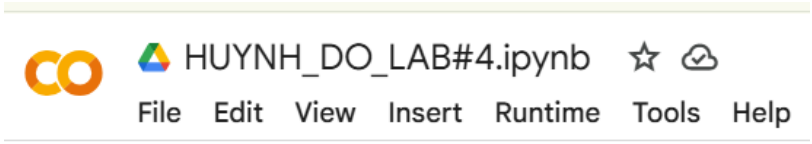
Huynh Do Lab#4:

Objective:

This lab assignment exposes student to using Dendrograms diagrams to support Hierarchical Clustering. This process is trying to find groups, where you start with one big group, and you split it apart, split it apart, split it apart until you have individual cases



1. Import libraries

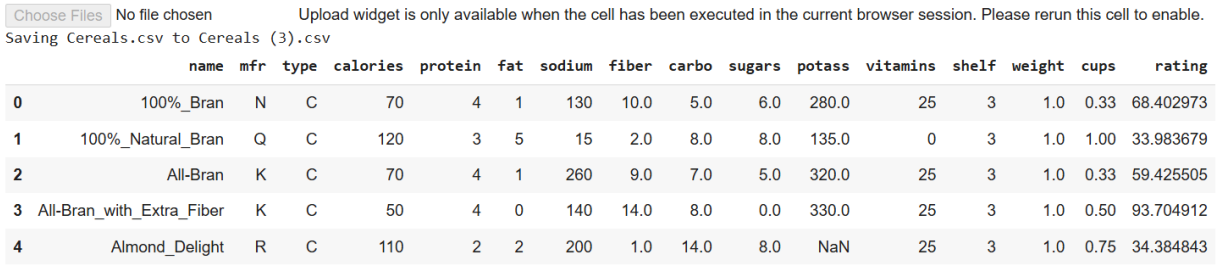
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
from mpl_toolkits.mplot3d import Axes3D
from google.colab import files
```

The code above imports several Python libraries commonly used in data analysis.

1. import pandas as pd
   - For working with tables/dataframes (Excel, but Pythonized).
2. import numpy as np
   - For number crunching, arrays, and mathematical operations.
3. import matplotlib.pyplot as plt
   - For basic plotting (2D graphs, scatterplots, histograms, etc.).
4. import seaborn as sns
   - For prettier and more complex statistical plots (heatmaps, violin plots, etc.).
5. from sklearn.cluster import KMeans
   - For K-Means clustering (unsupervised learning to group similar data).
6. from sklearn.preprocessing import StandardScaler
   - For scaling/normalizing features so that different units don't mess up models.
7. from sklearn.decomposition import PCA
   - For reducing the number of features (dimensionality reduction).
8. from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
   - dendrogram: draws a tree diagram of clusters.
   - linkage: defines how to compute distance between clusters.
   - fcluster: slices the dendrogram into flat clusters.

9. from mpl_toolkits.mplot3d import Axes3D
    - For making 3D plots (scatterplots in 3D space).
10. from google.colab import files
    - For uploading and downloading files inside Google Colab.

2. Upload file Cereals.csv



Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving Cereals.csv to Cereals (3).csv
```

	name	mfr	type	calories	protein	fat	sodium	fiber	carbo	sugars	potass	vitamins	shelf	weight	cups	rating
0	100%_Bran	N	C	70	4	1	130	10.0	5.0	6.0	280.0	25	3	1.0	0.33	68.402973
1	100%_Natural_Bran	Q	C	120	3	5	15	2.0	8.0	8.0	135.0	0	3	1.0	1.00	33.983679
2	All-Bran	K	C	70	4	1	260	9.0	7.0	5.0	320.0	25	3	1.0	0.33	59.425505
3	All-Bran_with_Extra_Fiber	K	C	50	4	0	140	14.0	8.0	0.0	330.0	25	3	1.0	0.50	93.704912
4	Almond_Delight	R	C	110	2	2	200	1.0	14.0	8.0	NaN	25	3	1.0	0.75	34.384843

The above screen shot is used to upload and load a CSV file named **Cereals.csv** into a Pandas Data Frame in a Google Colab environment.

When upload is done:

3. Dendrograms diagrams to support Hierarchical Clustering

- Remove All Cereals with Missing Values

```
# ------------------------------------------------------------
# Remove All Cereals with Missing Values
# ------------------------------------------------------------
df_clean = df.dropna().copy()
print("\nData After Dropping Missing Values:")
print(df_clean.head())
print(f"\nRows after cleaning: {len(df_clean)} rows remaining")
```

- df.dropna()
  Creates a new DataFrame by removing any rows in df that contain missing values (NaN).
- .copy()
  Creates a completely independent copy of the resulting DataFrame to avoid modifying the original df by reference.
- df_clean
  A cleaned version of the original DataFrame, free of missing values and safe to modify independently.

```
Data After Dropping Missing Values:
                        name mfr type  calories  protein  fat  sodium  fiber  \
0                  100%_Bran   N    C        70        4    1     130   10.0
1          100%_Natural_Bran   Q    C       120        3    5      15    2.0
2                    All-Bran   K    C        70        4    1     260    9.0
3  All-Bran_with_Extra_Fiber   K    C        50        4    0     140   14.0
5    Apple_Cinnamon_Cheerios   G    C       110        2    2     180    1.5

   carbo  sugars  potass  vitamins  shelf  weight  cups     rating
0    5.0     6.0   280.0        25      3     1.0  0.33  68.402973
1    8.0     8.0   135.0         0      3     1.0  1.00  33.983679
2    7.0     5.0   320.0        25      3     1.0  0.33  59.425505
3    8.0     0.0   330.0        25      3     1.0  0.50  93.704912
5   10.5    10.0    70.0        25      1     1.0  0.75  29.509541

Rows after cleaning: 74 rows remaining
```

- Select Only Numeric Features and Standardize

```
# ------------------------------------------------------------
# Select Only Numeric Features and Standardize
# ------------------------------------------------------------
df_numeric = df_clean.select_dtypes(include=[np.number])
scaler = StandardScaler()
df_scaled = scaler.fit_transform(df_numeric)
```

- select_dtypes(include=[np.number]) → Filters only numeric columns.
- StandardScaler() → Initializes a scaler object for normalization.
- fit_transform() → Standardizes all numeric features based on learned mean and variance.
- df_scaled → A scaled NumPy array where all features are on the same scale.

- Run K-Means Clustering

```
# ------------------------------------------------------------
# Run K-Means Clustering
# ------------------------------------------------------------
kmeans = KMeans(n_clusters=4, random_state=42)
kmeans.fit(df_scaled)
labels_kmeans = kmeans.labels_

# Attach KMeans Cluster labels
df_clean['KMeans_Cluster'] = labels_kmeans

print("\n KMeans Cluster Assignments:")
print(df_clean[['KMeans_Cluster']].head())

print("\n KMeans Cluster Centroids:")
kmeans_centroids = pd.DataFrame(kmeans.cluster_centers_, columns=df_numeric.columns)
print(kmeans_centroids)
```

1. **kmeans = KMeans(n_clusters=4, random_state=42)**
   - Initializes a KMeans clustering model set to form **4 clusters**.
   - random_state=42 ensures reproducible clustering results.
2. **kmeans.fit(df_scaled)**
   - Fits the KMeans model to the **scaled numeric data**.
   - Assigns each data point to one of the 4 clusters.
3. **labels_kmeans = kmeans.labels_**
   - Extracts the cluster labels (an array indicating which cluster each point belongs to).
4. **df_clean['KMeans_Cluster'] = labels_kmeans**
   - Adds a new column KMeans_Cluster to the df_clean DataFrame.
   - Each row now carries the cluster assignment.
5. **kmeans_centroids = pd.DataFrame(kmeans.cluster_centers_, columns=df_numeric.columns)**
   - Converts the cluster centers (centroids) into a DataFrame.
   - Columns match the original numeric feature names for easier interpretation.
6. **print(kmeans_centroids)**
   - Displays the numerical centers of each cluster across all features.

```
 KMeans Cluster Assignments:
   KMeans_Cluster
0               2
1               3
2               2
3               2
5               0

 KMeans Cluster Centroids:
    calories   protein       fat    sodium     fiber     carbo    sugars  \
0  0.199162 -0.926249  0.000000  0.075498 -0.666503 -0.595339  1.008898
1 -0.479532  0.171694 -0.515152 -0.032453 -0.148525  0.634387 -0.878861
2 -2.216901  1.391180 -0.333333  0.173970  3.666168 -2.086018 -0.794871
3  1.076051  0.565399  1.058824 -0.060966  0.464670 -0.127917  0.600009

     potass  vitamins     shelf    weight      cups    rating
0 -0.726334 -0.183083 -0.549730 -0.202203  0.229015 -0.983650
1 -0.254374  0.056405 -0.188293 -0.434882  0.286892  0.607885
2  3.004149 -0.183083  0.948401 -0.202203 -1.857851  2.257956
3  0.860877  0.148980  0.877223  1.129647 -0.511953 -0.363377
```

- Apply Hierarchical Clustering

```
# ------------------------------------------------------------
# Apply Hierarchical Clustering
# ------------------------------------------------------------
linkage_single = linkage(df_scaled, method='single')
linkage_complete = linkage(df_scaled, method='complete')
```

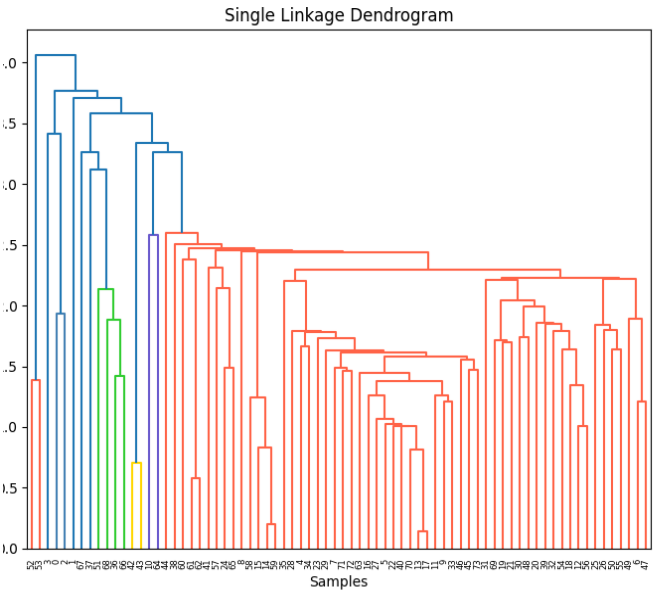
1. **linkage(df_scaled, method='single')**
   - Performs hierarchical clustering on the scaled data.
   - **method='single'** uses **single linkage**, meaning the distance between two clusters is defined as the shortest distance between any two points in the two clusters.
   - The result is stored in linkage_single.
2. **linkage(df_scaled, method='complete')**
   - Also performs hierarchical clustering, but with **complete linkage**.
   - **method='complete'** defines the distance between two clusters as the longest distance between any two points across clusters.
   - The result is stored in linkage_complete.

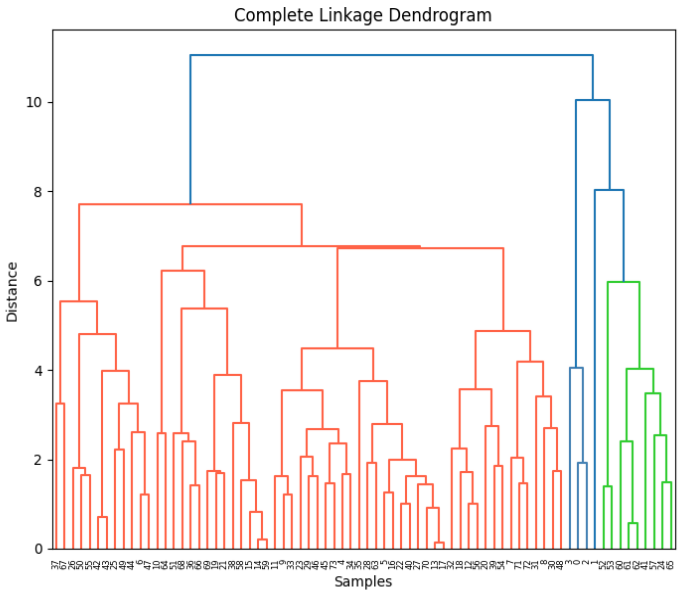
- Compare Dendrograms (Single vs Complete Linkage)

```
# ----------------------------------------------------
# Compare Dendrograms (Single vs Complete Linkage)
# ----------------------------------------------------
plt.figure(figsize=(14, 6))
plt.subplot(1, 2, 1)
dendrogram(linkage_single)
plt.title('Single Linkage Dendrogram')
plt.xlabel('Samples')
plt.ylabel('Distance')

plt.subplot(1, 2, 2)
dendrogram(linkage_complete)
plt.title('Complete Linkage Dendrogram')
plt.xlabel('Samples')
plt.ylabel('Distance')
plt.tight_layout()
plt.show()
```

1. **plt.figure(figsize=(14, 6))**
   - Creates a new figure with a size of 14 units wide and 6 units tall.
2. **plt.subplot(1, 2, 1)**
   - Sets up the first subplot in a 1-row, 2-column layout.
   - Activates the left plot area.
3. **dendrogram(linkage_single)**
   - Draws the dendrogram for the **single linkage** clustering.
   - Displays how clusters merge based on shortest distances.
4. **plt.title('Single Linkage Dendrogram')**
   - Sets the title for the first subplot.
5. **plt.xlabel('Samples') and plt.ylabel('Distance')**
   - Labels the x-axis and y-axis of the first subplot.
6. **plt.subplot(1, 2, 2)**
   - Moves to the second subplot (right side).
7. **dendrogram(linkage_complete)**
   - Draws the dendrogram for the **complete linkage** clustering.
   - Displays how clusters merge based on farthest distances.
8. **plt.title('Complete Linkage Dendrogram')**
   - Sets the title for the second subplot.





What are the meanings of the above graphs?

- In **single linkage**, cereals start linking too early, even when they aren't that similar overall.
  → **This graph implies that** : there are a few individual cereals that are similar to many others, causing a "chain effect."
- In **complete linkage**, cereals only merge when the whole groups are close together, not just one or two random pairs.
  → **Tells us**: there are actual **tight, separate groups** among cereals based on nutrition or other features.

Analyze Cluster Centroids Again

```
# ------------------------------------------------------------
# Analyze Cluster Centroids Again
# ------------------------------------------------------------
# (Already printed after KMeans)
# cluster via Hierarchical:
num_clusters = 4
complete_clusters = fcluster(linkage_complete, num_clusters, criterion='maxclust')
df_clean['CompleteLinkage_Cluster'] = complete_clusters

print("\n Hierarchical Clustering (Complete Linkage) Cluster Assignments:")
print(df_clean[['CompleteLinkage_Cluster']].head())
```

1. **num_clusters = 4**
   - Sets the desired number of clusters to 4.

2. **fcluster(linkage_complete, num_clusters, criterion='maxclust')**
   - Forms flat clusters from the linkage_complete hierarchy.
   - The criterion='maxclust' ensures exactly 4 clusters are produced.
   - Stores the cluster labels into complete_clusters.
3. **df_clean['CompleteLinkage_Cluster'] = complete_clusters**
   - Adds a new column CompleteLinkage_Cluster to the df_clean DataFrame.
   - Each row is assigned to one of the 4 complete linkage clusters.
4. **print(df_clean[['CompleteLinkage_Cluster']].head())**
   - Displays the first five rows, showing the assigned CompleteLinkage_Cluster labels.

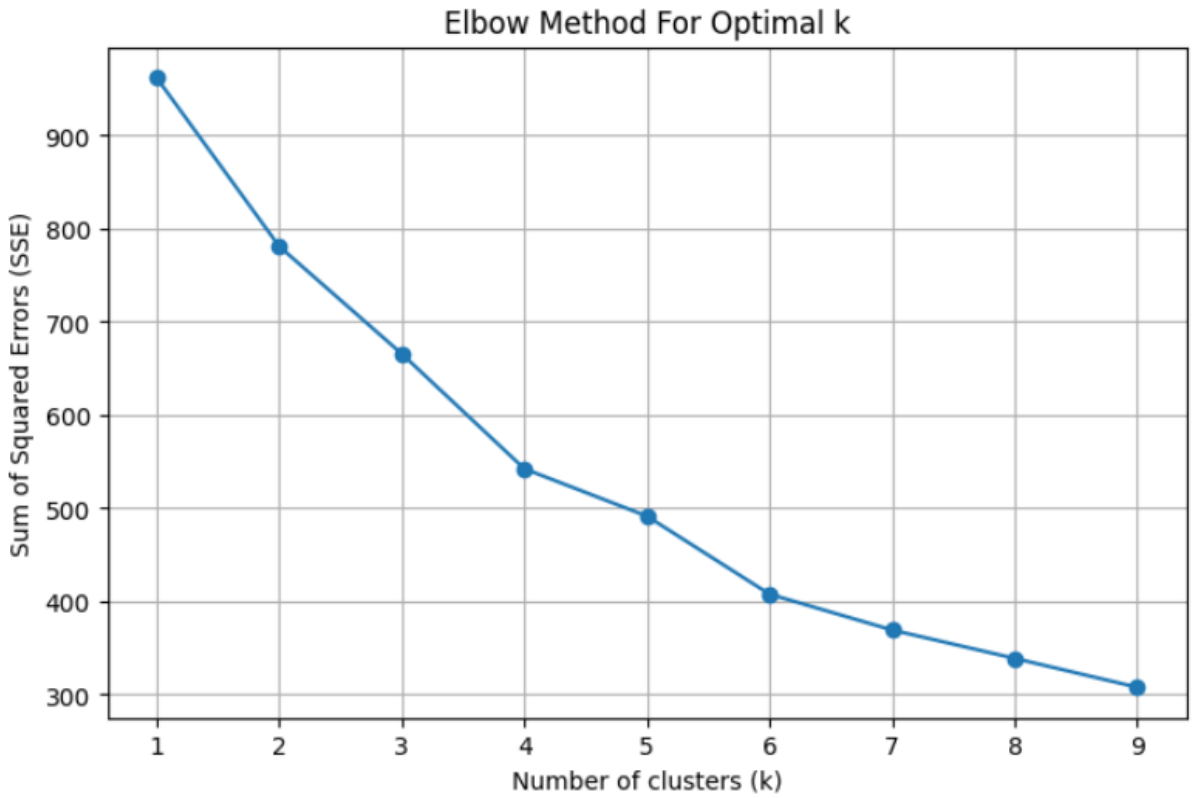
```
 Hierarchical Clustering (Complete Linkage) Cluster Assignments:
   CompleteLinkage_Cluster
0                        2
1                        4
2                        2
3                        2
5                        1
```

- How Many Clusters Would You Use?

```
# ------------------------------------------------
# How Many Clusters Would You Use?
# ------------------------------------------------
# Elbow method for KMeans (for backup visual)
sse = []
for k in range(1, 10):
    km = KMeans(n_clusters=k, random_state=42)
    km.fit(df_scaled)
    sse.append(km.inertia_)

plt.figure(figsize=(8,5))
plt.plot(range(1,10), sse, marker='o')
plt.title('Elbow Method For Optimal k')
plt.xlabel('Number of clusters (k)')
plt.ylabel('Sum of Squared Errors (SSE)')
plt.grid(True)
plt.show()
```

- sse = [] → Stores error values for different k values.
- range(1,10) → Tests k values from 1 to 9.
- km.inertia_ → Captures how compact the clusters are.
- plot(range(1,10), sse) → Draws the Elbow Method graph.
- Elbow point → A visual "bend" indicating a good number of clusters to pick.



Interpretation:

- After a certain point (around `k=3` or `k=4`), the curve **bends** and **flattens**.
  → Adding more clusters **barely improves** the fit anymore.
- The **"elbow" point** is where that bending happens.
  → This elbow suggests the **best number of clusters** to pick.

Conclusion:

Based on Elbow Method + Dendrogram visual split, the best number of clusters to use = 4 clusters.