Scientific Data Management
Programming Assignment 1:
Hierarchical Clustering

Group 4

1 Introduction

Hierarchical clustering is a general family of clustering algorithms that build nested clusters by merging or splitting them successively. This hierarchy of clusters is represented as a tree (or dendrogram). The root of the tree is the unique cluster that gathers all the samples, the leaves being the clusters with only one sample.[1]

On the first section of this assignment, we have implemented the naive greedy approach for hierarchical clustering with $O(n^3)$ time complexity and $O(n^2)$ memory usage, with the Euclidean distance function and single-linkage method. Later, we compared our results to the built-in hierarchical clustering implementation in the Python Scikit-Learn library.

Secondly, we have implemented one of the more efficient algorithms for hierarchical clustering, utilizing a nearest-neighbor chain with time complexity $O(n^2)$, using the same Euclidean distance function and single-linkage method. Later, we compared our results to the results obtained by built-in hierarchical clustering implementation in the Python Scikit-Learn library.

[1] https://scikit-learn.org/stable/modules/clustering.html

2 Naive Implementation

2.1 Custom Implementation

The custom implementation performs hierarchical clustering using a naive approach.

First, *euclidean_distance()* function calculates the Euclidean distance between two points using NumPy's linalg.norm method. *initialize_distance_matrix()* function initializes a symmetric distance matrix where 'distance_matrix[i][j]' stores the Euclidean distance between points 'i' and 'j'. *naive_hc()* function implements the naive hierarchical clustering algorithm. The algorithm initializes a list of singleton clusters and a distance matrix. Each point starts in its own cluster. The loop continues until only one cluster remains. In each iteration, the algorithm:

- Finds the two clusters (i and j) that are closest to each other by iterating over all possible pairs of clusters and selecting the minimum distance.
- Merges these two clusters and updates the linkage matrix with the merge information.
- Updates the cluster labels.

The function returns a linkage matrix which is required to plot the dendrogram.

2.1.1 Applying to Randomly Generated Data

The naive implementation is demonstrated by generating a random dataset and plotting a dendrogram from the resulting linkage matrix.

We set a random seed to ensure that the same random points are generated every time the code is run, which helps in reproducing the results consistently, and then generate 10 random points in a 2D space. The results are visualized through a dendrogram.

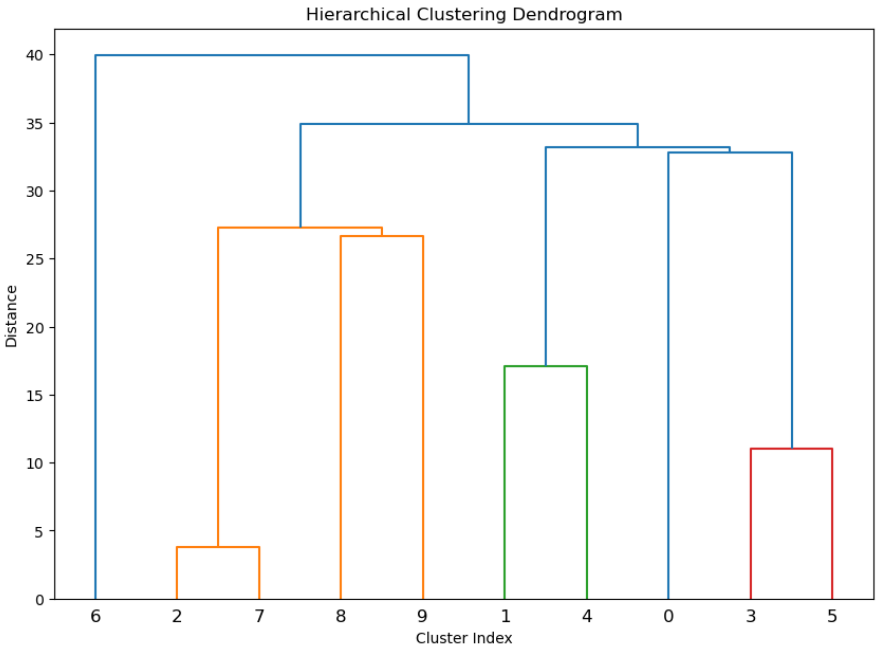

Figure 1: Naive Implementation applied to random data points

2.1.2 Applying to Dataset

To apply our implementation to the credit card dataset of the Kaggle notebook, we first need to load and preprocess our data.

The dataset is loaded into a pandas DataFrame. The customer ID column, which is irrelevant for clustering, is removed. Missing values are forward-filled, i.e., replaced with the last valid value encountered. Data is standardized using StandardScaler, which removes the mean and scales each feature/variable to unit variance. This is crucial for many machine learning algorithms that assume data is normally distributed and/or scales sensitive. After standardization, the data is normalized. Normalization scales input vectors individually to unit norm. This step can be useful when using algorithms that are sensitive to the magnitude of data and helps to maintain numerical stability. PCA is used to reduce the dimensionality of the data to 2 principal components. This is often done to reduce the complexity of the data, improve algorithm performance, and help visualize high-dimensional data. A subset of 1000 data points is sampled from the principal components for clustering. Sampling is necessary because the *naive_hc()* function is computationally intensive, particularly for large datasets. The random state is set for reproducibility. Finally, a dendrogram is plotted using the linkage matrix.

2.2 Built-in Implementation (Agglomerative Clustering)

In order to compare our implementation to those available in Python libraries, we have implemented sklearn.cluster.AgglomerativeClustering which recursively merges pair of clusters of sample data and uses linkage distance.[2]

The AgglomerativeClustering object conducts hierarchical clustering using a bottom-up method where each data point initially forms its own cluster, which are then progressively merged. The method of merging, or linkage criterion, is determined by different metrics:

- Ward linkage aims to minimize the total within-cluster variance. This approach reduces the sum of squared differences within all clusters, similar to the objective of k-means clustering but applied in a hierarchical agglomerative manner.
- Complete or maximum linkage reduces the maximum distance between any two observations in different clusters.
- Average linkage decreases the mean distance between all observations across every pair of clusters.
- Single linkage focuses on minimizing the distance between the nearest members of different clusters.

[2] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html

Although single linkage is sensitive to noisy data, it can be computed very efficiently, making it suitable for hierarchical clustering of large datasets. Additionally, single linkage can effectively handle non-globular data shapes.[3]

The built-in implementation also uses single-linkage to achieve efficiency with the large dataset and is implemented on the same data sample derived from a PCA-reduced dataset sampled down to 1000 observations, as described previously. The implementation is visualized with a dendrogram.

2.3 Comparing Implementations

2.3.1 Dendrograms

For the purpose of verifying our algorithm, we compared the results on a small subset of the data and visualized it as dendrograms. The following figures show results for both algorithms implemented on n = 100 rows and n = 1000 rows:

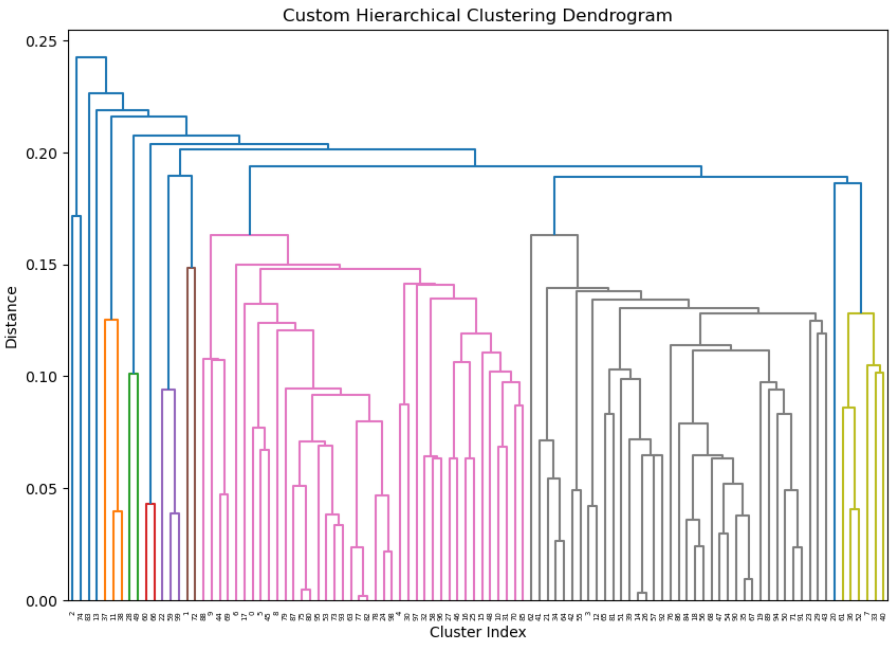

Figure 2: Naive Implementation Dendrogram for 100 rows

[3]https://scikit-learn.org/stable/modules/clustering.html

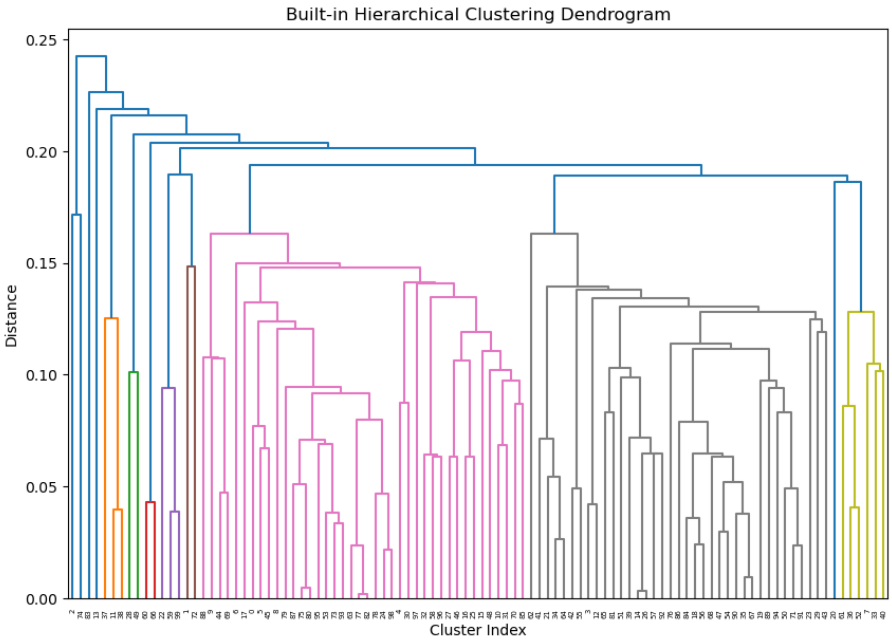

Figure 3: AgglomerativeClustering Dendrogram for 100 rows

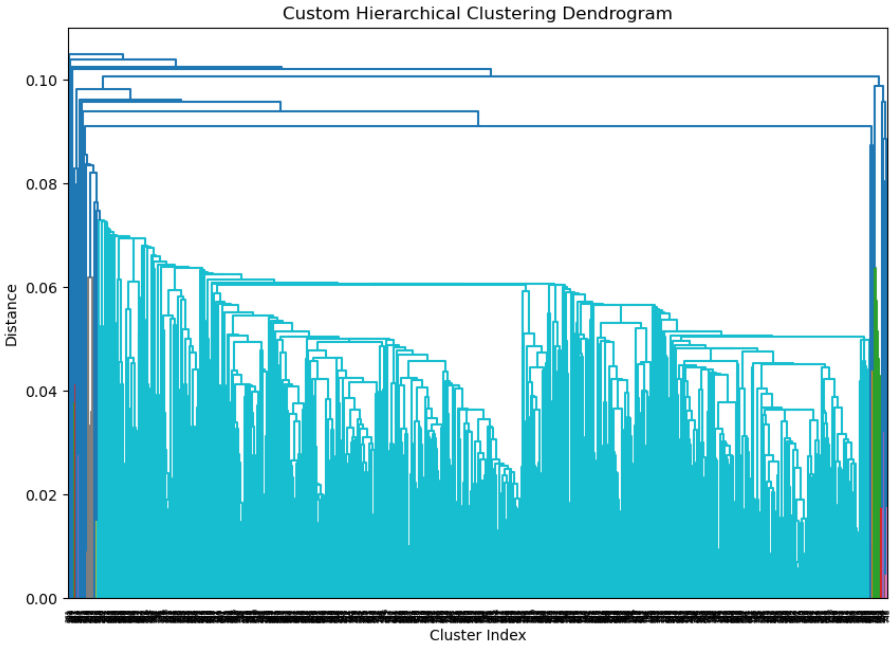

Figure 4: Naive Implementation Dendrogram for 1000 rows

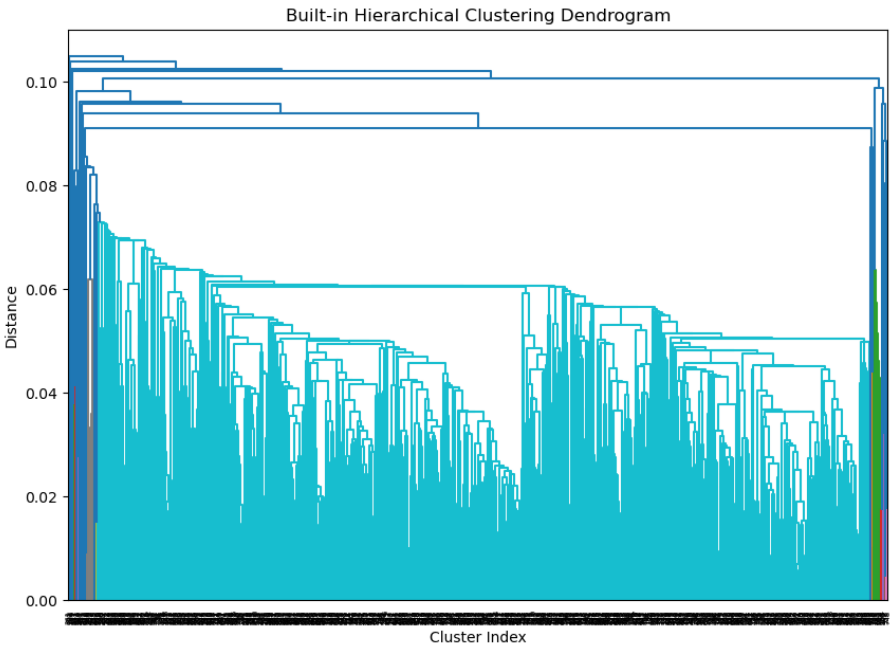

Figure 5: AgglomerativeClustering Dendrogram for 1000 rows

2.3.2 Execution Times

To evaluate the runtime of the naive hierarchical clustering function and compare it with a built-in implementation such as AgglomerativeClustering from Scikit-Learn, we have performed the following steps:

1. Select Sample Sizes: Used different sample sizes from the credit card dataset of the Kaggle notebook, such as n=100, 500, 1000, 1500 rows.

2. Measure Execution Time: For each sample size, measured the time it takes to run both the naive and the built-in clustering algorithms.

3. Plot the Results: Using the matplotlib library, visualized the runtime comparison as a function of sample size.

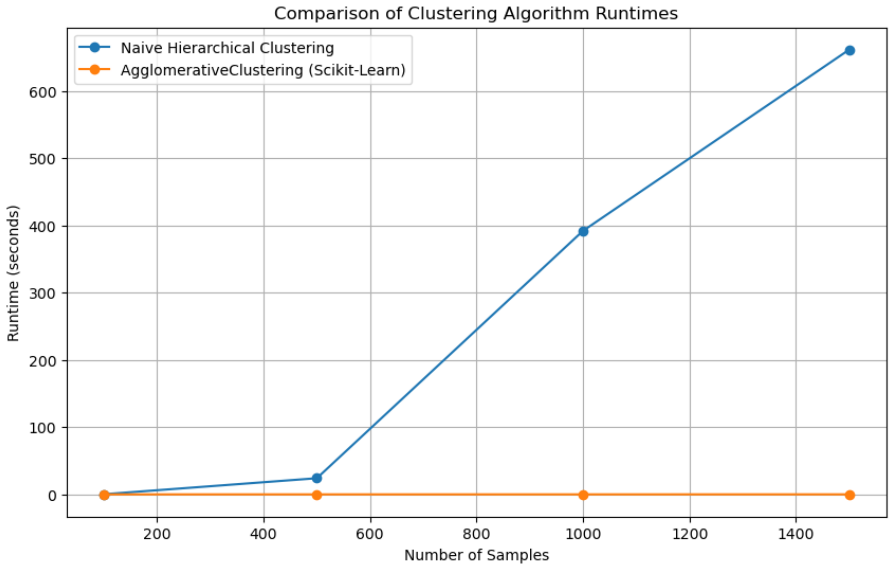

Figure 6: Comparison of Runtimes

This plot shows how each algorithm scales with increasing sample sizes. The custom naive implementation is significantly slower, especially as the number of samples increases, due to its less efficient complexity. This comparison clearly demonstrates the benefits of using optimized, built-in methods for large datasets, especially in terms of computational efficiency.

The algorithms were implemented on hardware with: Intel Core i7-1260P with 12 cores and 16 logical processors, 32 GB of installed physical memory, nearly 1 TB of storage, running Windows 11 Pro with Python 3.10.9 and Pandas version 2.1.1 and Scikit-Learn version 1.3.0. Scikit-Learn's implementations

are optimized to make better use of multiple cores through internal parallelization (where possible), unlike the naive approach which does not parallelize computations. The naive approach is also inefficient in terms of memory usage, especially with handling distance matrices that grow quadratically with the number of data points, leading to substantial overhead and slower access patterns. Evaluation of the runtime for each sample size and for each clustering method is as following:

Runtimes (seconds)				
Sample Sizes	n=100	n=500	n=1000	n=1500
Naive:	0.216000556	23.99701166	391.76409	661.085194
Sklearn:	0.003617286	0.0	0.0136358	0.03355216

The results we are seeing are plausible and illustrate a critical point about the efficiency of built-in algorithms versus a naive implementation. The AgglomerativeClustering from Scikit-Learn is highly optimized, utilizing efficient data structures and algorithms that drastically reduce computation time and complexity. While the naive implementation provides educational insight into how algorithms work, it is not suitable for real-world applications where performance and speed are critical.

Theoretical Complexity: The naive hierarchical clustering algorithm that was implemented is expected to run in $O(n^3)$ time complexity, where n is the number of data points. This is due to the algorithm repeatedly calculating the minimum distance between clusters and updating the distance matrix in each iteration until all points are merged into one cluster.

Observed Behavior: As shown in our measurements, the runtime for the naive implementation increases significantly with the number of data points, reflecting the cubic growth predicted by the $O(n^3)$ complexity. Specifically, as the number of data points increases from 100 to 1500, the runtime grows from about 0.216 seconds to over 661 seconds, which is consistent with a cubic growth rate.

3 Efficient Implementation

3.1 Custom Nearest-Neighbour Chain Algorithm (NNC)

The Nearest Neighbor Chain algorithm is implemented using a stack-based approach and uses a distance matrix to calculate the distances from each to all other points beforehand. The algorithm iterates through the dataset, where each point represents a cluster. These clusters are merged based on their nearest neighbour using single linkage until only one cluster remains.

Each iteration takes the top cluster off the stack and computes the closest cluster. If this cluster is not on the current top of the stack both are added to the stack. Otherwise both clusters will be merged in the set of all clusters. Key components of the implementation include:

- Each cluster is labeled with a unique index for reference in the linkage matrix.
- The algorithm precomputes a distances matrix between all pairs of points to improve efficiency.
- Each point is wrapped to form a cluster structure, which represents the set of all clusters.
- The *find_closest_cluster()* function computes the closest cluster in the set of all clusters
- The algorithm uses consistent tie breaking based on the indexed cluster labels if more than one nearest neighbour candidate is found.
- The *merge_clusters()* function combines two clusters into a single cluster.
- The *add_cluster _label()* function adds a new indexed cluster after a merge.

The function returns a linkage matrix to plot a dendogram based on the input data.

3.1.1 Applying NNC to Randomly Generated Data

As in section 2.1.1 we computed the linkage matrix with NNC using random generated data and the same random seed. While the clusters appear to be the same, the order in which they are presented is different.

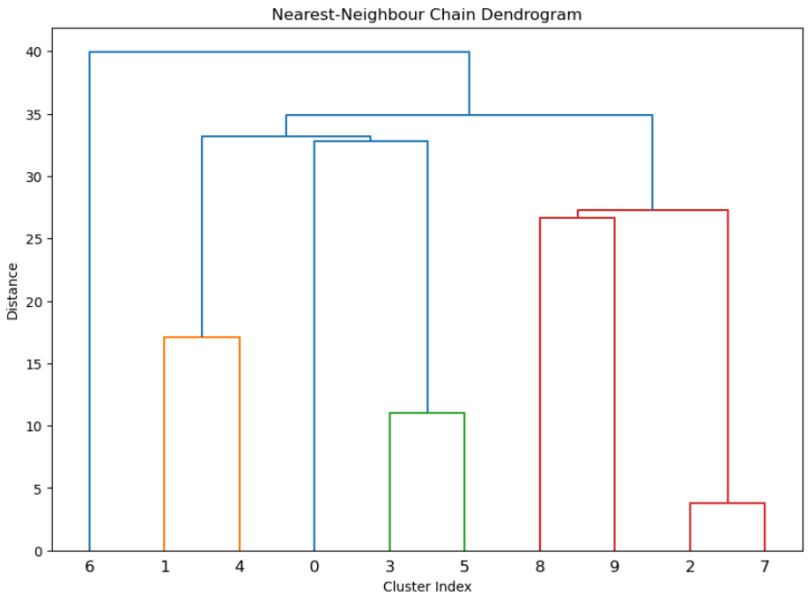

Figure 7: NNC applied to random points

3.1.2 Applying NNC to the KAGGLE Dataset

We applied identical preprocessing to the credit card dataset as in 2.1.2.

3.2 Comparing Implementations

3.2.1 Dendograms

We compared the resulting NNC dendograms with subsets of n=100 and n=1000 (sampled from the KAGGLE credit card dataset) to the sklearn built-in implementation of Agglomerative Clustering. For better comparison we reorderd the NNC linkage matrix with the *distance_sort*[4] argument option of the dendogram function.

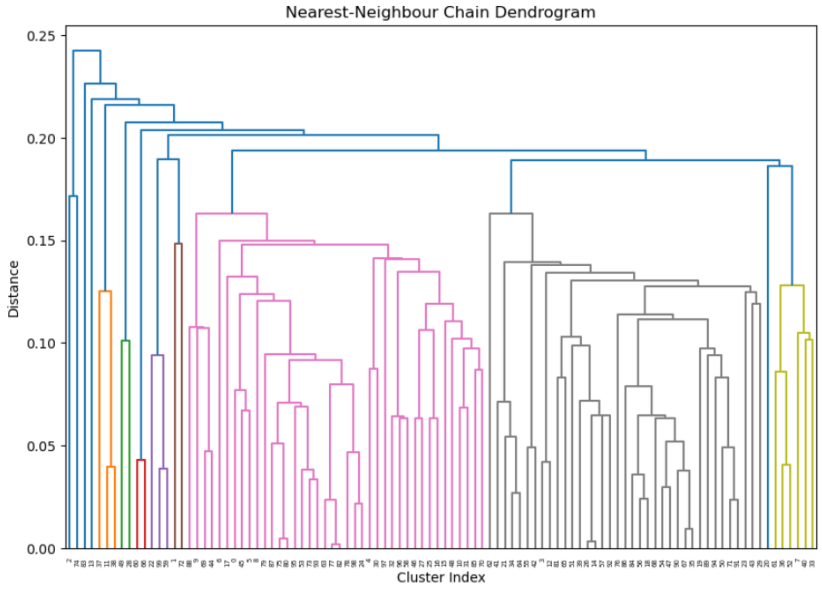

Figure 8: NNC Implementation Dendrogram for 100 rows

[4]https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.dendrogram.html

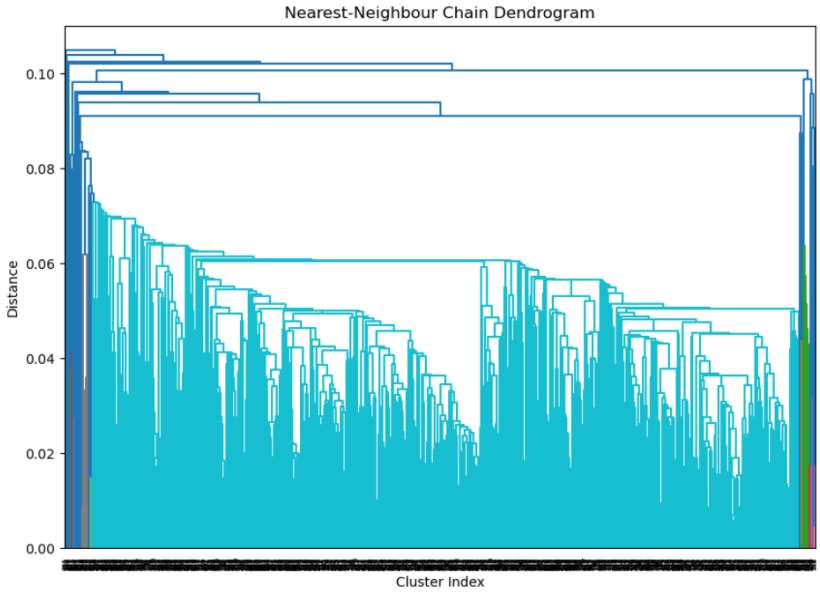

Figure 9: NNC Implementation Dendrogram for 1000 rows

3.2.2 Execution Times

We added NNC to the evalutation of 2.3.2 with an additional data sample of n=2000. The resulting graph shows the the runtimes of each clustering method that we used.

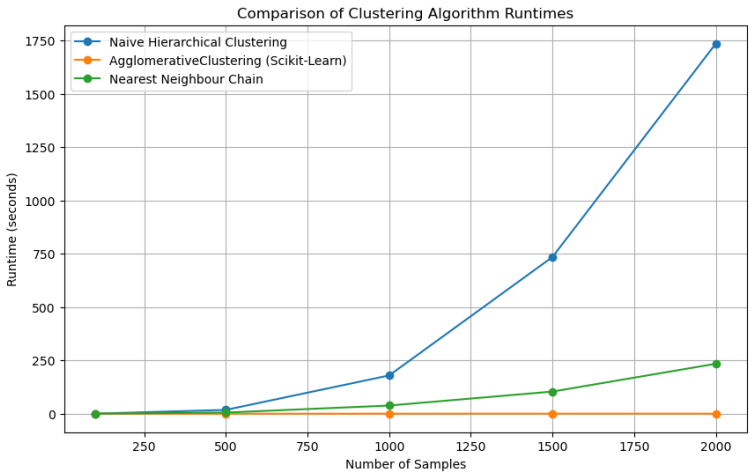

Figure 10: Comparison of Runtimes

The algorithms ran on hardware with: Intel Core i7-1355U with 12 cores, 32GB of physical memory, running Windows 11 Pro, 760GB of free storage Python 3.11.8, Pandas version 2.2.1 and Scikit-Learn version 1.4.1.

Runtimes (seconds)					
Sample Sizes	n=100	n=500	n=1000	n=1500	n=2000
Naive:	0.1532	17.7958	179.4143	735.1778	1736.0754
Sklearn:	0.01794	0.0074	0.0228	0.0427	0.0733
NNC:	0.0632	5.2874	38.1957	103.7897	234.1310

Theoretical Complexity: NNC is supposed to run at a time complexity of $O(n^2)$, where n is the number of data points. This is due to each iteration requiring O(n) time to find the minimum distance to at most n clusters. Additionally the total number of iterations is O(n) since every cluster will be added to the stack only once. Each iteration will ether remove two clusters from the stack or add one. This results to a time complexity of $O(n^2)$.

Observed Behavior: Taken from our measurements, the runtime for the NNC implementation increases in at least a squared manner corresponding to the number of data points. As the number of data points increases from 500 to 1000, the runtime grows from 5.2874 seconds to 38.1957 seconds, which suggests an above squared growth rate.

Further examination - Time ratio computation:

$$\frac{T(2n)}{T(n)} = growth\ rate \tag{1}$$

$$\frac{T(1000)}{T(500)} = \frac{38.1957}{5.2874} \approx 7.22 \tag{2}$$

$$\frac{T(2000)}{T(1000)} = \frac{234.1310}{38.1957} \approx 6.13 \tag{3}$$

$O(n^2)$ time complexity has a growth factor of 4, which consequently means that the NNC implementation is above that. One issue might be the remove operations in the deque that stores its clusters, since each remove operation has a time complexity of $O(n)$ where n is the number of clusters at each iteration.