# Experiment No 8

Aim: Implementation of any one Hierarchical Clustering method

Theory:

A Hierarchical clustering method works via grouping data into a tree of clusters. Hierarchical clustering begins by treating every data point as a separate cluster. Then, it repeatedly executes the subsequent steps:

Identify the 2 clusters which can be closest together, and
Merge the 2 maximum comparable clusters. We need to continue these steps until all the clusters are merged together.
In Hierarchical Clustering, the aim is to produce a hierarchical series of nested clusters. A diagram called a Dendrogram (A Dendrogram is a tree-like diagram that statistics the sequences of merges or splits) graphically represents this hierarchy and is an inverted tree that describes the order in which factors are merged (bottom-up view) or clusters are broken up (top-down view).

Hierarchical clustering is a method of cluster analysis in data mining that creates a hierarchical representation of the clusters in a dataset. The method starts by treating each data point as a separate cluster and then iteratively combines the closest clusters until a stopping criterion is reached. The result of hierarchical clustering is a tree-like structure, called a dendrogram, which illustrates the hierarchical relationships among the clusters.

Hierarchical clustering has a number of advantages over other clustering methods, including:

1. The ability to handle non-convex clusters and clusters of different sizes and densities.
2. The ability to handle missing data and noisy data.
3. The ability to reveal the hierarchical structure of the data, can be useful for understanding the relationships among the clusters. However, it also has some drawbacks, such as:
4. The a need for a criterion to stop the clustering process and determine the final number of clusters.
5. The computational cost and memory requirements of the method can be high, especially for large datasets.
6. The results can be sensitive to the initial conditions, linkage criterion, and distance metric used.

In summary, Hierarchical clustering is a method of data mining that groups similar data points into clusters by creating a hierarchical structure of the clusters.

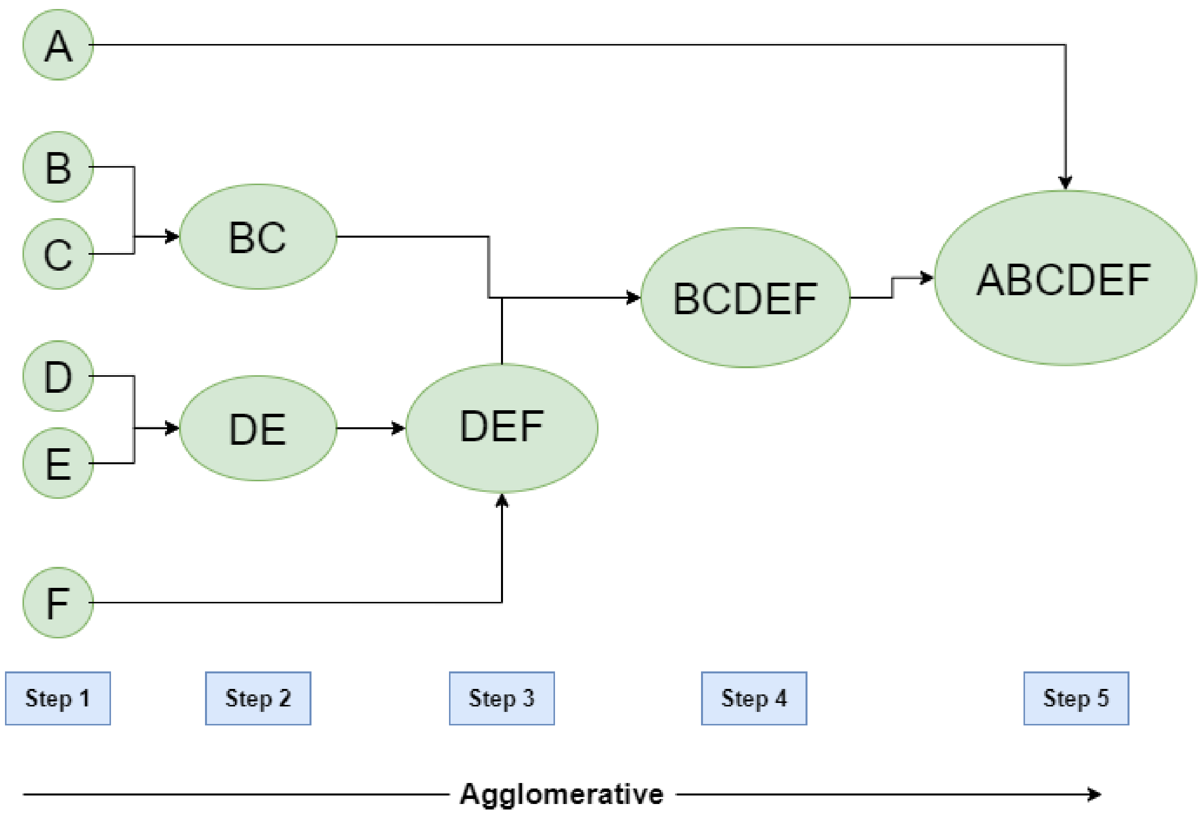
7. This method can handle different types of data and reveal the relationships among the clusters. However, it can have high computational cost and results can be sensitive to some conditions.

**Agglomerative**: Initially consider every data point as an individual Cluster and at every step, merge the nearest pairs of the cluster. (It is a bottom-up method). At first, every dataset is considered an individual entity or cluster. At every iteration, the clusters merge with different clusters until one cluster is formed.

The algorithm for Agglomerative Hierarchical Clustering is:

- Calculate the similarity of one cluster with all the other clusters (calculate proximity matrix)
- Consider every data point as an individual cluster
- Merge the clusters which are highly similar or close to each other.
- Recalculate the proximity matrix for each cluster
- Repeat Steps 3 and 4 until only a single cluster remains.

Let's say we have six data points A, B, C, D, E, and F.



Step 1: Consider each alphabet as a single cluster and calculate the distance of one cluster from all the other clusters.

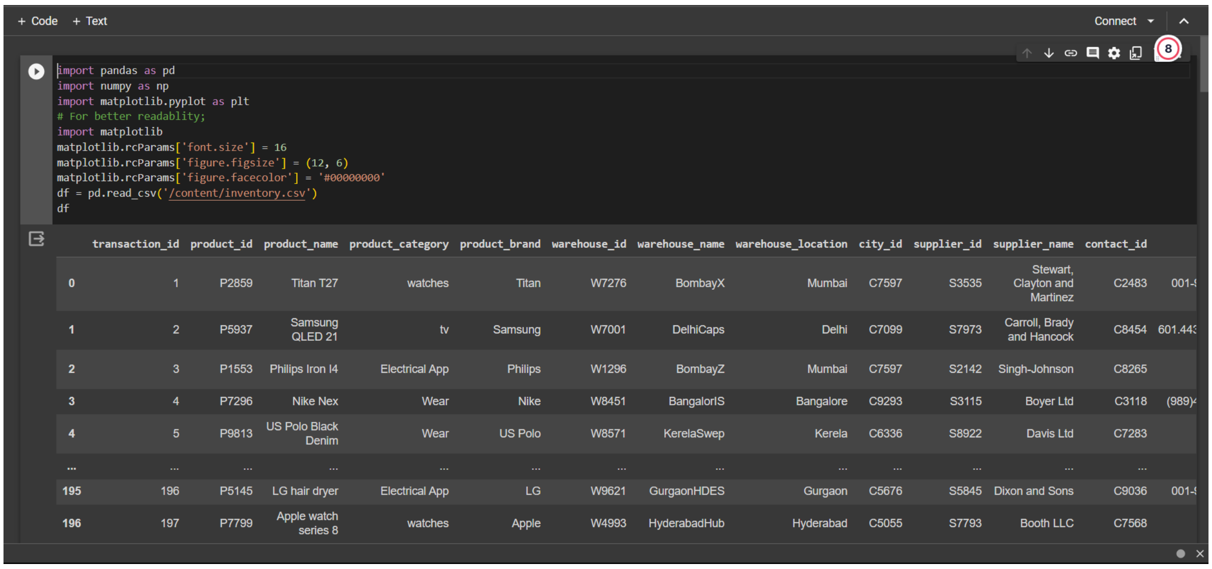
Step 2: In the second step comparable clusters are merged together to form a single cluster. Let's say cluster (B) and cluster (C) are very similar to each other therefore we merge them in the second step similarly to cluster (D) and (E) and at last, we get the clusters [(A), (BC), (DE), (F)]

Step 3: We recalculate the proximity according to the algorithm and merge the two nearest clusters([(DE), (F)]) together to form new clusters as [(A), (BC), (DEF)]

Step 4: Repeating the same process; The clusters DEF and BC are comparable and merged together to form a new cluster. We're now left with clusters [(A), (BCDEF)].

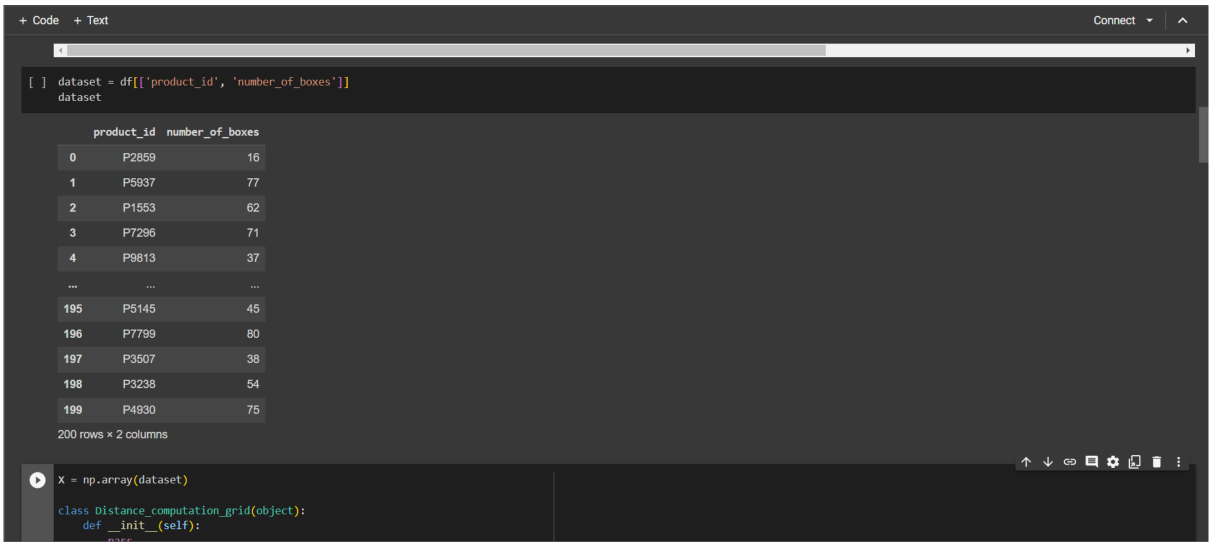
Step 5: At last the two remaining clusters are merged together to form a single cluster [(ABCDEF)].

Output:



```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
# For better readablity;
import matplotlib
matplotlib.rcParams['font.size'] = 16
matplotlib.rcParams['figure.figsize'] = (12, 6)
matplotlib.rcParams['figure.facecolor'] = '#00000000'
df = pd.read_csv('/content/inventory.csv')
df
```

| | transaction_id | product_id | product_name | product_category | product_brand | warehouse_id | warehouse_name | warehouse_location | city_id | supplier_id | supplier_name | contact_id | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | P2859 | Titan T27 | watches | Titan | W7276 | BombayX | Mumbai | C7597 | S3535 | Stewart, Clayton and Martinez | C2483 | 001- |
| 1 | 2 | P5937 | Samsung QLED 21 | tv | Samsung | W7001 | DelhiCaps | Delhi | C7099 | S7973 | Carroll, Brady and Hancock | C8454 | 601.443 |
| 2 | 3 | P1553 | Philips Iron I4 | Electrical App | Philips | W1296 | BombayZ | Mumbai | C7597 | S2142 | Singh-Johnson | C8265 | |
| 3 | 4 | P7296 | Nike Nex | Wear | Nike | W8451 | BangalorIS | Bangalore | C9293 | S3115 | Boyer Ltd | C3118 | (989) |
| 4 | 5 | P9813 | US Polo Black Denim | Wear | US Polo | W8571 | KerelaSwep | Kerela | C6336 | S8922 | Davis Ltd | C7283 | |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | |
| 195 | 196 | P5145 | LG hair dryer | Electrical App | LG | W9621 | GurgaonHDES | Gurgaon | C5676 | S5845 | Dixon and Sons | C9036 | 001- |
| 196 | 197 | P7799 | Apple watch series 8 | watches | Apple | W4993 | HyderabadHub | Hyderabad | C5055 | S7793 | Booth LLC | C7568 | |

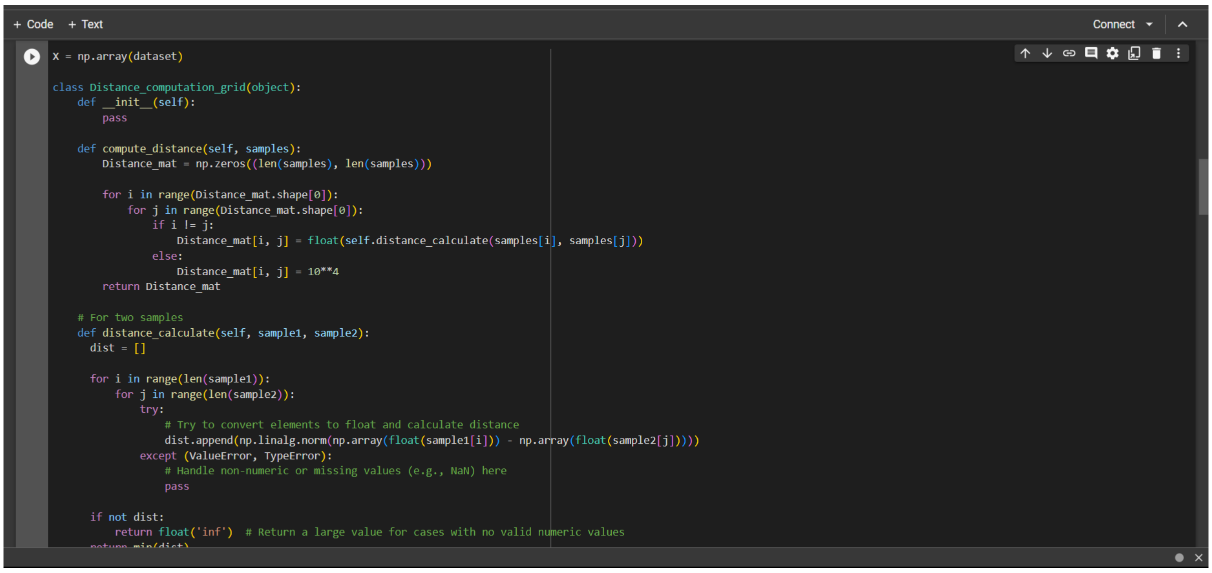


```
dataset = df[['product_id', 'number_of_boxes']]
dataset
```

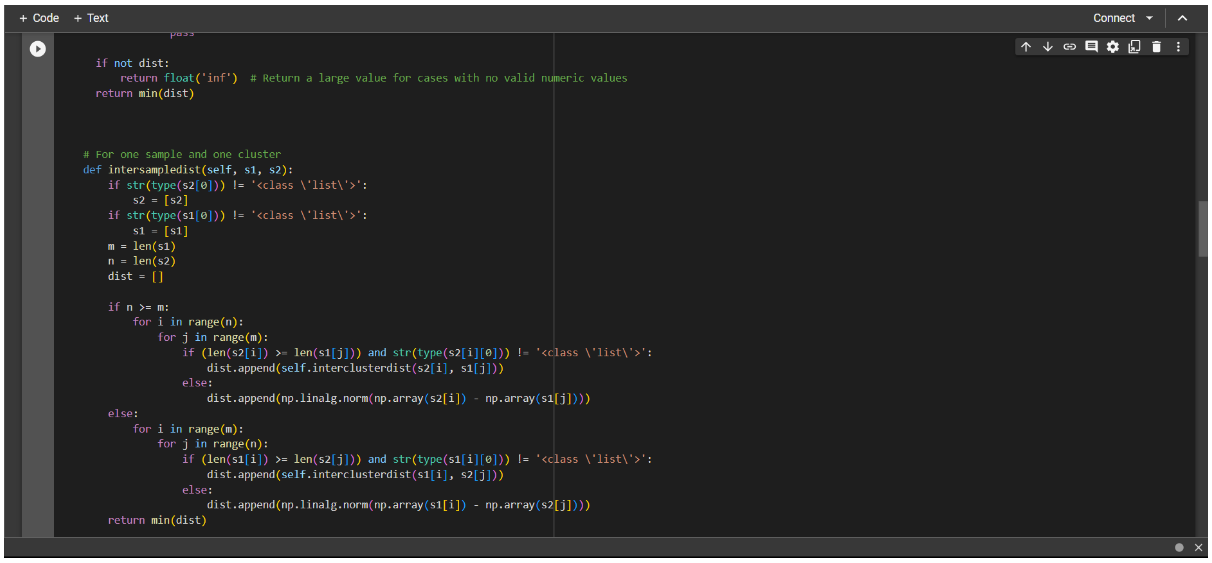
| | product_id | number_of_boxes |
|---|---|---|
| 0 | P2859 | 16 |
| 1 | P5937 | 77 |
| 2 | P1553 | 62 |
| 3 | P7296 | 71 |
| 4 | P9813 | 37 |
| ... | ... | ... |
| 195 | P5145 | 45 |
| 196 | P7799 | 80 |
| 197 | P3507 | 38 |
| 198 | P3238 | 54 |
| 199 | P4930 | 75 |

200 rows × 2 columns

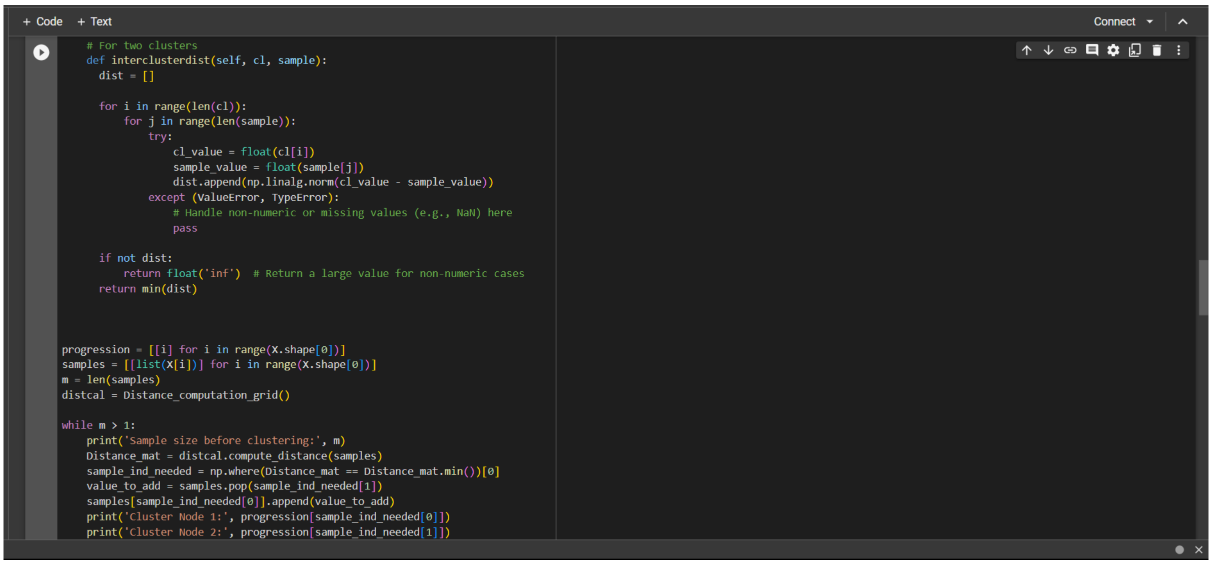
```
X = np.array(dataset)

class Distance_computation_grid(object):
    def __init__(self):
```



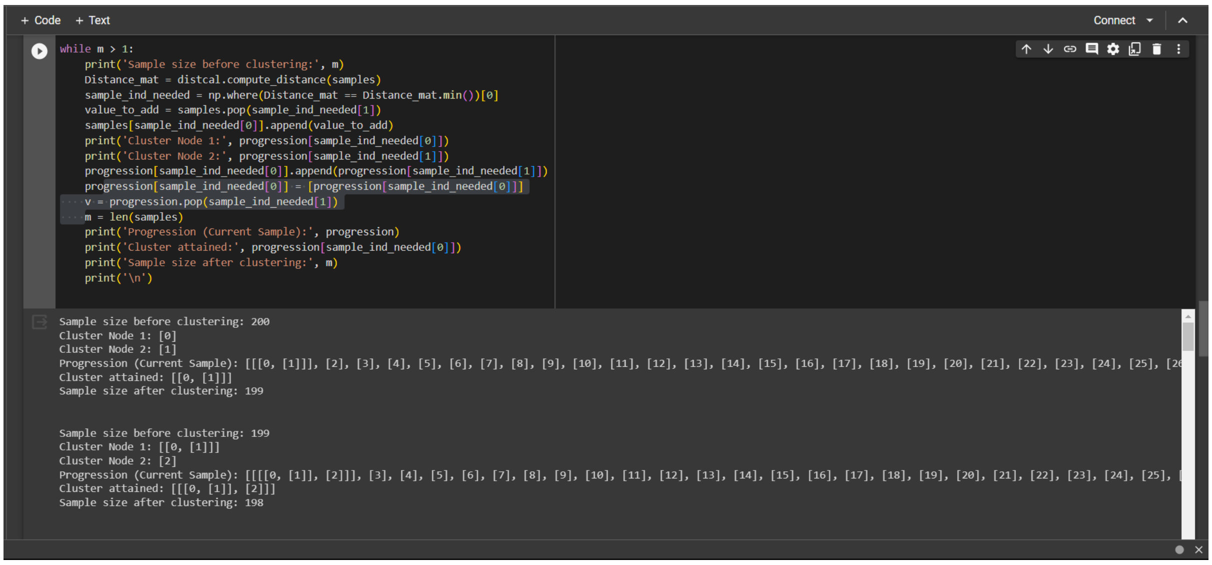
```
X = np.array(dataset)

class Distance_computation_grid(object):
    def __init__(self):
        pass

    def compute_distance(self, samples):
        Distance_mat = np.zeros((len(samples), len(samples)))

        for i in range(Distance_mat.shape[0]):
            for j in range(Distance_mat.shape[0]):
                if i != j:
                    Distance_mat[i, j] = float(self.distance_calculate(samples[i], samples[j]))
                else:
                    Distance_mat[i, j] = 10**4
        return Distance_mat

    # For two samples
    def distance_calculate(self, sample1, sample2):
      dist = []

      for i in range(len(sample1)):
          for j in range(len(sample2)):
              try:
                  # Try to convert elements to float and calculate distance
                  dist.append(np.linalg.norm(np.array(float(sample1[i])) - np.array(float(sample2[j]))))
              except (ValueError, TypeError):
                  # Handle non-numeric or missing values (e.g., NaN) here
                  pass

      if not dist:
          return float('inf')  # Return a large value for cases with no valid numeric values
```



```
                pass

        if not dist:
            return float('inf')  # Return a large value for cases with no valid numeric values
        return min(dist)



    # For one sample and one cluster
    def intersampledist(self, s1, s2):
        if str(type(s2[0])) != '<class \'list\'>':
            s2 = [s2]
        if str(type(s1[0])) != '<class \'list\'>':
            s1 = [s1]
        m = len(s1)
        n = len(s2)
        dist = []

        if n >= m:
            for i in range(n):
                for j in range(m):
                    if (len(s2[i]) >= len(s1[j])) and str(type(s2[i][0])) != '<class \'list\'>':
                        dist.append(self.interclusterdist(s2[i], s1[j]))
                    else:
                        dist.append(np.linalg.norm(np.array(s2[i]) - np.array(s1[j])))
        else:
            for i in range(m):
                for j in range(n):
                    if (len(s1[i]) >= len(s2[j])) and str(type(s1[i][0])) != '<class \'list\'>':
                        dist.append(self.interclusterdist(s1[i], s2[j]))
                    else:
                        dist.append(np.linalg.norm(np.array(s1[i]) - np.array(s2[j])))
        return min(dist)
```

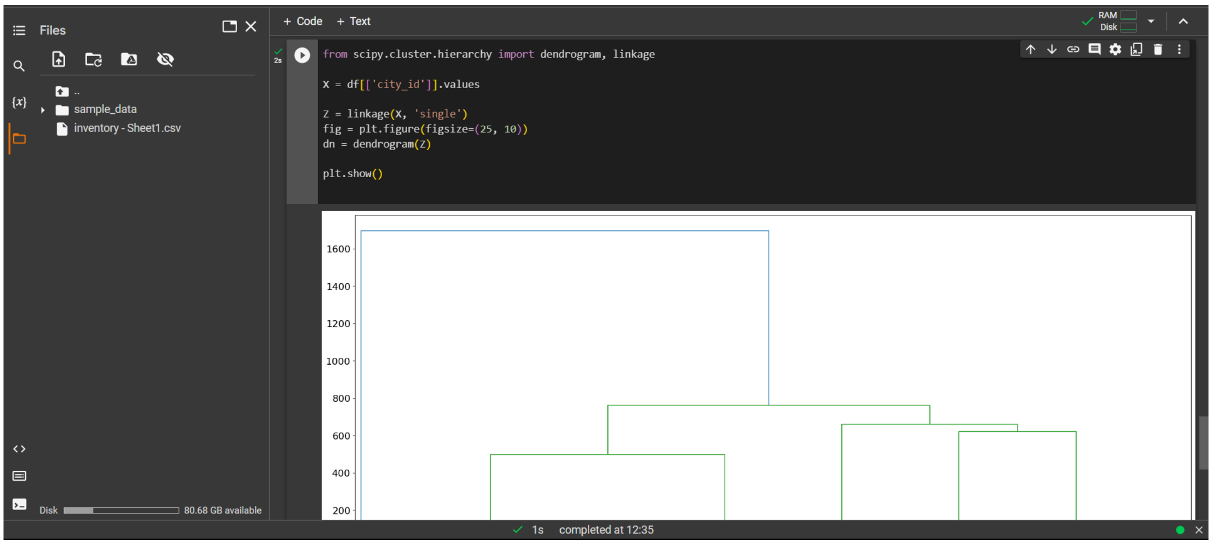


```
    # For two clusters
    def interclusterdist(self, cl, sample):
        dist = []

        for i in range(len(cl)):
            for j in range(len(sample)):
                try:
                    cl_value = float(cl[i])
                    sample_value = float(sample[j])
                    dist.append(np.linalg.norm(cl_value - sample_value))
                except (ValueError, TypeError):
                    # Handle non-numeric or missing values (e.g., NaN) here
                    pass

        if not dist:
            return float('inf')  # Return a large value for non-numeric cases
        return min(dist)



progression = [[i] for i in range(X.shape[0])]
samples = [[list(X[i])] for i in range(X.shape[0])]
m = len(samples)
distcal = Distance_computation_grid()

while m > 1:
    print('Sample size before clustering:', m)
    Distance_mat = distcal.compute_distance(samples)
    sample_ind_needed = np.where(Distance_mat == Distance_mat.min())[0]
    value_to_add = samples.pop(sample_ind_needed[1])
    samples[sample_ind_needed[0]].append(value_to_add)
    print('Cluster Node 1:', progression[sample_ind_needed[0]])
    print('Cluster Node 2:', progression[sample_ind_needed[1]])
```

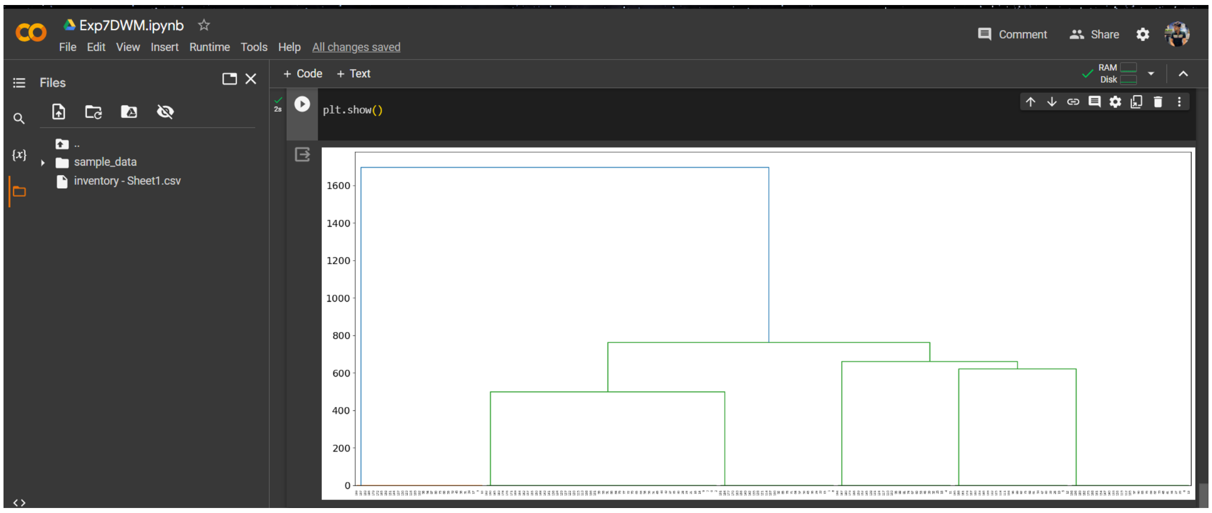


```
while m > 1:
    print('Sample size before clustering:', m)
    Distance_mat = distcal.compute_distance(samples)
    sample_ind_needed = np.where(Distance_mat == Distance_mat.min())[0]
    value_to_add = samples.pop(sample_ind_needed[1])
    samples[sample_ind_needed[0]].append(value_to_add)
    print('Cluster Node 1:', progression[sample_ind_needed[0]])
    print('Cluster Node 2:', progression[sample_ind_needed[1]])
    progression[sample_ind_needed[0]].append(progression[sample_ind_needed[1]])
    progression[sample_ind_needed[0]] = [progression[sample_ind_needed[0]]]
    v = progression.pop(sample_ind_needed[1])
    m = len(samples)
    print('Progression (Current Sample):', progression)
    print('Cluster attained:', progression[sample_ind_needed[0]])
    print('Sample size after clustering:', m)
    print('\n')
```

```
Sample size before clustering: 200
Cluster Node 1: [0]
Cluster Node 2: [1]
Progression (Current Sample): [[[0, [1]]], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26
Cluster attained: [[0, [1]]]
Sample size after clustering: 199


Sample size before clustering: 199
Cluster Node 1: [[0, [1]]]
Cluster Node 2: [2]
Progression (Current Sample): [[[[0, [1]], [2]]], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [
Cluster attained: [[[0, [1]], [2]]]
Sample size after clustering: 198
```



```
from scipy.cluster.hierarchy import dendrogram, linkage

x = df[['city_id']].values

Z = linkage(x, 'single')
fig = plt.figure(figsize=(25, 10))
dn = dendrogram(Z)

plt.show()
```



```
plt.show()
```