be carried out using Python popular machine learning library, Scikit-Learn. Since the wine dataset is originally a classification dataset with target attribute and a class label, we will exclude target attribute. The input attributes will be normalized. The choice of the normalization technique was between MinMaxScaler() and StandardScalar() functions. With MinMaxScalar, the relative relationship between data points will be normalized within 0 and 1 which is suitable for k-means clustering for multivariate dataset, i.e., multiple features, that are not on the same scale, Also, the choice of StandardScalar() is to ensure normal distribution of data. StandardScalar centers data around 0 with a standard deviation of 1. Which is good for improving the performance of clustering algorithms that assume normality.

Agglomerative clustering will be used in the second part of the experiment. Agglomerative Clustering is a hierarchical clustering algorithm that groups similar data points together. It starts with each data point as its own cluster and progressively merges clusters based on their similarity. The algorithm computes a similarity or distance matrix and utilizes linkage criteria to decide which clusters to merge. This process continues until the desired number of clusters is achieved or until only one cluster remains. Agglomerative Clustering produces a dendrogram, which is a tree-like structure representing the merging process. It offers flexibility in selecting the number of clusters by cutting the dendrogram at different heights. This algorithm is commonly used for exploratory analysis and visual interpretation of data to identify hierarchical relationships and form meaningful clusters.

WINE DATASET

The wine dataset is a popular dataset with measurements of chemical properties of variants of wine. It contains 178 data points or records representing a wine sample. The attributes that make up the sample are alcohol, malic acid, ash, alkalinity of ash, magnesium, total phenols, flavanoids, nonflavanoid phenols, proanthocyanins, color intensity, hue, OD280/OD315 of diluted wines.

importing wine data

```
df = pd.read_csv('wine.data')
```

```
df.head()
```

	1	14.23	1.71	2.43	15.6	127	2.8	3.06	.28	2.29	5.64	1.04	3.92	1065
0	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
1	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
2	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
3	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735
4	1	14.20	1.76	2.45	15.2	112	3.27	3.39	0.34	1.97	6.75	1.05	2.85	1450

Numerical exploratory

```
: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 177 entries, 0 to 176
Data columns (total 14 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   1       177 non-null    int64
 1   14.23   177 non-null    float64
 2   1.71    177 non-null    float64
 3   2.43    177 non-null    float64
 4   15.6    177 non-null    float64
 5   127     177 non-null    int64
 6   2.8     177 non-null    float64
 7   3.06    177 non-null    float64
 8   .28     177 non-null    float64
 9   2.29    177 non-null    float64
 10  5.64    177 non-null    float64
 11  1.04    177 non-null    float64
 12  3.92    177 non-null    float64
 13  1065    177 non-null    int64
dtypes: float64(11), int64(3)
memory usage: 19.5 KB
```

PREPROCESSING

After loading the wine data, the header was turn off because the wine.data dataset does not have a header, which means the function should consider the first row as part of the dataset rather than a column header.

Then the class label was removed because, once again, in clustering we do not need it. Then the data was normalized using both MinMaxScaler() and StandardScaler(), which both affected the algorithm by giving different results. But the StandardScaler, had best performance effect.

```
: # Function to load the Wine dataset
def load_wine():
    # Load the Wine dataset from UCI Machine Learning Repository
    wine_data = pd.read_csv('wine.data', header=None)

    # Remove class label and target column
    X = wine_data.iloc[:, 1:]

    # Normalize the input attributes
#     scaler = MinMaxScaler()
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    return X_scaled
```

K- MEANS CLUSTERING

a) Applying K-means clustering to the dataset using with K values 2, 3, 4, 5, 6, 7, 8, 9 10

Default parameters:

random_state = 42

n_init=10

n_clusters = 2, 3, 4, 5, 6, 7, 8, 9, 10

```python
# # Task 1: K-means clustering with different K values
def kmeans_clustering(X_scaled):
    k_values = [2, 3, 4, 5, 6, 7, 8, 9, 10]
    kmeans_results = []

    for k in k_values:
        kmeans = KMeans(n_clusters=k, n_init=10, random_state=42)
        kmeans.fit(X_scaled)
        kmeans_results.append(kmeans)

        labels = kmeans.labels_

        # Calculate WSS and BSS
        wss = kmeans.inertia_
        bss = kmeans.score(X_scaled) * X_scaled.shape[0]

        # Calculate Silhouette coefficient
        silhouette_avg = silhouette_score(X_scaled, labels)

        # Calculate Davies-Bouldin index
        db_index = davies_bouldin_score(X_scaled, labels)

        # Plot the cluster assignments
        plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=labels, cmap='rainbow')
        plt.title(f'K-means clustering with K={k}')
        plt.show()

        # Print the evaluation metrics
        print(f'K={k}: WSS={wss:.2f}, BSS={bss:.2f}, Silhouette={silhouette_avg:.2f}, DB Index={db_index:.2f}')


    return kmeans_results
```

k-means algorithm

Rational for choice of parameters:

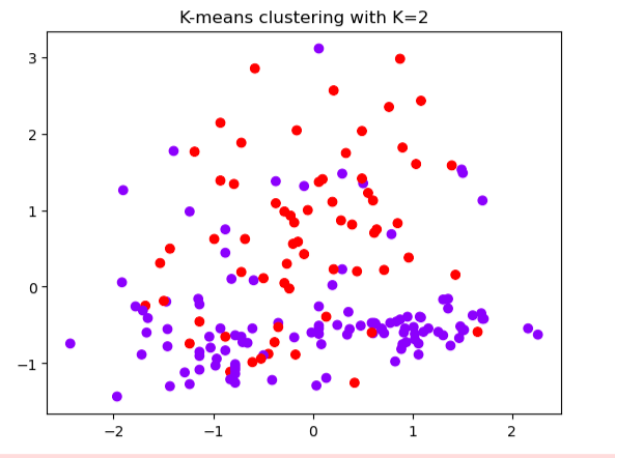
- n_clusters: This parameter determines the number of clusters to create. In the provided code, the n_clusters parameter takes values from the k_values list, which specifies different numbers of clusters to evaluate. Testing multiple values of n_clusters, is meant to assess the quality of the clustering results for different cluster numbers.
- n_init: The K-means algorithm uses random initialization of cluster centroids. The n_init parameter specifies the number of times the algorithm will be run with different initializations. The final results will be based on the run that produces the lowest within-cluster sum of squares (WCSS). In the provided code, n_init is set to

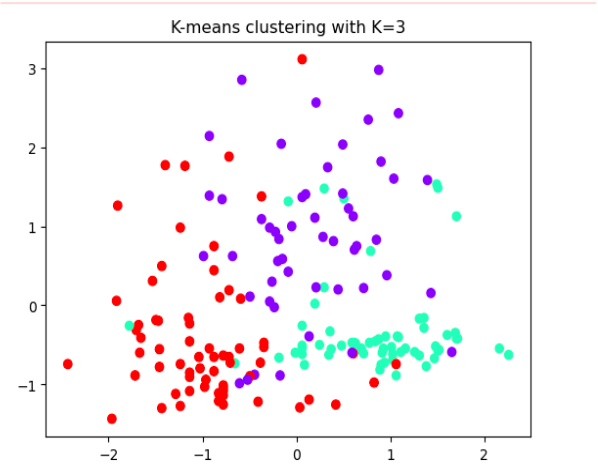
10, meaning the algorithm will be run 10 times with different initializations, and the best result will be selected.

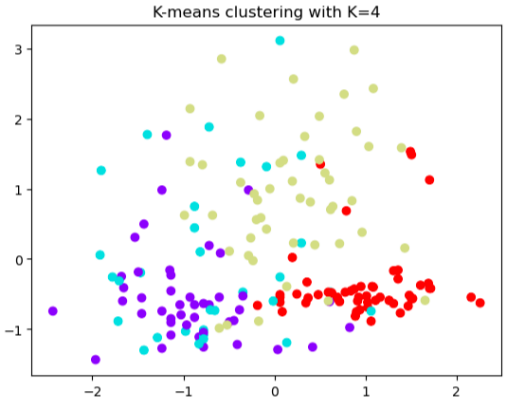
- random_state: This parameter controls the random number generator used for the initialization of centroids. By setting random_state to a fixed value (in this case, 42), the initialization will be reproducible, ensuring consistent results when running the code multiple times.
- By using a range of n_clusters values, performing multiple initializations (n_init), and fixing the random seed (random_state), the code allows for a systematic evaluation of K-means clustering performance with different numbers of clusters,

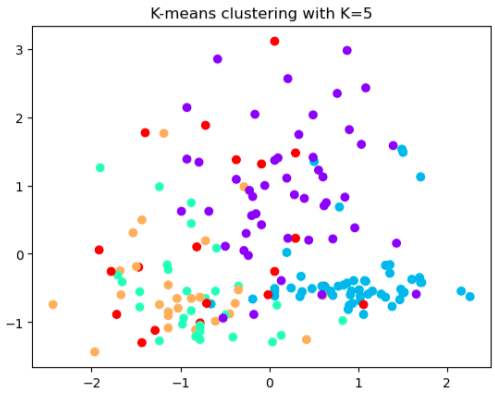
Comparing resulting performance of all 9 outcomes: k=2, 3, 4, 5, 6, 7, 8, 9 and 10

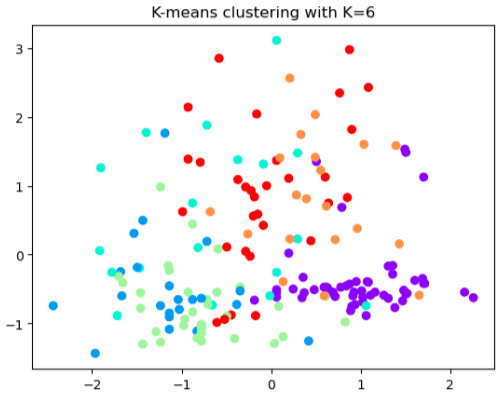
a) The scatter plot:

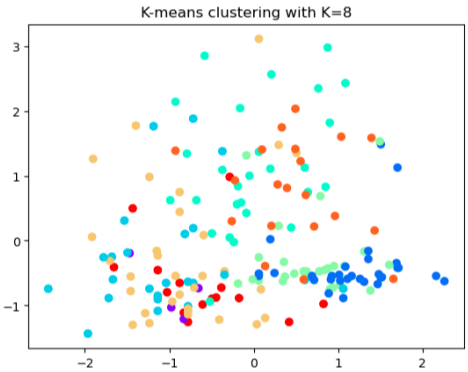


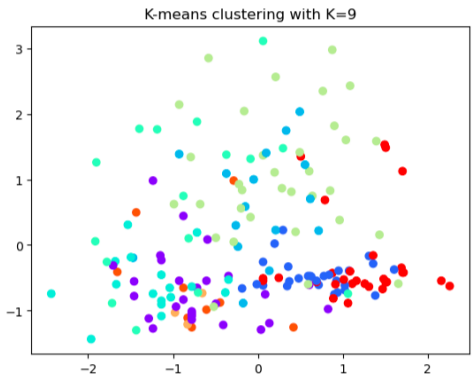


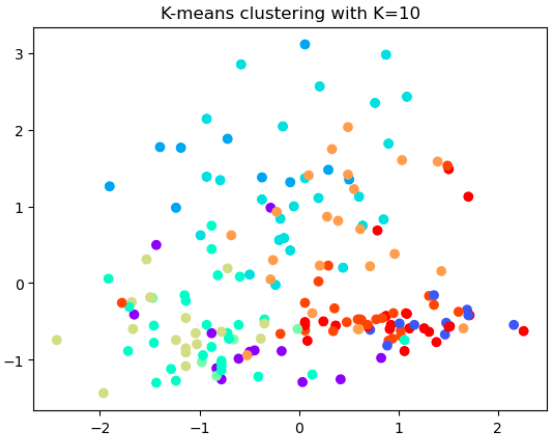

K-means clustering with K=4


K-means clustering with K=5


K-means clustering with K=6


K-means clustering with K=8


K-means clustering with K=9


K-means clustering with K=10

Evaluating the quality of the k-means clustering:

The calculation of accuracy for a clustering model differs from that of supervised learning models due to the nature of clustering being an unsupervised learning task. Unlike supervised learning, there are no known labels or ground truth to compare the clustering predictions against.

Nevertheless, several evaluation metrics exist to evaluate the effectiveness of clustering, although they do not directly measure accuracy. Here are a few widely used evaluation metrics for clustering:

1. Within-cluster sum of squares (WCSS): this measures the sum of the squared distances within each cluster point and its centroid. A lower value shows similarity, hence desirable. It also measures compactness.
2. Between-cluster sum of squares (BCSS): this parameter measures the sum of squared distances between cluster centroids and the mean of the dataset. It focuses on variation or dissimilarity between clusters. The more the BCSS value the higher the level of variation.
3. Silhouette coefficients: is the measure of closeness of each sample to its assigned cluster compared to neighboring clusters. It lies between -1 and 1. Higher values mean more similarity with other data points in the same cluster and different from neighboring clusters.
4. Davies-Bouldin index evaluates the quality of clustering by considering both the within-cluster dispersion and the between-cluster separation. It measures the average similarity between each cluster and its most similar cluster, while also considering the average dissimilarity to other clusters. Lower values indicate better-defined and well-separated clusters.

Scores by clusters

K-VALUE	WCSS	BCSS	SILHOUETTE	DB INDEX
2	1659.01	295303.42	0.27	1.45
3	1277.93	227471.27	0.28	1.39
4	1175.71	209275.52	0.25	1.82
5	1104.86	196665.38	0.23	1.69
6	1042.39	185544.92	0.20	1.83
7	988.05	175873.49	0.21	1.60
8	940.71	167446.05	0.14	1.83
9	902.08	160569.94	0.15	1.75
10	866.80	154290.25	0.13	1.75

Hierarchical Clustering

Accuracy of different k values compared to the actual number of clusters.

K-VALUE	ACCURACY
2	0.0
3	0.6966292134831461
4	0.033707865168539325
5	0.46629213483146065
6	0.07865168539325842
7	0.016853932584269662
8	0.4550561797752809
9	0.15730337078651685
10	0.2303370786516854

Observation:

After testing both normalization techniques, StandardScalar gave the best performance and true performance, hence, the preferred normalization technique. Just as expected, the k-mean with value 3 had the highest accuracy which is the actual number of clusters in dataset.

Applying hierarchical clustering and comparing the generated hierarchical structures.

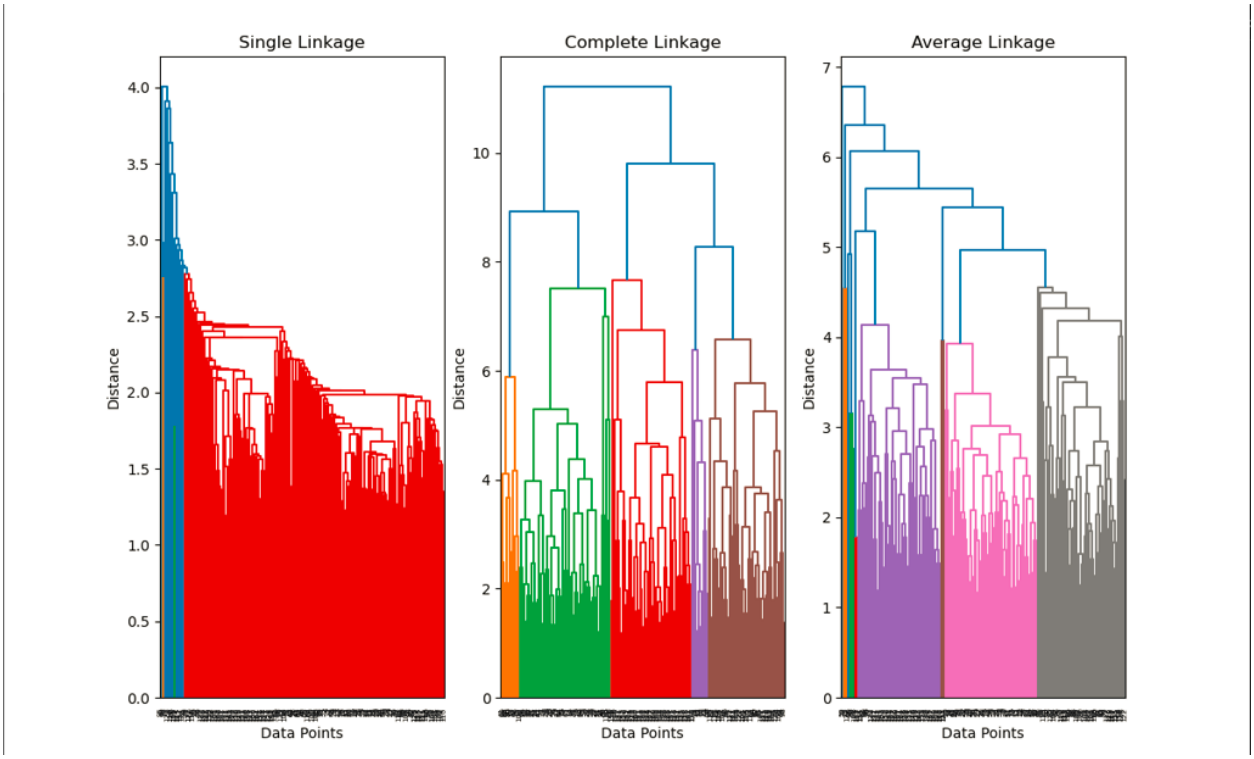
AgglomerativeClustering with n_clusters = 3 was used to construct a hierarchical clustering. The linkage methods single, complete and average was passed to the linkage. While single merges the shortest distance between two clusters's closet points, complete linkage merges the largest distance between furthest points of any two clusters, and finally average merges the average distance between all of the points of two clusters. The model is then trained on the X_scaled dataset. A linkage matrix was created between the independent variables and the methods single, complete and average to create a link.

In [83]:
```
#Task 3: Hierarchical clustering with different linkage methods
def hierarchical_clustering(X_scaled):
    linkage_methods = ['single', 'complete', 'average']
    hierarchical_results = []

    for method in linkage_methods:
        hierarchical = AgglomerativeClustering(n_clusters=3, linkage=method)
        hierarchical.fit(X_scaled)
        linkage_matrix = linkage(X_scaled, method=method)
        hierarchical_results.append(linkage_matrix)

    return hierarchical_results
```

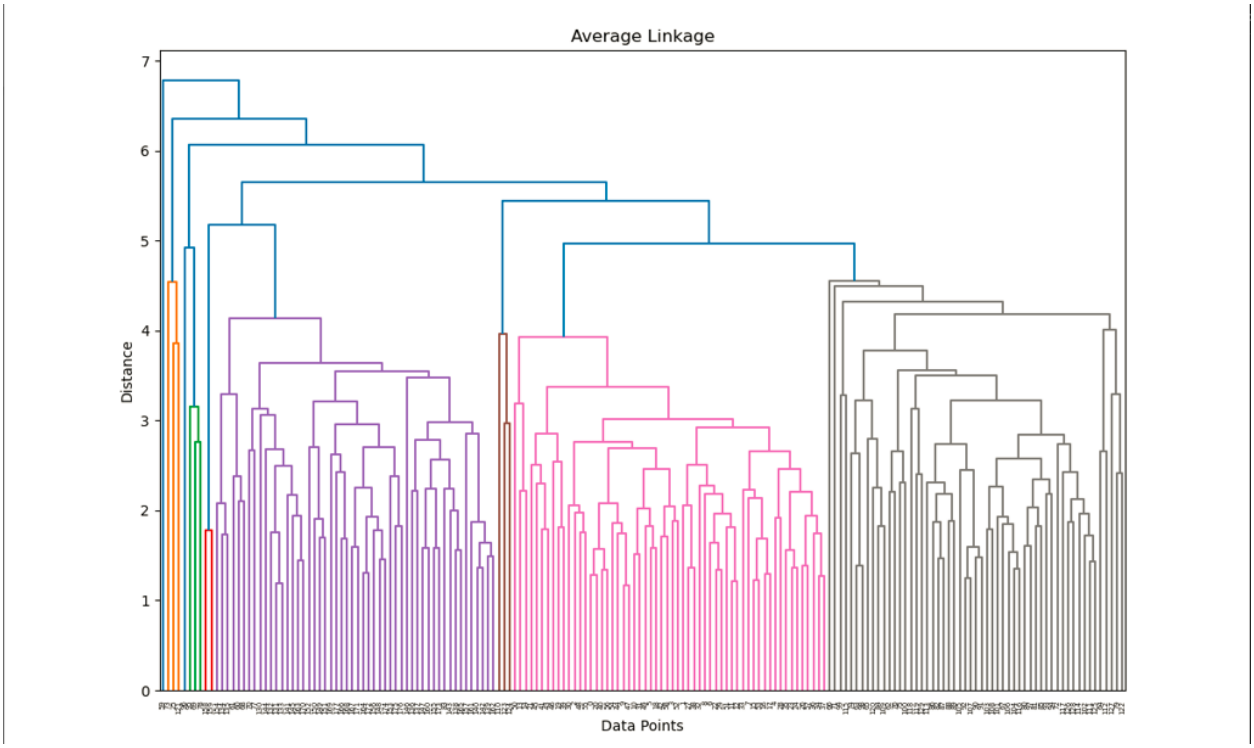
Diagram of the linkages:

For each of the linkage method a plot was constructed. Below is the code.

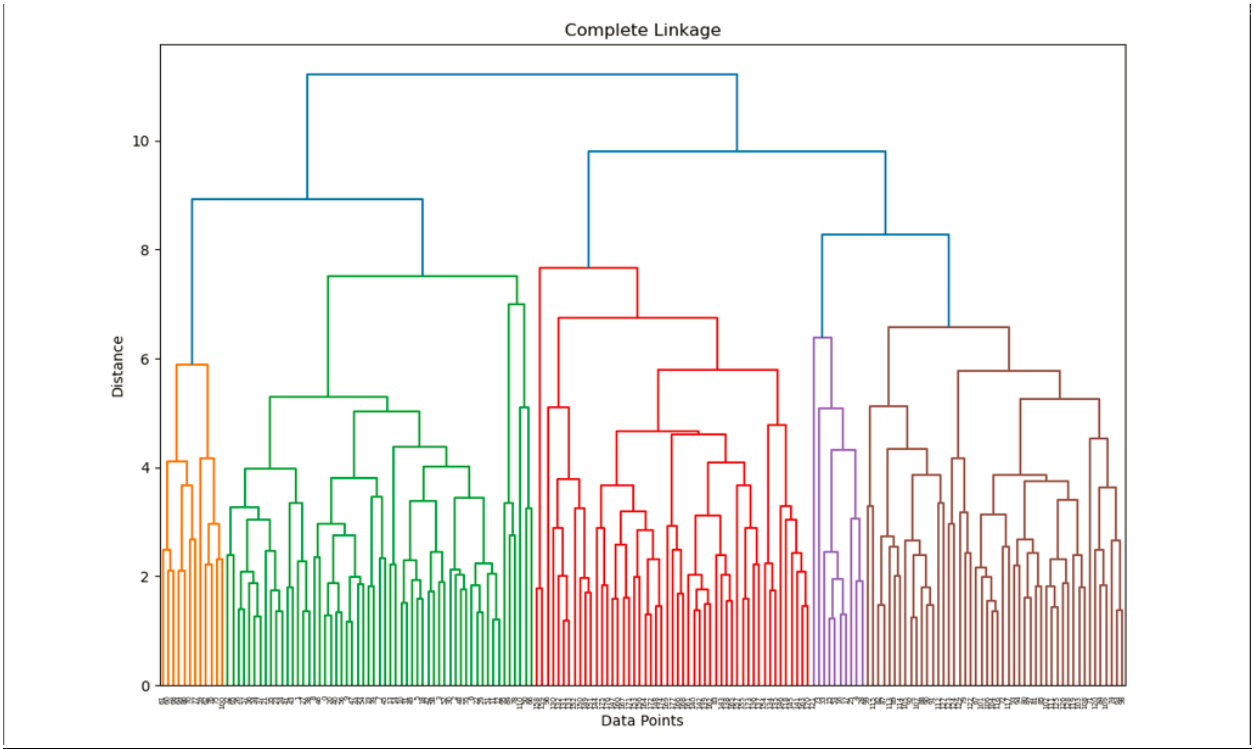
```
:
# Visualization of hierarchical clustering results
def visualize_hierarchical_clusters(hierarchical_results):
    for i, method in enumerate(['single', 'complete', 'average']):
        plt.figure(figsize=(12, 8))
        plt.title(f'{method.capitalize()} Linkage')
        plt.xlabel('Data Points')
        plt.ylabel('Distance')
        dendrogram(hierarchical_results[i])
        plt.savefig(f'hierarchical_clustering_{method}.png')

    plt.tight_layout()
    plt.show()

```



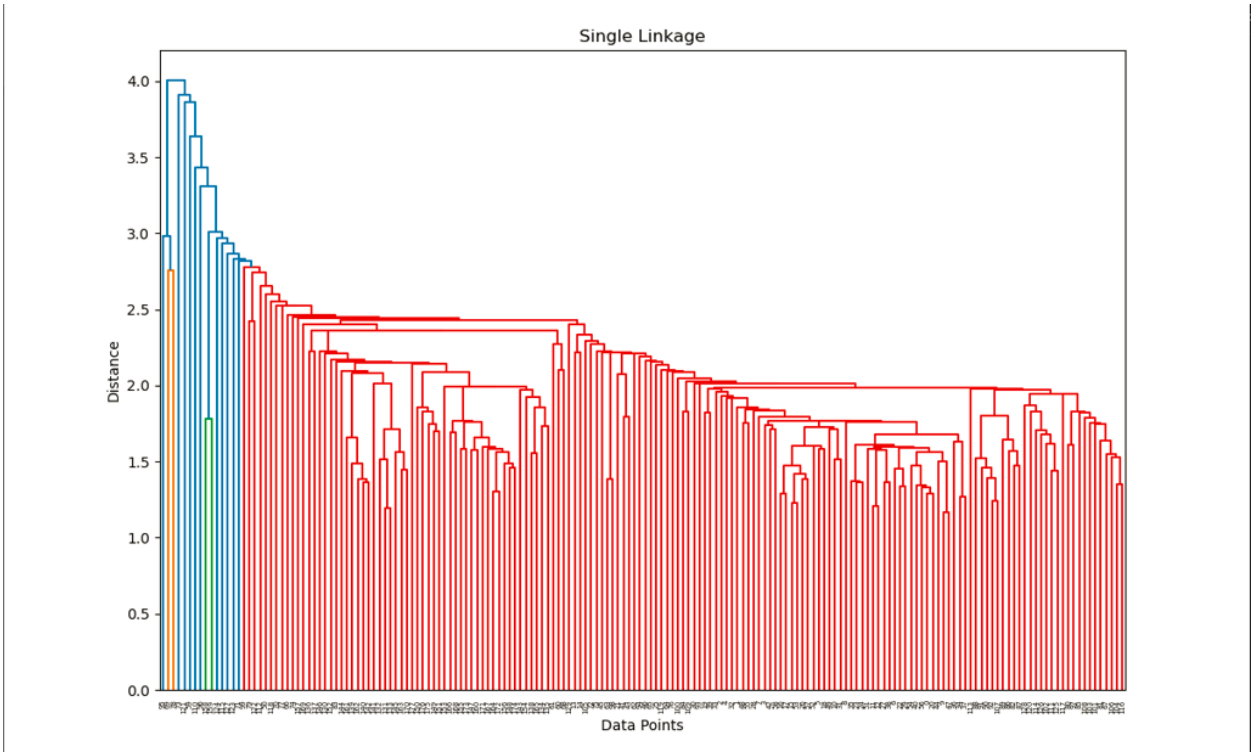
Al of the linkages.



Average linkage in detail



Complete linkage in detail.



Single linkage in detail.