COMP-6651: Clustering Algorithms Analysis

Nitheesh Kumar Kambala (40299620), Divyesh Pravinkumar Patel (40301889),
Sotirios Damas (40317602), Farid Faraji (40324126)

I. Introduction

Clustering algorithms play a pivotal role in data analysis, with performance critically dependent on the selection of dissimilarity measures and computational efficiency. A diverse set of methodologies is considered, including density-based techniques (DBSCAN and OPTICS), distribution-based approaches (Gaussian Mixture Models), centroid/medoid-based algorithms (k-means and k-medoids), exemplar-based strategies (Affinity Propagation and Mean-Shift), and hierarchical methods (BIRCH and Agglomerative Hierarchy). Particular attention is given to computing dissimilarity coefficients using (weighted) Euclidean distance, while addressing the challenges of normalization and relative weighting that can substantially affect clustering outcomes.

Detailed algorithmic descriptions are provided with comprehensive input/output parameter lists and thorough complexity analyses. In parallel, several clustering evaluation metrics—including the Silhouette Score, Davies-Bouldin Index, Calinski-Harabasz Index, Adjusted Rand Index, Mutual Information, Split, and Diameter—are mathematically defined and interpreted, establishing a robust framework for performance assessment and comparative analysis.

Nuances in k-means clustering highlight key limitations, such as the inadequacy of relying solely on the Silhouette Score as an accuracy indicator. Illustrative examples demonstrate that an exact k-means solution can yield a poor silhouette score, while enhancements based on Elkan's triangle inequality approach offer significant improvements in computational efficiency. These insights provide valuable guidance for the design of scalable and robust clustering techniques.

II. Computation of the dissimilarity coefficients

The dissimilarity between data points in clustering algorithms is often measured using distance metrics, among which the (weighted) Euclidean distance is one of the most widely used. The Euclidean distance between two points x and y in an n-dimensional space is defined as:

$$D(x,y) = \sqrt{\sum_{i=1}^{n} w_i (x_i - y_i)^2}, \tag{1}$$

where w_i represents the weight assigned to the ith feature [9]. The inclusion of weights allows for the adjustment of the importance of different dimensions in the computation of dissimilarity, which can be crucial when features are measured on different scales or have varying significance.

A key consideration in the computation of Euclidean distance is the need for normalization. If features have different units or ranges, those with larger scales can disproportionately influence the distance measure, leading to biased clustering results. Common normalization techniques include:

- **Min-Max Scaling**: Rescales features to a range of [0,1] using:
$$x_i' = \frac{x_i - \min(x_i)}{\max(x_i) - \min(x_i)}. \tag{2}$$
This ensures that all features contribute equally but is sensitive to outliers.
- **Z-Score Standardization**: Transforms data to have zero mean and unit variance:
$$x_i' = \frac{x_i - \mu_i}{\sigma_i}. \tag{3}$$
This method is robust to different feature scales and is commonly used in clustering applications.
- **Unit Vector Scaling**: Normalizes each data point to have unit length:
$$x_i' = \frac{x_i}{\|x\|}. \tag{4}$$
This is particularly useful when the magnitude of the vector is not relevant, such as in text clustering. The choice of normalization and weighting scheme significantly influences clustering performance. Without proper normalization, distance-based clustering algorithms such as k-means or hierarchical clustering may produce misleading results, grouping points based on dominant features rather than meaningful similarity patterns [4]. Additionally, improper weighting can lead to feature dominance, where certain dimensions drive the clustering process while others are ignored.
To mitigate these issues, practitioners often conduct sensitivity analyses by experimenting with different normalization techniques and weightings, ensuring that clusters reflect meaningful patterns in the data rather than arbitrary scale differences.

III. Algorithm descriptions and their detailed complexity analysis

A. Density based

a) DBSCAN (Density-Based Spatial Clustering of Applications with Noise): DBSCAN requires two parameters: the minimum number of points ($MinPts$) and the neighborhood radius (ϵ). It groups densely packed data points into a single cluster by creating a circle of radius ϵ around each data point and classifying them into **Core points**, **Border points**, and **Noise** [1].

A data point is a **Core point** if the circle around it contains at least $MinPts$ points. If the number of points is less than $MinPts$, it is classified as a **Border point**. If no other data points exist within the ϵ radius, it is treated as **Noise**.

If the distance between two core points is less than ϵ, then all data points within their ϵ-radius are merged into a single cluster; otherwise, two separate clusters are maintained. This process repeats iteratively until all data points have been visited. This algorithm's main advantage is that it can discover arbitrarily shaped clusters and resistance to the outliers. However, the choice of ϵ and $MinPts$ significantly affects the clustering results, and an inappropriate selection may lead to poor cluster formation which can be considered the main drawback of this algorithm.

Algorithm 1 DBSCAN Algorithm

```
1: Input: A set of data points $P$, minimum number of points $MinPts$, neighborhood radius $\epsilon$
2: Output: A set of clusters and noise points
3: Initialize all points in $P$ as unvisited
4: Initialize an empty list of clusters $C = \emptyset$
5: for each unvisited point $p \in P$ do
6:     if $p$ is unvisited then
7:         Mark $p$ as visited
8:         Retrieve all points in the $\epsilon$-neighborhood of $p$
9:         if the number of points in the neighborhood of $p$ is greater than or equal to $MinPts$ then
10:            Create a new cluster and add $p$ to the cluster
11:            ExpandCluster(p, $MinPts$, $\epsilon$)
12:        else
13:            Mark $p$ as noise
14:        end if
15:    end if
16: end for
17: Return the set of clusters and noise points
```

Algorithm 2 ExpandCluster Algorithm

```
1: Input: Core point $p$, minimum number of points $MinPts$, neighborhood radius $\epsilon$
2: Output: Expanded cluster
3: Initialize the cluster with $p$
4: Initialize the list of points to check with the $\epsilon$ - neighborhood of $p$
5: while there are points to check in the list do
6:     Take the first point $q$ from the list
7:     if $q$ is unvisited then
8:         Mark $q$ as visited
9:         Retrieve all points in the $\epsilon$-neighborhood of $q$
10:        if the number of points in the neighborhood of $q$ is greater than or equal to $MinPts$ then
11:            Add all unvisited neighbors of $q$ to the list
12:        end if
13:    end if
14:    if $q$ is not in any cluster then
15:        Add $q$ to the current cluster
16:    end if
17: end while
```

The detailed complexity analysis of DBSCAN consisting of three main operations is as follows

1) **Distance Calculation:** For each data point, DBSCAN needs to calculate the distance to every other data point to check if it lies within the ϵ-neighborhood. The distance calculation between two data points will be $\mathcal{O}(d)$ where d is the number of dimensions.
2) **Neighborhood Search:** For each data point, DBSCAN checks all other data points to see if they lie within ϵ-radius. This process checks all n points for each data point which leads to $\mathcal{O}(n)$ time per point.
3) **Cluster Expansion:** Once a core point is identified, DBSCAN expands the cluster by checking the neighbors recursively. In the worst case, the expansion could visit all n points if all points are connected.
4) **Total Time Complexity:** For each point p, DBSCAN checks the distance to all other data points and performs neighborhood calculation which corresponds to $\mathcal{O}(n \cdot d)$ time complexity per data point. If we calculate for all the data points the time complexity corresponds to $\mathcal{O}(n^2 \cdot d)$.
5) **Total Space Complexity:** The dataset contains n data points with d features, storing these data points requires $\mathcal{O}(n \cdot d)$ space. The DBSCAN algorithm maintains an assigned cluster labels list and boolean visited flag for each data point which requires additional $\mathcal{O}(n)$ space each. Moreover, the DBSCAN algorithm stores neighborhood information temporarily while expanding the cluster which will result in additional $\mathcal{O}(n)$ space in the worst-case scenario where all data points belong to that cluster. If we consider this information, the space complexity of the DBSCAN algorithm results in $\mathcal{O}(n \cdot d)$.

Thus, the overall time complexity for the DBSCAN algorithm is $\mathcal{O}(n^2 \cdot d)$ and the space complexity is $\mathcal{O}(n \cdot d)$ where n is the number of data points and d is the number of dimensions in each data point.

b) OPTICS (Ordering Points to Identify Clustering Structure): OPTICS is an extension of DBSCAN which handles varying densities in a dataset. OPTICS creates an ordered list or reachability plot of data points based on density-reachability to extract clusters which results in better clustering even if the data points are less dense.

The key concepts of OPTICS are **Core distance**: the smallest ϵ that makes a point a core point i.e., the neighborhood has at least $MinPts$ data points, **Reachability Distance**: the distance at which a point can be reached from a core point and a **Priority Queue**: queue used to expand clusters by visiting the least-reachable points first [2].

OPTICS utilizes these key concepts by iteratively expanding clusters starting from core points. It maintains a priority queue to ensure that points are processed in increasing order of their reachability distance. Each time a new core point is encountered, its directly reachable neighbors are added to the queue with updated reachability distances. This dynamic ordering enables OPTICS to explore dense regions first while gradually moving towards less dense areas, ensuring a smooth clustering structure even in datasets with varying densities.

To extract clusters from the ordered list with reachability distances provided by the OPTICS algorithm we have to first identify valleys and peaks. A valley corresponds to a dense cluster because data points in that region have low reachability distances and the sparse regions with high reachability distances are considered as peaks. Peaks are used to identify transitions between clusters or noise. A user-defined or automatic threshold ϵ' is defined to separate clusters. A cluster starts when the reachability distance drops below ϵ' and ends when it exceeds ϵ'. While the noise is represented by isolated points with high reachability distances greater than ϵ'. Finally, we extract all continuous valleys as clusters and the peaks indicate separations between clusters.

Algorithm 3 OPTICS Algorithm

```
1: Input: A set of points $P$, minimum number of points $MinPts$, maximum distance $\epsilon$
2: Output: Ordered list of points with reachability distances
3: Initialize all points as unvisited
4: Initialize an empty ordered list $O$
5: for each unvisited point $p \in P$ do
6:     ExpandClusterOrder($p, P, MinPts, \epsilon, O$)
7: end for
8: Return the ordered list $O$
```

Algorithm 4 ExpandClusterOrder

```
1: Input: A point $p$, dataset $P$, $MinPts$, $\epsilon$, ordered list $O$
2: Output: Updates the order list $O$ with reachability distances
3: Compute the $\epsilon$-neighborhood of $p$
4: Compute core distance of $p$ (distance to its $MinPts$-th nearest neighbor)
5: Add $p$ to ordered list $O$ with reachability distance undefined
6: if $p$ is a core point then
7:     Initialize a priority queue $Q$ sorted by reachability distance
8:     For each neighbor $q$ of $p$, update reachability distance if necessary and add $q$ to $Q$
9:     while $Q$ is not empty do
10:        Extract the point $r$ with the smallest reachability distance from $Q$
11:        Compute $\epsilon$-neighborhood of $r$
12:        Compute core distance of $r$
13:        Add $r$ to ordered list $O$
14:        if $r$ is a core point then
15:            Update reachability distances of its neighbors and add them to $Q$
16:        end if
17:    end while
18: end if
```

The detailed complexity analysis of the OPTICS algorithm consisting of two main operations is as follows

1) **ϵ-Neighborhood Search:** Finding all neighbors of a point within radius ϵ. For each data point, we compare distance with all other n points each having d features. Each data point takes $\mathcal{O}(n \cdot d)$ time to find neighbors and for all data points the time complexity results in $\mathcal{O}(n^2 \cdot d)$.
2) **Priority Queue operations:** We use this priority queue to maintain and update the reachability distances of each data point from the current core point. Inserting and removing points take $\mathcal{O}(\log n)$ per operation and in the worst case, we insert all n points leading to $\mathcal{O}(n \log n)$ time.
3) **Total Time Complexity:** As we need to perform ϵ-Neighborhood Search for all data points similar to **DBSCAN**, the time complexity of this operation for all data points results to $\mathcal{O}(n^2 \cdot d)$ time and the priority queue operations which takes $\mathcal{O}(n \log n)$ time in the worst case. By considering the rules of Big-O notation and ignoring the lower order terms the time complexity of OPTICS corresponds to $\mathcal{O}(n^2 \cdot d)$.
4) **Total Space Complexity:** Storing n data points containing d features take up $\mathcal{O}(n \cdot d)$ space. While expanding the cluster order we need an additional priority queue to process the data points with respect to reachability distance and in the worst case we have to insert all n data points in the queue which takes up $\mathcal{O}(n)$ space. The result of the OPTICS algorithm is an ordered list of all data points along with their reachability distances, so we need additional $\mathcal{O}(n)$ space. By considering all these requirements, the overall space complexity of the OPTICS algorithm is $\mathcal{O}(n \cdot d)$.

Since the OPTICS algorithm is an extension of the DBSCAN algorithm which resolves the drawback of fixed neighborhood radius density issue, the time and space complexity of both algorithms are similar in worst case scenario i.e, the overall time complexity of the OPTICS algorithm is $\mathcal{O}(n^2 \cdot d)$ and the space complexity is $\mathcal{O}(n \cdot d)$ where n is the number of data points and d is the number of dimensions in each data point.

B. Distribution based

a) Gaussian Mixture Model (GMM) Algorithm: The Gaussian Mixture Model (GMM) is a distribution-based clustering method that assumes each cluster is generated from a distinct Gaussian (normal) distribution. A GMM is typically estimated using the Expectation-Maximization (EM) algorithm, which alternates between assigning data points to Gaussian components probabilistically (E-step) and updating the parameters (M-step) to maximize the overall data likelihood [10], [11]. The user must specify the number of Gaussian components K in advance or use criteria like the Bayesian Information Criterion (BIC) to select it.

The detailed complexity analysis of the Gaussian Mixture Model using EM consists of the following main operations:

1) **Probability Calculation (E-step):** For each iteration, we compute the likelihood of each data point $\mathbf{x}_i$ under each of the K Gaussian components:
    - Evaluating a d-dimensional Gaussian $\mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ can be $\mathcal{O}(d^2)$ after precomputing the inverse of the covariance matrix, since the main cost is the matrix-vector multiplication and determinant calculation.

Algorithm 5 Gaussian Mixture Model (GMM) via EM Algorithm

```
1: Input:
     • Dataset $\mathcal{D} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, with each $\mathbf{x}_i \in \mathbb{R}^d$
     • Number of components $K$
     • (Optional) Initial parameters $\{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k\}_{k=1}^K$
     • Convergence threshold or maximum iterations $\ell$
2: Output:
     • Estimated parameters $\{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k\}_{k=1}^K$
     • Responsibilities $\gamma_{i,k}$ (soft cluster assignments)
3: Initialization: Randomly initialize or use a heuristic (e.g., $k$-means) to set $\boldsymbol{\mu}_k$, $\boldsymbol{\Sigma}_k$, and $\pi_k$ for each of the $K$ Gaussian components.
4: while not converged and iteration $\le \ell$ do
5:   E-step (Compute Responsibilities):
       $$\gamma_{i,k} = \frac{\pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^K \pi_j \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$
       for each $i = 1, \ldots, n,\ k = 1, \ldots, K$
6:   M-step (Update Parameters):
       $$N_k = \sum_{i=1}^n \gamma_{i,k}, \quad \pi_k = \frac{N_k}{n},$$
       $$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{i=1}^n \gamma_{i,k} \mathbf{x}_i,$$
       $$\boldsymbol{\Sigma}_k = \frac{1}{N_k} \sum_{i=1}^n \gamma_{i,k} (\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^\top$$
7:   Evaluate the log-likelihood or parameter change to check for convergence
8: end while
9: Return: The final parameters $\{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k\}$ and the responsibilities $\gamma_{i,k}$ for each data point $\mathbf{x}_i$
```

- Overall, updating responsibilities for n data points across K components yields $\mathcal{O}(n \cdot K \cdot d^2)$ per iteration.

2) **Parameter Updates (M-step):** Updating the parameters $\{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k\}$ similarly involves summations over all n data points, weighted by $\gamma_{i,k}$.
   - Computing means $\boldsymbol{\mu}_k$ costs $\mathcal{O}(n \cdot K \cdot d)$.
   - Updating covariances $\boldsymbol{\Sigma}_k$ can cost up to $\mathcal{O}(n \cdot K \cdot d^2)$ if each covariance is full rank (i.e., not diagonal).

   Combined with the E-step, the total time per iteration often remains on the order of $\mathcal{O}(n \cdot K \cdot d^2)$ for full covariance matrices.
3) **Overall Time Complexity:** Let ℓ be the number of EM iterations until convergence. Then the total time complexity is approximately $\mathcal{O}(\ell \cdot n \cdot K \cdot d^2)$. If simpler (e.g., diagonal) covariances are assumed, the d^2 factor can reduce to $\mathcal{O}(d)$.
4) **Total Space Complexity:**
   - *Dataset Storage:* $\mathcal{O}(n \cdot d)$.
   - *Responsibilities:* $\mathcal{O}(n \cdot K)$ for storing $\gamma_{i,k}$.
   - *Parameters:* $\mathcal{O}(K \cdot d^2)$ for storing full covariance matrices, plus $\mathcal{O}(K \cdot d)$ for means and $\mathcal{O}(K)$ for mixing coefficients.

   Hence, the total space complexity is $\mathcal{O}(n \cdot d + n \cdot K + K \cdot d^2)$, which for large n is dominated by $\mathcal{O}(n \cdot d)$ and $\mathcal{O}(n \cdot K)$.

Thus, in the base case (with full covariance matrices), the overall time complexity for the Gaussian Mixture Model (EM algorithm) is $\mathcal{O}(\ell \cdot n \cdot K \cdot d^2)$ and the space complexity is $\mathcal{O}(n \cdot K + K \cdot d^2)$, where n is the number of data points, K is the number of Gaussian components, d is the dimensionality, and ℓ is the number of EM iterations.

C. Centroid/medoid based

a) K-Means Algorithm: K-Means is a centroid-based clustering algorithm that partitions a dataset into K clusters. The algorithm iteratively refines the cluster assignments by minimizing the intra-cluster variance (distance between points in the same cluster) and maximizing the inter-cluster variance (distance between different clusters). The key steps in the algorithm are initialization of centroids, assignment of points to the nearest centroids, and recalculation of centroids until convergence is reached. [18]–[20]

The cost function of the k-means algorithm is the sum of squared distances from each data point to the centroid of its assigned cluster:

$$J = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - M_i\|^2$$

where:

- k is the number of clusters
- C_i is the set of points in the i-th cluster
- M_i is the centroid of the i-th cluster

Algorithm 6 K-Means Algorithm

```
1: Input: A set of data points $P$, number of clusters $K$
2: Output: A set of clusters $C = \{C_1, C_2, \ldots, C_K\}$ and their corresponding centroids $M = \{M_1, M_2, \ldots, M_K\}$
3: Select k random points as initial centroids: Randomly pick $K$ data points from the data points $P$ to act as initial cluster centroids $M = \{M_1, M_2, \ldots, M_K\}$.
4: repeat
5:   for each point $p \in P$ do
6:     Assign $p$ to the cluster $C_i$ with the closest centroid $M_i$ by calculating Euclidean distance.
7:   end for
8:   for each cluster $C_i$ do
9:     Recalculate the centroid $M_i$ of cluster $C_i$ as the mean of all data points in cluster $C_i$.
10:  end for
11: until Convergence is reached (i.e., centroids $M$ do not change)
12: Return the set of clusters $C = \{C_1, C_2, \ldots, C_K\}$ and their corresponding centroids $M = \{M_1, M_2, \ldots, M_K\}$.
```

The detailed complexity analysis of the K-Means Algorithm consists of following main operations

1) **Distance Calculation:** For each point p, the algorithm calculates the distance to each centroid to determine the nearest cluster. The distance calculation for each point will be $\mathcal{O}(d)$, where d is the number of dimensions in the data.
2) **Cluster Assignment:** For each data point, the K-Means algorithm assigns it to the nearest cluster by computing the distance to each centroid. This results in $\mathcal{O}(n \cdot K)$ time complexity, where n is the number of data points.
3) **Centroid Update:** After assigning points to clusters, the centroids are recalculated as the mean of the points in each cluster. The update step takes $\mathcal{O}(n)$ time for each centroid.
4) **Total Time Complexity:** In the worst case, the K-Means algorithm will perform T iterations, where each iteration involves calculating distances for all points and updating centroids. The total time complexity is $\mathcal{O}(T \cdot n \cdot K \cdot d)$, where T is the number of iterations, n is the number of data points, K is the number of clusters, and d is the number of dimensions.
5) **Total Space Complexity:** The K-Means algorithm stores the data points and the centroids. The space complexity is $\mathcal{O}(n \cdot d + K \cdot d)$, where n is the number of data points, d is the number of dimensions, and K is the number of clusters.

Thus, the overall time complexity for the K-Means algorithm is $\mathcal{O}(T \cdot n \cdot K \cdot d)$, and the space complexity is $\mathcal{O}(n \cdot d + K \cdot d)$, where n is the number of data points, K is the number of clusters, and d is the number of dimensions.

b) K-Medoids Algorithm: K-Medoids is a clustering algorithm similar to K-Means but is more robust to outliers. Instead of using centroids (mean points) like in K-Means, K-Medoids selects actual data points as cluster representatives, called *medoids*. The goal of the algorithm is to minimize the total dissimilarity (sum of distances) between points and their assigned medoid. [17], [21], [22]

The detailed complexity analysis of the K-Medoids Algorithm consists of the following main operations

1) **Initialization:** The algorithm randomly selects K medoids from the given n data points. This operation takes $\mathcal{O}(K)$ since it requires choosing K points from the dataset.
2) **Data Point Assignment:** Each data point p is assigned to closest medoid by computing Manhattan distance from all K medoids. Time complexity for this step will be computes its distance to all K centroids. Thus, it takes $\mathcal{O}(n \cdot K \cdot d)$, where d is the number of dimensions in the data.
3) **Swapping step:** For each medoid, we attempt to swap it with every non-medoid data point to minimize the total cost. The total time complexity for this step is $\mathcal{O}(K \cdot n^2 \cdot d)$.
4) **Convergence:** The algorithm runs for I iterations until medoids stabilize, leading to an overall time complexity of $\mathcal{O}(K \cdot n^2 \cdot d)$.
5) **Total Time Complexity:** In the worst case, K-Medoids algorithm will perform T iterations, where each iteration involves calculating distance, assigning data points, and swapping. The total time complexity is $\mathcal{O}(T \cdot K \cdot n^2 \cdot d)$.
6) **Total Space Complexity:** The K-Medoids algorithm stores the data points and the centroids. The space complexity is $\mathcal{O}(n \cdot d + K \cdot d)$.

Algorithm 7 K-Medoids Algorithm

```
1:  Input: A set of data points $P$, number of clusters $K$
2:  Output: A set of clusters $C = \{C_1, C_2, \ldots, C_K\}$ and their corresponding medoids
3:  Randomly select $K$ medoids from the $n$ data points
4:  repeat
5:      for each data point $p \in D$ do
6:          Assign $p$ to the cluster corresponding to the closest medoid by calculating Manhattan distance.
7:      end for
8:      for each medoid $m \in K$ do
9:          for each data point $o$ assigned to $m$ do
10:             Swap $m$ and $o$, then compute the total cost of the new configuration
11:         end for
12:         Select the data point $o$ with the lowest cost as the new medoid
13:     end for
14: until No changes occur in the cluster assignments
15: Return the set of clusters $C = \{C_1, C_2, \ldots, C_K\}$ and their corresponding medoids $M = \{M_1, M_2, \ldots, M_K\}$
```

D. Exemplar based

a) Affinity Propagation clustering algorithm: Consider a set of data points $\{x_1, x_2, \ldots, x_N\}$. In Affinity Propagation, the algorithm is provided with a similarity matrix S, where each entry $s(i, j)$ quantifies how well data point j can serve as an exemplar for data point i. This method accepts any type of similarity measure—for instance, negative Euclidean distance for real-valued data or the Jaccard coefficient for nonmetric data—making it versatile for various applications. Instead of requiring the number of clusters to be pre-specified, the algorithm incorporates a preference value $s(j, j)$ for each data point, so that those with higher preferences are more likely to be chosen as exemplars. These preferences, which may be set to the median or minimum of the input similarities, directly influence the number of clusters generated.

Affinity Propagation aims to determine the optimal set of exemplars by maximizing the overall similarity between each data point and its assigned exemplar. This objective is achieved through an iterative message-passing process where two types of messages are exchanged: responsibilities and availabilities. The responsibility, $r(i, k)$, indicates how suitable point k is to serve as the exemplar for point i, taking into account all other potential exemplars. Meanwhile, the availability, $a(i, k)$, represents the accumulated evidence supporting point k as a viable exemplar for point i, reflecting endorsements from other data points.

During the iterative process, both responsibilities and availabilities are updated until convergence or a set number of iterations is reached. Responsibilities are sent from data points to candidate exemplars to express the degree of preference for each candidate, while availabilities are passed in the reverse direction to indicate the overall support for each candidate as an exemplar. This continuous exchange of messages enables the algorithm to converge on a set of exemplars that maximize the net similarity between data points and their corresponding exemplars, thereby forming clusters without requiring an a priori specification of their number.

$$r[i,j] = \begin{cases} s[i,j] - \max\limits_{k\neq j}\big(a[i,k] + s[i,k]\big), & i \neq j, \\ s[i,j] - \max\limits_{k\neq j}\{s[i,k]\}, & i = j, \end{cases} \tag{1}$$

In parallel, the availability $a(i,k)$ reflects the accumulated evidence that point k should serve as the exemplar for point i, and is updated according to:

$$a[i,j] = \begin{cases} \min\{0,\, r[j,j]\} \;+\; \sum\limits_{k\neq i,j} \max\{0,\, r[k,j]\}, & i \neq j, \\ \sum\limits_{k\neq i} \max\{0,\, r[k,j]\}, & i = j. \end{cases} \tag{2}$$

Once both responsibility and availability have been computed, their values are iteratively refined as follows to achieve convergence:

$$r[i,j] \;=\; (1-\lambda)\, r[i,j] \;+\; \lambda\, r'[i,j] \tag{3}$$

$$a[i,j] \;=\; (1-\lambda)\, a[i,j] \;+\; \lambda\, a'[i,j] \tag{4}$$

In this formulation, the damping factor λ is introduced to mitigate numerical oscillations. Here, $r'(i,j)$ and $a'(i,j)$ denote the responsibility and availability values from the previous iteration, respectively. It is recommended that λ be chosen such that $0.5 \leq \lambda < 1$. Although a higher λ may help reduce oscillatory behavior, it is not a foolproof solution and may also slow down the execution of the Affinity Propagation algorithm. Once convergence is achieved, the set of exemplars K is selected by [14]:

$$K = \underset{j\in\{1,2,\ldots,N\}}{\arg\max} \;\{r(i,j) + a(i,j)\}\,. \tag{5}$$

Space Complexity: The algorithm stores several $N \times N$ matrices (similarity matrix S, responsibility matrix R, and availability matrix A) and some additional arrays for exemplars and assignments. Thus, the overall space complexity is

$$O(N^2).$$

Time Complexity: Let N be the number of data points and T the number of iterations until convergence.

Responsibility Update: For each point n and each potential exemplar k, the update

$$r(n,k) \leftarrow s(n,k) - \max_{l\neq k}\{a(n,l) + s(n,l)\}$$

requires $O(N)$ work for the inner maximum. With $O(N^2)$ such pairs, this step costs

$$O(N^3)$$

Algorithm 8 Affinity Propagation Algorithm

```
1: **Input:** Similarity matrix $S$
2: **Output:** Exemplars and cluster assignments
3: Initialize $S \leftarrow -D$ (or heuristics), $R \leftarrow 0$, $A \leftarrow 0$
4: **repeat**
5:   **for** each point $n$ **do**
6:     **for** each potential exemplar $k$ **do**
7:       $r(n,k) \leftarrow s(n,k) - \max_{l\neq k}(a(n,l) + s(n,l))$
8:     **end for**
9:   **end for**
10:  **for** each potential exemplar $k$ **do**
11:    $a(k,k) \leftarrow \sum_{n\neq k} \max(0, r(n,k))$
12:    **for** each point $n \neq k$ **do**
13:      $a(n,k) \leftarrow \min\left(0, r(k,k) + \sum_{m\neq n,k} \max(0, r(m,k))\right)$
14:    **end for**
15:  **end for**
16:  **Update termination condition**
17:  **for** each potential exemplar $k$ **do**
18:    **if** $r(k,k) + a(k,k) \geq 0$ **then**
19:      $\mathcal{K} \leftarrow \mathcal{K} \cup \{k\}$
20:      $h(k) \leftarrow k$
21:    **end if**
22:  **end for**
23:  **for** each point $n \notin \mathcal{K}$ **do**
24:    $h(n) \leftarrow \arg\max_{k\in\mathcal{K}} s(n,k)$
25:  **end for**
26: **until** termination condition is met
27: **Return** exemplars $\mathcal{K}$ and assignments $h(n)$
```

per iteration.

Availability Update: This update has two parts:

- **Self-availability:** For each exemplar k,

$$a(k,k) \leftarrow \sum_{n\neq k} \max(0, r(n,k))$$

costs $O(N)$ per exemplar, totaling $O(N^2)$.

- **Non-self availability:** For each point $n \neq k$ and exemplar k, the update

$$a(n,k) \leftarrow \min\left(0,\, r(k,k) + \sum_{m\neq n,k} \max(0, r(m,k))\right)$$

involves a summation over $O(N)$ elements, leading to $O(N^3)$ per iteration.

Assignment Steps: Exemplar identification and cluster assignments require at most $O(N^2)$ per iteration.

Thus, each iteration is dominated by the $O(N^3)$ operations, and over T iterations the total time complexity is

$$O(T \cdot N^3).$$

b) Mean-Shift clustering algorithm: Mean-shift clustering begins with a set of multivariate, continuous data points $\{x_n\}_{n=1}^{N} \subset \mathbb{R}^D$ and relies on a kernel density estimate to capture the underlying probability density function of the data. The density at a point x is estimated using a kernel function K and a bandwidth parameter σ, which controls the smoothness of the estimate. A larger σ produces a smoother density function, while a smaller σ allows for finer detail. The density is given by:

$$p(x) \;=\; \frac{1}{N}\sum_{n=1}^{N} K\Big(\frac{\|x-x_n\|^2}{\sigma^2}\Big), \tag{5}$$

where K might be a Gaussian kernel, for instance, making the density estimation sensitive to local data concentrations.

Once the density is estimated, the algorithm iteratively shifts each point towards regions of higher density. This is achieved by updating each point using a function $f(x)$, which computes a weighted average of the nearby data points. The iterative update is written as:

$$x^{(\tau+1)} \;=\; f\big(x^{(\tau)}\big), \tag{6}$$

where the current point $x^{(\tau)}$ is moved in the direction that increases the density. The weighting ensures that points closer to $x^{(\tau)}$ have a larger influence on the update, driving the point toward a nearby mode of the density function.

For the commonly used Gaussian kernel, the update function simplifies elegantly. The function $f(x)$ becomes a weighted sum of all data points, where the weights correspond to the posterior probability $p(n|x)$ that a point x_n contributes to the density at x. This is expressed as:

$$f(x) \;=\; \sum_{n=1}^{N} p(n \mid x)\, x_n, \tag{7}$$

with

$$p(n|x) \;=\; \frac{\exp\Big(-\frac{\|x-x_n\|^2}{2\,\sigma^2}\Big)}{\sum_{n'=1}^{N}\exp\Big(-\frac{\|x-x_{n'}\|^2}{2\,\sigma^2}\Big)}. \tag{8}$$

This formulation highlights that each update step effectively moves x toward the center of mass of its neighborhood, where nearby points contribute more due to the exponential decay in distance.

After several iterations, each data point converges to a stable position, typically at a local maximum (mode) of the density function. In mean-shift clustering, these modes serve as the cluster centers, and each data point is assigned to the cluster corresponding to the mode it converged to. In practice, the iterative updates are terminated when the change in position falls below a predetermined tolerance, ensuring that the algorithm does not run indefinitely. A postprocessing step is often necessary to merge convergence points that are very close together, which helps to eliminate numerical artifacts and results in a more robust identification of clusters. By carefully selecting the bandwidth σ, the algorithm can adapt to various data distributions and successfully identify clusters of arbitrary shapes.

Mean-shift clustering processes a dataset $\{x_n\}_{n=1}^{N} \subset \mathbb{R}^d$ using an iterative update scheme. In each iteration, for a given point x, the algorithm computes weights and updates x by evaluating the influence of every other point in the dataset. For each data point, if we denote by T the number of iterations required until convergence, then in every iteration the following operations occur [16]:

Algorithm 9 Mean-Shift (MS) Algorithm

```
1:  Input:
      • Dataset $X = \{x_1, x_2, \ldots, x_N\}$ of $N$ points in $\mathbb{R}^d$
      • Bandwidth parameter $\sigma$
      • Convergence threshold $\epsilon$
2:  Output:
      • Cluster assignments for data points
3:  for each point $x_n \in X$ do
4:      Initialize $\mathbf{x} \leftarrow x_n$
5:      repeat
6:          Compute the probability weights:
              $p(n \mid \mathbf{x}) \leftarrow \frac{\exp\left(-\frac{1}{2}\|\mathbf{x}-x_n\|^2/\sigma^2\right)}{\sum_{n'=1}^{N}\exp\left(-\frac{1}{2}\|\mathbf{x}-x_{n'}\|^2/\sigma^2\right)}$
7:          Update the mean-shift vector:
              $\mathbf{x} \leftarrow \sum_{n=1}^{N} p(n \mid \mathbf{x}) x_n$
8:      until convergence (i.e., $\|\mathbf{x} - z_n\| < \epsilon$)
9:      Assign mode: $z_n \leftarrow \mathbf{x}$
10: end for
11: Apply connected-components algorithm to $\{z_n\}_{n=1}^{N}$ with
    threshold $\epsilon$ to form clusters
12: Return: Cluster assignments
```

$$\text{Compute weights: } p(n|x) = \frac{\exp\Big(-\frac{\|x-x_n\|^2}{2\sigma^2}\Big)}{\sum_{n'=1}^{N}\exp\Big(-\frac{\|x-x_{n'}\|^2}{2\sigma^2}\Big)}, \tag{9}$$

which requires calculating the distance $\|x-x_n\|$ for each of the N points (each distance computation costing $\mathcal{O}(d)$ operations). Thus, a single iteration for one point costs $\mathcal{O}(N\cdot d)$ operations.

Subsequently, the algorithm updates the current point as

$$x^{(\tau+1)} = \sum_{n=1}^{N} p(n|x^{(\tau)})\, x_n, \tag{10}$$

which is another weighted sum over N points, again requiring $\mathcal{O}(N \cdot d)$ operations.

Since these two operations are performed in each of the T iterations for each of the N data points, the overall time complexity of the iterative part is

$$\mathcal{O}(T \cdot N \cdot (N \cdot d)) = \mathcal{O}(T \cdot N^2 \cdot d). \tag{11}$$

In many practical scenarios, the number of iterations T is small and may be treated as a constant, resulting in a quadratic complexity $\mathcal{O}(N^2 \cdot d)$. Additionally, after convergence, a connected-components algorithm is applied to merge nearby modes into unique clusters. This step typically operates in

$\mathcal{O}(N \log N)$ or $\mathcal{O}(N)$ time, and thus does not dominate the overall computational cost.

Regarding space complexity, the algorithm stores the dataset of N points in $\mathbb{R}^d$, which requires

$$\mathcal{O}(N \cdot d) \tag{12}$$

space. During the iterative updates, temporary storage for the weight vector (of length N) and the updated point x (of length d) is needed. Hence, the additional memory requirement is $\mathcal{O}(N + d)$, which is dominated by the dataset storage. Therefore, the overall space complexity is

$$\mathcal{O}(N \cdot d). \tag{13}$$

E. Hierarchical based

a) BIRCH (Balance Iterative Reducing and Clustering using Hierarchies): BIRCH is a hierarchical clustering algorithm designed to handle very large datasets efficiently by building an incremental tree structure called the *CF Tree* [12]. Each leaf node of the CF Tree stores *Clustering Features (CF)*, which summarize local subclusters using three statistics:

- **N**: The number of data points in the subcluster.
- **LS**: The linear sum of the data points.
- **SS**: The sum of squares of the data points.

BIRCH has two main parameters: the **Branching factor** (B), which controls the maximum number of children per node, and the **Threshold** (T), which limits the maximum radius or diameter of subclusters in the leaf nodes.

The algorithm inserts each incoming data point into the closest subcluster in the CF Tree. If adding the point causes the subcluster to exceed the threshold, the leaf node may split to maintain compact subclusters. After building the CF Tree, a final clustering step (e.g., a simplified agglomerative merge or k-means on the leaf entries) is often applied to produce the desired number of clusters. This approach is both *incremental* and *memory-efficient*, making BIRCH particularly suitable for large-scale datasets.

The detailed complexity analysis of BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) consists of the following main operations:

1) **CF Tree Construction:** BIRCH incrementally inserts each data point into a CF Tree by traversing from the root to the appropriate leaf node. At each step, it compares the point with subclusters to find the closest one.
   - *Distance Calculation:* Each comparison between a point and a subcluster's centroid takes $\mathcal{O}(d)$ time, where d is the number of dimensions.
   - *Traversal Cost:* The tree height is usually small due to the branching factor (B). In many practical cases, insertion is close to $\mathcal{O}(d)$ or $\mathcal{O}(\log(n) \cdot d)$ per point, leading to a total of $\mathcal{O}(n \cdot d)$ or $\mathcal{O}(n \log(n) \cdot d)$ over n points.
2) **Node Splitting:** If adding a point causes a subcluster to exceed the threshold T (controlling maximum subcluster diameter), the leaf node may split, which can propagate upward if the branching factor is exceeded. This split is typically infrequent if T is well-chosen, but in the worst case (if T is too small), repeated splits can degrade performance to $\mathcal{O}(n^2 \cdot d)$.
3) **Optional Global Clustering:** After building the CF Tree, BIRCH often applies a final clustering step (e.g., k-means) on the leaf subclusters. If m is the number of leaf entries (where $m \ll n$), this step is typically $\mathcal{O}(\ell \cdot m \cdot k \cdot d)$ (for k-means), which is much smaller compared to clustering all n points directly.
4) **Total Time Complexity:** In typical cases, the CF Tree construction and maintenance yields about $\mathcal{O}(n \cdot d)$ or $\mathcal{O}(n \log(n) \cdot d)$ time. However, in the worst case, many

Algorithm 10 BIRCH Algorithm

```
1: Input:
     • Dataset $D = \{x_1, x_2, \ldots, x_n\}$ of $n$ points in $d$ dimensions
     • Branching factor $B$ (max children per node)
     • Threshold $T$ (maximum radius or diameter of leaf sub-clusters)
     • (Optional) Desired number of clusters $L$ for a final clustering phase
2: Output:
     • CF-Tree (Clustering Feature Tree)
     • (Optional) Final set of clusters if the final clustering step is performed
3: Initialize an empty CF-Tree with the root as a leaf node
4: for each point $x_i \in D$ do
5:   Insert $x_i$ into the CF-Tree:
6:     Start from the root node
7:     If the current node is not a leaf:
8:       Find the child sub-cluster whose centroid is closest to $x_i$ (using $(N, LS, SS)$)
9:       Descend into that child node
10:    Else (the current node is a leaf):
11:      Find the leaf-entry $L_j$ whose centroid is closest to $x_i$
12:      If adding $x_i$ to $L_j$ keeps the sub-cluster diameter $\leq T$:
13:        Update the CF of $L_j$ by incrementing $N$, adding $x_i$ to $LS$, and adding $x_i^2$ to $SS$
14:      Else:
15:        Split $L_j$ into two sub-clusters according to BIRCH splitting criteria
16:        If the leaf node exceeds branching factor $B$:
17:          Split the leaf node
18:        If a non-leaf node splits and exceeds $B$:
19:          Recursively split internal nodes up to the root if necessary
20: end for
21: (Optional) Condense or refine the CF-Tree (remove outliers or merge small sub-clusters)
22: (Optional) Final clustering phase: apply a standard clustering algorithm (e.g., $k$-means) on the leaf sub-cluster centroids if desired
23: Return: The final CF-Tree (and optionally the final clusters)
```

splits can lead to $\mathcal{O}(n^2 \cdot d)$.

5) **Total Space Complexity:** Storing n data points with d features requires $\mathcal{O}(n \cdot d)$ space. The CF Tree itself is usually much smaller than n, but in the worst case (if the threshold T is not effective in merging points), it can grow and approach $\mathcal{O}(n)$ leaf entries. Overall, BIRCH commonly requires $\mathcal{O}(n \cdot d)$ space in typical scenarios.

Thus, the overall time complexity for the BIRCH algorithm is typically $\mathcal{O}(n \cdot d)$ to $\mathcal{O}(n \log n \cdot d)$ (worst-case $\mathcal{O}(n^2 \cdot d)$) and the space complexity is approximately $\mathcal{O}(n \cdot d)$.

b) Agglomerative Hierarchy clustering algorithm: Agglomerative Hierarchical Clustering starts with each data point as its own cluster and *iteratively merges* the two most similar clusters until a stopping criterion is met (for example, until a single cluster remains or until a predefined number of clusters is reached) [13]. The main input parameters are:

- A **distance metric** (e.g., Euclidean).
- A **linkage criterion**, which determines how to compute distances between clusters (common variants are single, complete, average, or Ward's linkage).

During each iteration, the pair of clusters with the smallest distance (based on the chosen linkage) is merged into a single cluster. This process yields a *dendrogram*, a tree-like structure that captures the hierarchy of merges. Cutting the dendrogram at a certain level produces the desired number of clusters. Agglomerative methods can produce more interpretable results (via the dendrogram) but can be computationally expensive for large datasets due to repeated distance computations between clusters.

The detailed complexity analysis of Agglomerative Hierarchical Clustering consists of the following main operations:

1) **Distance Matrix Construction:** In the standard naive approach, we compute and store the pairwise distances between all n data points in a matrix of size $n \times n$.
   - *Distance Calculation:* Each pairwise comparison is $\mathcal{O}(d)$, and there are $\frac{n(n-1)}{2}$ pairs, leading to $\mathcal{O}(n^2 \cdot d)$.
   - *Matrix Storage:* The distance matrix itself requires $\mathcal{O}(n^2)$ space.
2) **Merging Clusters:** The algorithm starts with n singleton clusters. At each iteration, it merges the two closest clusters according to a chosen linkage criterion (e.g., single, complete, average, or Ward's).
   - *Finding Closest Clusters:* Naively scanning the entire distance matrix can be $\mathcal{O}(n^2)$ each iteration.
   - *Updating the Distance Matrix:* After a merge, we remove the rows/columns corresponding to the merged clusters and add/update a row/column for the new merged cluster. This also requires up to $\mathcal{O}(n)$ operations each time.

   Since we perform $n-1$ merges to go from n clusters down to 1, the repeated scans lead to high complexity.
3) **Total Time Complexity:** In the naive implementation, finding and merging pairs over $n-1$ iterations is $\mathcal{O}(n^3)$, since each merge step can be $\mathcal{O}(n^2)$ and we have roughly n merges. Building the distance matrix adds $\mathcal{O}(n^2 \cdot d)$, but $\mathcal{O}(n^3)$ dominates for large n.

Algorithm 11 Agglomerative Hierarchical Clustering

```
1:  Input:
      • Dataset $D = \{x_1, x_2, \ldots, x_n\}$
      • Distance function $dist(x_i, x_j)$ (e.g., Euclidean)
      • Linkage criterion (single, complete, average, Ward)
      • (Optional) Desired number of clusters $K$
2:  Output:
      • A dendrogram representing the merge history
      • (Optional) Final set of clusters if $K$ is specified
3:  Initialize each point $x_i \in D$ as its own cluster; set $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$
4:  Construct a distance matrix $Dist$ where $Dist[i,j] = dist(x_i, x_j)$
5:  while the number of clusters in $\mathcal{C}$ is greater than $K$ (or until one cluster remains if $K$ is not specified) do
6:      Find the pair of clusters $(C_p, C_q)$ in $\mathcal{C}$ with the smallest distance based on the linkage criterion
7:      Merge $C_p$ and $C_q$ into a new cluster $C_{\text{new}}$
8:      Update $\mathcal{C}$: remove $C_p$ and $C_q$, then add $C_{\text{new}}$
9:      Update the distance matrix $Dist$:
10:         Remove rows and columns corresponding to $C_p$ and $C_q$
11:         Compute distances from $C_{\text{new}}$ to all other clusters in $\mathcal{C}$ based on the chosen linkage method
12:     Record the merge of $(C_p, C_q)$ in the dendrogram
13: end while
14: Return: The dendrogram (and the final clusters if $K$ is specified)
```

4) **Total Space Complexity:** The algorithm requires $\mathcal{O}(n^2)$ space to store the distance matrix, which is typically the limiting factor. Additional bookkeeping for cluster memberships or dendrogram storage is $\mathcal{O}(n)$, but is negligible compared to the matrix.

Thus, the overall time complexity for the naive Agglomerative Hierarchical Clustering is $\mathcal{O}(n^3)$ and the space complexity is $\mathcal{O}(n^2)$, where n is the number of data points and d is the dimension for each data point.

IV. Metrics

A. Silhouette Score

The silhouette score for a data point i is given by:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))} \tag{14}$$

where:

- $a(i)$ is the average intra-cluster distance i.e., the mean distance from i to all other points in its own cluster.
- $b(i)$ is the average nearest-cluster distance i.e., the mean distance from i to all points in the nearest cluster.

$$silhouette = \frac{1}{N} \sum_{i}^{N} s(i) \tag{15}$$

The silhouette score for the entire dataset is the mean of individual silhouette scores [3].

Interpretation:

- $s(i) \approx 1 \rightarrow$ Well-clustered
- $s(i) \approx 0 \rightarrow$ Overlapping clusters
- $s(i) < 0 \rightarrow$ Incorrect clustering

Complexity Analysis:

1) Computing $a(i)$: For each data point, compute the average distance to all data points in its own cluster. If there are N points and the largest cluster has C points, it takes $\mathcal{O}(C)$ time.
2) Computing $b(i)$: For each data point, compute the average distance to points in the nearest cluster. In the worst case, we need to check all k clusters which takes $\mathcal{O}(kC)$.
3) For all N points, the worst case time complexity is $\mathcal{O}(NkC)$, if clusters are of equal size $C \approx \frac{N}{k}$, the complexity simplifies to $\mathcal{O}(N^2)$.

B. Davies-Bouldin Index

Defined as:

$$DB = \frac{1}{k}\sum_{i=1}^{k}\max_{j\neq i}\left(\frac{s_i + s_j}{d_{ij}}\right) \tag{16}$$

where

- k is the number of clusters.
- s_i is the intra-cluster dispersion i.e., the average distance of points in clusters i to its centroid.
- d_{ij} is the distance between cluster centroids i and j.

Interpretation:

- Lower values indicate better clustering.
- Measures cluster compactness (intra-cluster distance) and separation (inter-cluster distance) [4].

Complexity Analysis:

1) Computing s_i for each cluster takes $\mathcal{O}(N)$
2) Computing pairwise centroid distance d_{ij} takes $\mathcal{O}(k^2)$
3) Overall complexity is $\mathcal{O}(N + k^2)$
   - if $k \ll N$, the dominant term is $\mathcal{O}(N)$
   - if $k \approx N$, worst case complexity is $\mathcal{O}(N^2)$

C. Calinski-Harabasz Index

Defined as:

$$CH = \frac{\text{trace}(B_k)}{\text{trace}(W_k)} \times \frac{N-k}{k-1} \tag{17}$$

where

- N is the total number of data points.
- k is the number of clusters.
- B_k measures separation using between-cluster scatter matrix
- W_k measures compactness using within-cluster scatter matrix

Between-cluster Scatter Matrix(B_k) is defined as:

$$B_k = \sum_{i=1}^{k} n_i(c_i - c)(c_i - c)^T \tag{18}$$

where

- n_i is the number of points in cluster i.
- c_i is the centroid of cluster i.
- c is the global centroid of all data points.
- This matrix captures how well-separated the clusters are from each other.

Within-cluster Scatter Matrix(W_k) is defined as:

$$W_k = \sum_{i=1}^{k}\sum_{x\in C_i}(x - c)(x - c)^T \tag{19}$$

where

- C_i is the set of points in cluster i.
- x is an individual data point.
- c_i is the centroid of cluster i.
- This matrix captures how tightly the data points are grouped within each cluster.

Interpretation:

- A higher CH index indicates better that the clusters are well-separated and compact.
- A lower CH index suggests poor clusters, either because the clusters overlap significantly or the clusters are too spread out.
- Evaluates ratio of between-cluster dispersion to within-cluster dispersion [5].

Complexity Analysis:

1) Computing centroid takes $\mathcal{O}(N)$
2) Computing scatter matrix takes $\mathcal{O}(N)$
3) Overall time complexity is $\mathcal{O}(N)$ making this metric efficient for large datasets.

D. Adjusted Rand Index (ARI)

Defined as: Given two cluster assignments U (Ground truth clustering) and V (predicted clustering)

$$ARI = \frac{\sum_{ij}\binom{n_{ij}}{2} - \text{expected index}}{\text{max index} - \text{expected index}} \tag{20}$$

The expected index is defined as:

$$\text{expected index} = \frac{[\sum_i \binom{a_i}{2}\sum_j \binom{b_j}{2}]}{\binom{N}{2}} \tag{21}$$

The max index is defined as:

$$\text{max index} = \frac{1}{2}[\sum_i \binom{a_i}{2} + \sum_j \binom{b_i}{2}] \tag{22}$$

where

- N is the total number of data points.
- n_{ij} the contingency table that represents the number of data points that belong to cluster i in U and cluster j in V.
- $a_i = \sum_j n_{ij}$ is the sum over row i i.e., total points in cluster i of U
- $b_j = \sum_i n_{ij}$ is the sum over column j i.e., total points in cluster j of V
- $\binom{x}{2} = \frac{x(x-1)}{2}$ represents the number of ways to choose 2 items from x.

Interpretation:

- ARI = 1 means the clustering of ground truth and prediction are identical.
- ARI = 0 means the clustering similarity is no better than random.
- ARI < 0 means the clustering is worse than random.
- Unlike raw Rand Index, ARI is corrected for chance making it more reliable when cluster sizes differ [6].

Complexity Analysis:

1) Computing contingency table(n_{ij}) takes $\mathcal{O}(N)$ time.
2) Computing row and column sums (ai, bj) takes $\mathcal{O}(k)$ time, where k is the number of clusters.
3) Computing binomial coefficients takes $\mathcal{O}(k)$ time.
4) Overall time complexity is $\mathcal{O}(N)$.

E. Mutual Information (MI)

Defined as: Given two clustering Ground truth (U) and Predicted Clustering (V)

$$MI(U, V) = \sum_i \sum_j P(i, j) \log \frac{P(i, j)}{P(i)P(j)} \tag{23}$$

where

- $P(i) = \frac{|C_i|}{N}$ is the probability of a randomly chosen point belonging to cluster i in U.
- $P(j) = \frac{|C_i|}{N}$ is the probability of a randomly chosen point belonging to cluster j in V.
- $P(i, j) = \frac{|C_i \cap C_j|}{N}$ is the joint probability of a randomly chosen point being i of U and cluster j of V.

Interpretation:

- If MI = 0 means the two clusterings are independent i.e., no information is shared.
- Higher MI means greater agreement between clusterings [7].

Complexity Analysis:

1) Constructing contingency table n_{ij} takes $\mathcal{O}(N)$ time.
2) Computing probabilities $P(i), P(j)$, and $P(i, j)$ takes $\mathcal{O}(k)$ where k is the number of clusters.
3) Computing entropy and MI sum means $\mathcal{O}(k^2)$ in the worst case.
4) Overall time complexity is $\mathcal{O}(N)$.

F. Split

Measures the number of times a single true cluster is fragmented across multiple predicted clusters. [8].

Defined as Given ground truth partitions U with clusters $\{C_1, C_2, ..., C_k\}$ and a predicted clustering V with clusters $\{D_1, D_2, ..., D_k\}$

$$S(U, V) = \sum_{i=1} k(\text{number of parts } C_i \text{ is split into in } V\text{- } 1) \tag{24}$$

Interpretation:

- Measures fragmentation of a cluster.
- Lower values are better (less fragmentation).
- $S(U, V) = 0$ is the ideal case where each cluster in U remains intact in V.
- Higher split scores relate to worst clustering performance (high fragmentation).

Complexity Analysis:

1) Assigning ground truth labels takes $\mathcal{O}(N)$ time
2) Counting occurrences takes $\mathcal{O}(k)$ time.
3) Overall time complexity is $\mathcal{O}(N)$.

G. Diameter

Defined as:

$$D = \max_{i,j \in C} d(i, j) \tag{25}$$

where $d(i, j)$ is the distance between two points in the same cluster.

Interpretation:

- Measures cluster compactness [9].
- Lower diameter meter tighter clusters.

Complexity Analysis:

1) Compute pairwise distances within each cluster in worst case take $\mathcal{O}(C^2)$ per cluster. If clusters are of size $\frac{N}{k}$ then it take $\mathcal{O}(N^2/k)$
2) Overall time complexity is $\mathcal{O}(N^2/k)$, if it is a one big cluster (worst case) then it is $\mathcal{O}(N^2)$ and if it is a many small clusters(best case) then it is $\mathcal{O}(N)$

V. K-MEANS ACCURACY

A. Why the Silhouette Score Does Not Assess Accuracy

The silhouette score measures the quality of clustering by evaluating how well each point is assigned to its cluster(cohesion) relative to other clusters(separation) but it does not guarantee the clusters match the true distribution.

- **High Silhouette Score in Suboptimal Clustering:** An incorrect clustering may still achieve a high silhouette score if the groups are well-separated, even though they don't represent the true clusters.
- **K-Means Objective vs. Silhouette Score:** K-means minimizes intra-cluster variance, while the silhouette score aims to balance cohesion and separation. This means the optimal clustering for k-means may not necessarily maximize the silhouette score.

B. Show that it is possible to obtain an exact k-means clustering and a bad silhouette score.

K-means can achieve an exact clustering by minimizing variance. However, close clusters, ambiguous assignments, and non-spherical cluster shapes with varying densities can lead to a low silhouette score. The following points explain why this occurs:

- **Clusters Are Not Well-Separated but Still Minimize Variance:** If the clusters are close to each other, the nearest cluster distance $b(i)$ will be small, leading to a low silhouette score. K-means minimize variance, but it does not necessarily ensure well-separated clusters.
- **Ambiguous Assignments:** When some points are equally distant from multiple cluster centers, this leads to ambiguous- ous assignments. This increases $b(i)$ (the

nearest-cluster distance) and thus reduces the silhouette score.

- **Clusters Are Elongated or Have Varying Densities:**K-means assuming spherical clusters with similar densities. If the optimal clusters have different densities or are elongated, the silhouette score can be misleading, even if k-means provides the correct clustering in terms of minimizing intra-cluster variance.

[23]

C. Provide an example (with a small number of points, say around 10).

Consider the following 10 points in 2D space:
$P = \{(1,1),(2,1),(1,2),(2,2),(8,8),(9,8),$
$(8,9),(9,9),(5,5),(5,6)\}$

We perform an exact k-means clustering with $k = 3$, which minimizes the intra-cluster variance. The resulting clusters are:

- **Cluster 1:** {(1,1), (2,1), (1,2), (2,2)}
- **Cluster 2:** {(8,8), (9,8), (8,9), (9,9)}
- **Cluster 3:** {(5,5), (5,6)}

This is an exact k-means solution because it minimizes intra-cluster variance.

Now, consider the silhouette score. The third cluster, {(5,5), (5,6)}, is close to both the other clusters. Since the silhouette score is based on the separation of clusters, the points in the third cluster will have a small nearest-cluster distance $b(i)$, leading to a low silhouette score. Despite this, the clustering is still optimal in terms of minimizing variance.

Thus, this example shows that the silhouette score does not necessarily measure the precision of the clustering.

VI. k-means Improvement

A. Show on a small example, how the triangle inequality idea of Elkan works.

Elkan's k-means algorithm optimizes standard clustering of k-means using the triangle inequality to reduce distance computations. Given a set of data points $X = \{x_1, x_2, ..., x_n\}$ and cluster centroids $C = \{c_1, c_2, ..., c_k\}$, Elkan's method maintains upper and lower bounds:

$$d(x_i, c_j) \geq d(c_a, c_b) - d(x_i, c_a) \tag{26}$$

where $d(x_i, c_j)$ represents the Euclidean distance between a point x_i and a centroid c_j, and $d(c_a, c_b)$ is the distance between centroids. This avoids redundant distance calculations.

Consider the following 10 data points in a 2D plane:

$$P_1(2,3), P_2(3,4), P_3(5,6), P_4(8,8), P_5(9,10),$$

$$P_6(12,14), P_7(14,15), P_8(16,18), P_9(18,20), P_{10}(20,22)$$

Initial Cluster Centroids (k = 2, Randomly Chosen:

- $C_1 = (3,4)$
- $C_2 = (14,15)$

Using Triangle Inequality to Reduce Distance Computations:

1. Compute the **distance between centroids**:

$$d(C_1, C_2) = \sqrt{(14-3)^2 + (15-4)^2} \approx 15.56 \tag{27}$$

2. For a data point $P_3(5,6)$, check its **current lower bound to** C_1:

$$d(P_3, C_1) = \sqrt{(5-3)^2 + (6-4)^2} \approx 2.83 \tag{28}$$

3. Instead of computing $d(P_3, C_2)$, use the **triangle inequality**:

$$d(P_3, C_2) \geq d(C_1, C_2) - d(P_3, C_1) \tag{29}$$

$$d(P_3, C_2) \geq 15.56 - 2.83 = 12.73 \tag{30}$$

If 12.73 is greater than the already known minimum distance, we **skip the calculation** for $d(P_3, C_2)$.

By applying this method to all points, the algorithm avoids unnecessary distance computations, significantly improving efficiency.

B. Compute the complexity of Elkan's algorithm

The standard k-means algorithm has a complexity of $O(nkd)$ per iteration, where n is the number of data points, k is the number of clusters, and d is the number of dimensions.

Elkan's method improves this by reducing the number of distance calculations using mathematical bounds. This makes it run in about $O(nk + kd^2)$ per iteration, which is faster when d is small.

C. What is the impact of the triangle inequality tests on the complexity of the resulting k-means algorithm for an equal number of iterations?

Elkan's algorithm speeds up convergence by reducing the number of direct distance calculations while keeping the same number of iterations. The improvement in complexity is most noticeable when k is large because fewer distances need to be computed. This makes the algorithm overall more efficient.

[24], [25]

VII. Metrics appropriateness

A. Density-Based Clustering

1) DBSCAN:

- **Best Metric:** Adjusted Rand Index (ARI)
Reasoning: DBSCAN is designed to discover clusters of arbitrary shape and to identify noise without requiring a pre-specified number of clusters. ARI is particularly suitable when ground truth labels are available, as it robustly measures how closely the discovered clusters match the expected groups while accounting for chance groupings [6].
- **Weak Metric:** Mutual Information (MI)
Reasoning: Although MI measures the amount of shared information between the predicted clusters and the true labels, it does not penalize random agreements as effectively as ARI. Thus, MI may be less intuitive when DBSCAN identifies many noise points or produces clusters with highly irregular shapes [28].

2) OPTICS:

- **Best Metric:** Silhouette Score
  Reasoning: OPTICS produces an ordering (reachability plot) from which clusters are extracted. The Silhouette Score effectively measures how well-separated and cohesive the resulting clusters are, even when the clustering structure is derived from a variable density threshold [3].
- **Weak Metric:** Davies-Bouldin Index (DB)
  Reasoning: While DB quantifies the ratio of within-cluster scatter to between-cluster separation, it may be misleading for OPTICS since the algorithm is designed to handle clusters of varying density. If the density thresholds yield overlapping or "mushy" clusters, DB may indicate poor separation even if the density structure is appropriate [4].

B. Distribution-Based Clustering

1) Gaussian Mixture Model (GMM):

- **Best Metric:** Calinski-Harabasz Index (CH)
  Reasoning: GMM models data as a mixture of Gaussian distributions, aiming for clusters with low within-cluster scatter and high between-cluster separation. The CH index, which measures the variance ratio, directly reflects this objective by rewarding well-separated, cohesive clusters [5].
- **Weak Metric:** Mutual Information (MI)
  Reasoning: MI can capture the information overlap between the predicted clusters and true labels. However, GMM's soft assignments may lead to overlapping clusters, and MI—if computed using hard assignments—might understate the nuanced membership, making it a less ideal measure of clustering quality in this context [28].

C. Centroid/Medoid-Based Clustering

1) k-Means:

- **Best Metric:** Davies-Bouldin Index (DB)
  Reasoning: k-Means minimizes the sum of squared distances to cluster centroids, aiming for compact, spherical clusters. The DB index evaluates the average ratio of intra-cluster scatter to inter-cluster separation, aligning closely with k-Means' objective [4].
- **Weak Metric:** Silhouette Score
  Reasoning: Although the Silhouette Score measures how well-separated clusters are, it may not fully capture the assumption of spherical clusters inherent in k-Means. When clusters are non-convex or unevenly sized, the Silhouette Score might not reflect the optimality of k-Means results [3].

2) k-Medoids:

- **Best Metric:** Davies-Bouldin Index (DB)
  Reasoning: k-Medoids minimizes the sum of distances to representative data points (medoids) and is robust to outliers. The DB index, by evaluating the ratio of within-cluster dispersion to between-cluster separation, effectively measures the quality of clustering achieved under arbitrary distance metrics [4].
- **Weak Metric:** Diameter
  Reasoning: Diameter, defined as the maximum intra-cluster distance, can expose clusters that are overly stretched even if the overall dispersion is low. This metric highlights if the chosen medoid is not centrally located, resulting in clusters with a few distant outliers [27].

D. Exemplar-Based Clustering

1) Affinity Propagation:

- **Best Metric:** Silhouette Score
  Reasoning: Affinity Propagation selects exemplars via message passing, resulting in clusters that may vary in size and compactness. The Silhouette Score provides an effective measure of how well each point fits within its cluster relative to others, indicating the overall quality of the exemplar-based clustering structure [3].
- **Weak Metric:** Davies-Bouldin Index (DB)
  Reasoning: Although DB evaluates cluster separation, Affinity Propagation can generate a large number of clusters if the preference parameter is high, potentially leading to overlap. A high DB value would signal such issues, but DB might not capture the nuances of exemplar selection as directly as the Silhouette Score [4].

2) Mean-Shift:

- **Best Metric:** Silhouette Score
  Reasoning: Mean-Shift clustering identifies modes in the data density without pre-specifying the number of clusters. The Silhouette Score effectively captures the distinctness of the density modes, indicating if clusters are both well-formed and well-separated [3].
- **Weak Metric:** Diameter
  Reasoning: Mean-Shift can produce clusters with large spatial extents if the bandwidth parameter is too large. Diameter is a straightforward measure to detect clusters that are overly stretched, which might suggest that multiple density peaks have been inappropriately merged [27].

E. Hierarchical-Based Clustering

1) BIRCH:

- **Best Metric:** Calinski-Harabasz Index (CH)
  Reasoning: BIRCH builds a CF-tree to form compact subclusters, implicitly minimizing within-cluster variance. The CH index measures the ratio of between-cluster dispersion to within-cluster dispersion, which closely aligns with BIRCH's objective of producing tight, well-separated clusters [5].
- **Weak Metric:** Silhouette Score
  Reasoning: Although BIRCH aims to create compact clusters, its reliance on a threshold can sometimes lead to merged clusters that are not well-separated. The Silhouette Score can help detect if the final clusters lack clear separation, thereby indicating potential over-condensation [3].

2) Agglomerative Hierarchical Clustering:

- **Best Metric:** Davies-Bouldin Index (DB)
  Reasoning: Agglomerative Hierarchical Clustering merges clusters based on a chosen linkage criterion, and when the dendrogram is cut at a given level, DB effectively reflects the balance between within-cluster compactness and between-cluster separation [4].
- **Weak Metric:** Split
  Reasoning: Hierarchical clustering can sometimes over-split natural clusters, fragmenting them across multiple branches. The Split metric specifically measures the degree to which a natural group is divided, thus highlighting if the clustering has fragmented cohesive groups beyond what is ideal [26].

VIII. Conclusion

Evaluating clustering accuracy requires understanding both the algorithm's goals and the data's characteristics. Selecting the appropriate evaluation metric is crucial, as different metrics focus on aspects like cohesion, separation, and alignment with known labels. For example, density-based algorithms prioritize cluster shape and noise resilience, while centroid-based methods focus on minimizing variance. Therefore, the metric chosen should align with the algorithm's objectives.

Some metrics work better with specific clustering methods. The Adjusted Rand Index (ARI) is effective for density-based algorithms like DBSCAN, while the Calinski-Harabasz (CH) Index suits Gaussian Mixture Models (GMM), evaluating the variance between and within clusters. K-Means, on the other hand, is best assessed using the Davies-Bouldin (DB) Index, which balances intra-cluster compactness and inter-cluster separation.

However, mismatched metrics can lead to misleading conclusions. The Silhouette Score, for example, may not work well with non-spherical clusters in K-Means. Similarly, Mutual Information (MI) can be ineffective for GMM because it doesn't account for its probabilistic nature. These issues highlight the importance of carefully selecting metrics that align with the algorithm.

In practice, a holistic approach is necessary, combining domain knowledge with algorithmic understanding. Using multiple evaluation metrics—like ARI or CH for ground truth validation and DB or Silhouette for internal cluster quality—ensures more reliable and accurate clustering results.

References

[1] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD)*, Portland, OR, USA, 1996, pp. 226–231.

[2] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "OPTICS: Ordering points to identify the clustering structure," in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, PA, USA, 1999, pp. 49–60.

[3] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis,"
J. Comput. Appl. Math., vol. 20, pp. 53-65, 1987.

[4] D. L. Davies and D. W. Bouldin, "A cluster separation measure,"
IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-1, no. 2, pp. 224-227, 1979.

[5] T. Calinski and J. Harabasz, "A dendrite method for cluster analysis,"
Communications in Statistics, vol. 3, no. 1, pp. 1-27, 1974.

[6] L. Hubert and P. Arabie, "Comparing partitions,"
Journal of Classification, vol. 2, no. 1, pp. 193-218, 1985.

[7] A. Strehl and J. Ghosh, "Cluster ensembles - A knowledge reuse framework for combining multiple partitions,"
J. Mach. Learn. Res., vol. 3, pp. 583-617, 2002.

[8] C. D. Manning, P. Raghavan, and H. Schütze, "Introduction to Information Retrieval,"
Cambridge University Press, 2008.

[9] A. K. Jain and R. C. Dubes, "Algorithms for Clustering Data,"
Prentice-Hall, 1988.

[10] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum Likelihood from Incomplete Data via the EM Algorithm," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 39, no. 1, pp. 1–22, 1977.

[11] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[12] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An efficient data clustering method for very large databases," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 103–114, 1996.

[13] G. N. Lance and W. T. Williams, "A general theory of classificatory sorting strategies: 1. Hierarchical systems," *The Computer Journal*, vol. 9, no. 4, pp. 373–380, 1967.

[14] R. Refianti, A. B. Mutiara, and S. Gunawan, "Time complexity comparison between affinity propagation algorithms,".

[15] B. J. Frey and D. Dueck, "Clustering by passing messages between data points," *Science*, vol. 315, no. 5814, pp. 972-976, 2007.

[16] M. A. Carreira-Perpiñán, "A review of mean-shift algorithms for clustering," *University of California, Merced*, March 2, 2015.

[17] P. Arora and S. Varshney, "Analysis of k-means and k-medoids algorithm for big data," *Procedia Computer Science*, vol. 78, 2016, pp. 507–512.

[18] K. P. Sinaga and M.-S. Yang, "Unsupervised K-means clustering algorithm," *IEEE Access*, vol. 8, pp. 80716–80727, 2020.

[19] S. Na, X. Liu, and Y. Guan, "Research on k-means clustering algorithm: An improved k-means clustering algorithm," in *Proceedings of the Third International Symposium on Intelligent Information Technology and Security Informatics*, 2010, pp. 63–67.

[20] R. Tibshirani, G. Walther, and T. Hastie, "K-Means clustering and related algorithms," Technical Report, Stanford University, 2004.

[21] D. Cao and B. Yang, "An improved k-medoids clustering algorithm," in *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, vol. 3, 2010, pp. 132–135.

[22] T. Wang, Q. Li, D. J. Bucci, Y. Liang, B. Chen, and P. K. Varshney, "K-medoids clustering of data sequences with composite distributions," *IEEE Transactions on Signal Processing*, vol. 67, no. 8, 2019, pp. 2093–2106.

[23] F. Wang, H.-H. Franco-Penya, J. D. Kelleher, J. Pugh, and R. Ross, "An analysis of the application of simplified silhouette to the evaluation of k-means clustering validity," in *Machine Learning and Data Mining in Pattern Recognition: 13th International Conference, MLDM 2017, New York, NY, USA, July 15-20, 2017, Proceedings 13*, 2017, pp. 291–305.

[24] C. Elkan, "Using the triangle inequality to accelerate k-means," in *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, 2003, pp. 147–153.

[25] A. O. Barajas, "K-Means clustering accelerated algorithms using the triangle inequality," 2015.

[26] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*, 2nd ed. Pearson, 2013.

[27] G. Gan, C. Ma, and J. Wu, *Data Clustering: Theory, Algorithms, and Applications*. SIAM, 2007

[28] N. X. Vinh, J. Epps, and J. Bailey, "Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance," *Journal of Machine Learning Research*, vol. 11, pp. 2837–2854, 2010.

Appendix

All team members worked together to understand the intuition behind the clustering algorithms and the various metrics available to measure performance. However, due to time constraints, we have split the work while creating this report in the following way which mention the section followed by the name of the team member.

- Introduction by Farid Faraji
- Computation of the dissimilarity coefficients by Farid Faraji
- Algorithm descriptions and their detailed complexity analysis
  - Density based by Nitheesh Kumar Kambala
  - Distribution based by Sotirios Damas
  - Centroid/medoid based by Divyesh Pravinkumar Patel
  - Exemplar based by Farid Faraji
  - Hierarchical based by Sotirios Damas
- Metrics by Nitheesh Kumar Kambala
- k-means Accuracy by Divyesh Pravinkumar Patel
- k-means Improvement by Divyesh Pravinkumar Patel
- Metrics appropriateness by Nitheesh Kumar Kambala and Sotirios Damas
- Conclusion by Farid Faraji