EX06

Textbook

- 10.2, 10.4, 10.6
- 11.2

Due 4/7

10.2

Suppose that the datamining task is to cluster points (with(x, y) representing location) into three clusters, where the points are
$A1$(2,10), $A2$(2,5), $A3$(8,4), $B1$(5,8), $B2$(7,5), $B3$(6,4), $C1$(1,2), $C2$(4,9).
The distance function is Euclidean distance. Suppose initially we assign $A1$, $B1$, and $C1$ as the center of each cluster, respectively. Use the *k-means* algorithm to show *only*

(a) The three cluster centers after the first round of execution.

```
# Zhenhao Zhang 4/4/24

from math import sqrt

# Initial cluster centers
cluster_centers = {
    'Cluster1': (2, 10),
    'Cluster2': (5, 8),
    'Cluster3': (1, 2)
}

# Points
points = {
    'A1': (2, 10),
    'A2': (2, 5),
    'A3': (8, 4),
    'B1': (5, 8),
    'B2': (7, 5),
    'B3': (6, 4),
    'C1': (1, 2),
    'C2': (4, 9)
}

# Function to calculate Euclidean distance
def euclidean_distance(p1, p2):
    return sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)
```

```
# Assignment step: Assign points to the nearest cluster
clusters = {'Cluster1': [], 'Cluster2': [], 'Cluster3': []}

for point_name, point_location in points.items():
    nearest_cluster = None
    min_distance = float('inf')
    for cluster_name, cluster_center in cluster_centers.items():
        distance = euclidean_distance(point_location, cluster_center)
        if distance < min_distance:
            nearest_cluster = cluster_name
            min_distance = distance
    clusters[nearest_cluster].append(point_location)

# Update step: Calculate new cluster centers
new_cluster_centers = {}

for cluster_name, cluster_points in clusters.items():
    if cluster_points: # To ensure division by zero does not happen
        new_center_x = sum(point[0] for point in cluster_points) / len(cluster_points)
        new_center_y = sum(point[1] for point in cluster_points) / len(cluster_points)
        new_cluster_centers[cluster_name] = (new_center_x, new_center_y)
    else:
        new_cluster_centers[cluster_name] = cluster_centers[cluster_name] # If no points, keep the
original center

print(new_cluster_centers)
```

The result is:

Cluster1: (2.0, 10.0)

Cluster2: (6.0, 6.0)

Cluster3: (1.5, 3.5)}

(b) The final three clusters.

```
from math import sqrt

# Correcting the points data structure to be a dictionary for the k-means clustering function
```

```
points = {
    'A1': (2, 10),
    'A2': (2, 5),
    'A3': (8, 4),
    'B1': (5, 8),
    'B2': (7, 5),
    'B3': (6, 4),
    'C1': (1, 2),
    'C2': (4, 9)
}


def euclidean_distance(p1, p2):
    return sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)


def k_means_clustering(points, initial_centers):
    """
    Perform k-means clustering until convergence.
    """
    centers = initial_centers.copy()
    cluster_assignments = None

    while True:
        new_cluster_assignments = {cluster: [] for cluster in centers}  # Initialize clusters

        # Assignment step
        for point_name, point_location in points.items():
            closest_center = min(centers, key=lambda center: euclidean_distance(point_location,
centers[center]))
            new_cluster_assignments[closest_center].append(point_name)

        # Check for convergence
        if new_cluster_assignments == cluster_assignments:
            break  # Clusters have stabilized

        cluster_assignments = new_cluster_assignments.copy()

        # Update step
        for cluster in centers:
            if cluster_assignments[cluster]:  # Ensure there are points assigned to the cluster
                assigned_points = [points[point_name] for point_name in cluster_assignments[cluster]]
                centers[cluster] = (
                    sum(point[0] for point in assigned_points) / len(assigned_points),
```

```
                        sum(point[1] for point in assigned_points) / len(assigned_points)
                    )

    return cluster_assignments, centers


# Initial cluster centers after the first round
initial_centers = {
    'Cluster1': (2.0, 10.0),
    'Cluster2': (6.0, 6.0),
    'Cluster3': (1.5, 3.5)
}

# Perform k-means clustering
final_clusters, final_centers = k_means_clustering(points, initial_centers)
print(final_clusters)
```

The output is:

Cluster 1: Contains points {A1, C2, B1}

Cluster 2: Contains points {A3, B2, B3}

Cluster 3: Contains points { C1, A2}

10.4
For the k-means algorithm, it is interesting to note that by choosing the initial cluster centers carefully, we may be able to not only speed up the algorithm's convergence, but also guarantee the quality of the final clustering. The k-means++ algorithm is a variant of k-means, which chooses the initial centers as follows. First, it selects one center uniformly at random from the objects in the data set. Iteratively, for each object p other than the chosen center, it chooses an object as the new center. This object is chosen at random with probability proportional to dist(p)^2, where dist(p) is the distance from p to the closest center that has already been chosen. The iteration continues until k centers are selected.
Explain why this method will not only speed up the convergence of the k-means algorithm, but also guarantee the quality of the final clustering results.

In their 2007 paper "K-means++: The advantages of careful seeding," D. Arthur and S. Vassilvitskii introduced the K-means++ algorithm as an enhancement. The k-means++ algorithm significantly improves the performance of the traditional k-means algorithm by optimizing the selection of initial cluster centers. This optimization specifically addresses the sensitivity of the k-means algorithm to the initial choice of centers and the potential for convergence to local optima that are not globally optimal. The key to k-means++ lies in its initialization step: it starts by randomly selecting a data point as the first cluster center, then for each remaining point, it selects the next center randomly with a probability proportional to the square of the distance to the nearest existing center. This method ensures that the initial centers are spread out, reducing the need for significant adjustments in the early stages and thereby speeding up convergence.

By choosing centers located in less dense areas of the dataset, k-means++ further optimizes the quality of clustering. This approach minimizes within-cluster variance, helping to avoid the algorithm getting stuck in local optima and making the final clustering more accurate. Additionally, the k-means++ algorithm demonstrates stronger adaptability and robustness when dealing with datasets of varying densities and shapes, maintaining efficiency and reliability across a wide range of application scenarios.

In summary, the improvements brought by the k-means++ algorithm not only reduce the number of iterations needed to reach a stable clustering state but also significantly enhance the overall quality of the clustering results by avoiding suboptimal solutions. This carefully designed initialization step addresses key weaknesses of the traditional k-means algorithm, providing a more reliable and effective clustering solution.

10.6
Both *k-means* and *k-medoids* algorithms can perform effective clustering.

(a) Illustrate the strength and weakness of *k-means* in comparison with *k-medoids*.

When comparing the k-means and k-medoids clustering algorithms, each has distinct advantages and disadvantages.

K-means is celebrated for its efficiency, especially with large datasets, owing to the computational simplicity of calculating means for updating cluster centers. This makes it not only faster but also easier to scale than k-medoids. It's particularly effective for datasets with roughly spherical clusters, as the algorithm optimally minimizes variance within clusters, leading to high-quality clustering in such scenarios. However, k-means has drawbacks, including a high sensitivity to outliers, since extreme values can significantly skew means. The algorithm's dependence on the initial choice of centroids can also lead to varying quality in the final clustering outcome, necessitating multiple runs for optimal results. Moreover, the assumption that clusters are spherical and of similar size may only hold true for some datasets, limiting its applicability. Lastly, k-means primarily utilizes Euclidean distance to measure similarities, which may only be suitable for some data types.

On the other hand, k-medoids present a robust alternative, especially in scenarios where data includes outliers or non-spherical clusters. Unlike k-means, k-medoids are less influenced by extreme values, making them more reliable for datasets with noise and outliers. It offers greater flexibility regarding the distance metrics it can accommodate, allowing for effective clustering of a wider variety of data types, including categorical and non-Euclidean data. This makes k-medoids more suitable for complex data structures where k-means might falter. However, this robustness comes at a computational cost; k-medoids are more computationally intensive, particularly with large datasets, as it requires exhaustive computation to identify the medoid of each cluster. This makes the algorithm less scalable and more complex to implement than k-means, potentially limiting its usability in large datasets.

In conclusion, K-means offers a fast and efficient solution for large, spherical clusters but may need help with outliers and complex cluster shapes. Conversely, k-medoids provide a more robust approach that accommodates a wider variety of data types and structures at the expense of computational efficiency and ease of implementation.

(b) Illustrate the strength and weakness of these schemes in comparison with a hierarchical clustering scheme (e.g., AGNES).

When comparing k-means/k-medoids clustering methods with hierarchical clustering schemes like AGNES, each exhibits unique advantages and disadvantages tailored to different data analysis scenarios. Hierarchical clustering, particularly AGNES, shines with its ability not to predetermine the number of clusters, offering a more flexible approach through the construction of a dendrogram. This feature facilitates the identification of cluster numbers post-analysis and reveals the hierarchical relationships within the data, providing deeper insights into the data structure. Moreover, hierarchical clustering's adaptability to various distance metrics and effectiveness in handling non-spherical clusters underscore its versatility across diverse data types. However, its significant computational complexity poses a challenge for large datasets, where the quadratic to cubic computational time becomes impractical. Additionally, the irreversible nature of its agglomerative steps can lead to permanent clustering decisions that might not be optimal, and its sensitivity to noise and outliers can complicate the interpretation of the dendrogram.

On the other hand, k-means and k-medoids present a more straightforward and computationally efficient solution for large datasets, with k-means, in particular, being known for their speed and simplicity. These methods demand a predefined number of clusters, which can be a drawback when such information is not readily available. K-means are notably susceptible to the initial placement of centroids, which can significantly influence the clustering outcome. It generally assumes spherical cluster shapes, making it less suited for datasets with complex geometries or significant outliers, where k-medoids offer a more robust alternative at a higher computational cost.

In essence, the selection between hierarchical clustering and k-means/k-medoids methods hinges on the dataset size, the known or unknown nature of the cluster numbers, sensitivity to outliers, and the computational resources at disposal. Hierarchical clustering is ideal for extracting detailed data structure and relationships in smaller datasets, while k-means and k-medoids are better suited for larger datasets requiring more straightforward, efficient clustering solutions.

11.2

AllElectronics carries 1000 products, $P1$, ..., $P1000$. Consider customers Ada, Bob, and Cathy such that Ada and Bob purchase three products in common, $P1$,$P2$, and $P3$. For the other 997 products, Ada and Bob independently purchase seven of them randomly. Cathy purchases 10 products, randomly selected from the 1000 products. In Euclidean distance, what is the probability that *dist* (Ada, Bob) > *dist* (Ada, Cathy)? What if Jaccard similarity (Chapter 2) is used? What can you learn from this example?

The probability for Ada and Bob is :

$$P_{A\text{-}B}(i) = \frac{\binom{997}{7}\binom{7}{i-3}\binom{990}{10-i}}{\binom{997}{7}^2} = \frac{\binom{7}{i-3}\binom{990}{10-i}}{\binom{997}{7}} \qquad 3 \le i \le 10$$

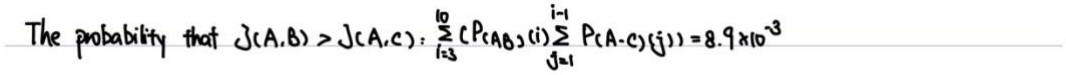
	dist(A, B)	J(A, B)	$P_{A\text{-}B}(i)$
$i=3$	3.7	0.18	0.95
$i=4$	3.5	0.25	6.8×10^{-3}
$i=5$	3.2	0.33	4.1×10^{-5}
$i=6$	2.8	0.43	2.1×10^{-7}
$i=7$	2.4	0.5	8.5×10^{-10}
$i=8$	2	0.67	2.6×10^{-12}
$i=9$	1.4	0.82	5.2×10^{-15}
$i=10$	0	1	5.3×10^{-18}

The probability for Ada and Cathy is :

$$P_{A\text{-}C}(j) = \frac{\binom{997}{7}\binom{10}{j}\binom{990}{10-j}}{\binom{997}{7}\binom{1000}{10}} = \frac{\binom{10}{j}\binom{990}{10-j}}{\binom{1000}{10}} \qquad 1 \le j \le 10$$

	dist(A, C)	J(A, C)	$P_{A\text{-}C}(j)$
$j=1$	4.2	0.052	9.2×10^{-3}
$j=2$	4	0.11	8.4×10^{-5}
$j=3$	3.7	0.18	6.9×10^{-9}
$j=4$	3.5	0.25	4.9×10^{-9}
$j=5$	3.2	0.33	3.0×10^{-11}
$j=6$	2.8	0.43	1.3×10^{-13}
$j=7$	2.4	0.5	6.1×10^{-16}
$j=8$	2	0.67	1.9×10^{-21}
$j=9$	1.4	0.82	3.8×10^{-21}
$j=10$	0	1	3.8×10^{-24}

Thus, the probability that dist(A, B) > dist(A, C): $\sum_{i=3}^{9}\left(P_{(A\text{-}B)}(i)\sum_{j=i+1}^{10}P_{(A\text{-}C)}(j)\right) = 4.7\times10^{-9}$



For Euclidean distance, the probability that the distance between Ada and Bob is greater than between Ada and Cathy is very low, at 4.7*10^-9. This means it's highly unlikely that Cathy's randomly chosen products would be closer to Ada's choices.

For the Jaccard similarity, the probability that the similarity between Ada and Bob is greater than that between Ada and Cathy is 8.9*10^-3. This relatively higher probability indicates that Cathy's random purchases can still be like Ada's.

This shows that the choice of metric can significantly impact the analysis. With many product options, the likelihood of randomness leading to similarity is low. However, the Jaccard similarity gives a relatively higher probability of similarity for random choices.