1

(a)

Step 1: Generate Candidate Sequences of Length 1 (C1)

Start by identifying all unique items across all transactions and form initial candidate sequences of length 1 (k=1), each containing a single item.

Step 2: Calculate Support and Filter to Get Frequent 1-Sequences (L1)

Compute the support for each candidate 1-sequence by counting the number of customers whose sequences are super-sequences of the candidate. Keep only those with a support of at least 2.

Step 3: Generate Candidate k-Sequences (Ck)

For k > 1, generate candidate k-sequences by joining frequent (k-1)-sequences with themselves, ensuring that the resulting sequence maintains a valid order and does not introduce duplicates unnecessarily.

Step 4: Calculate Support for Ck

Calculate the support for each candidate k-sequence by checking against the customer sequences to identify those that are super-sequences of the candidate.

Step 5: Filter to Get Frequent k-Sequences (Lk)

Retain only those candidate k-sequences with a support of at least 2.

Step 6: Repeat Steps 3-5 for Increasing k

Continue the process of generating candidates, calculating support, and filtering for increasing values of k until no more frequent k-sequences can be found.

In the given example.

L1: <{A}>, <{D}>, <{E}>, <{G}>

C2: <{A, D}>, <{A}, {D}>, <{A, E}>, <{A}, {E}>, <{A, G}>, <{A}, {G}>, <{D, E}>, <{D}, {E}>, <{D, G}>, <{D}, {G}>, <{E, G}>, <{E}, {G}>

L2: <{A}, {D}>, <{A}, {E}>, <{A, G}>, <{D, E}>,

C3: <{A}, {D}, {E}>, <{A, G}, {D}>, <{A}, {D, E}>, <{A, G}, {E}>

L3: <{A}, {D, E}>

(b)

Step 1: Initialize Candidate 1-Sequences (C1)

Start by identifying all unique items across all transactions to form initial candidate sequences of length 1 (1-itemsets).

Step 2: Calculate Support for C1 and Filter for L1

Compute the new support for each candidate 1-sequence. For each sequence, count all possible occurrences in all customers and divide by the number of customers that have the sequence as a subsequence. Retain those with a support of at least 2.

Step 3: Generate Candidate k-Sequences (Ck) for k > 1

For each level k, generate candidate k-sequences by joining the sequences from the previous level (Lk-1) while ensuring that the sequences respect the order of items. This process involves combining sequences that share common prefixes and ensuring that the resulting sequence does not violate the item order within any potential super-sequence.

Step 4: Calculate Support for Ck

For each candidate k-sequence, count all possible occurrences across all customers' sequences. This involves identifying all sets of indices that can map the candidate sequence into the customer sequences as defined. Then, compute the support as the total number of occurrences divided by the number of unique customers in which these occurrences are found.

Step 5: Filter to Get Frequent k-Sequences (Lk)

Retain those k-sequences with a computed support of at least 2.

Step 6: Repeat for Increasing k

Continue this process for increasing values of k until no more frequent k-sequences can be identified.

For the given example, there is no k-sequence which has a support of at least 2.

2(Use coupon on this question)

(a)

(i) Adaptation Steps:

1. **Generate the FP-tree:** Follow the standard procedure for FP-growth, which involves creating an FP-tree from the dataset, preserving the item order based on frequency.
2. **Modified FP-growth Mining:**
    - When recursively mining the FP-tree for itemsets, keep track of each itemset's support count.
    - Along with mining all itemsets, maintain a structure (like a dictionary or a heap) to keep track of the support counts of all itemsets mined.
3. **Identifying Closed Itemsets:**
    - After mining, you have a list of itemsets with their supports. A closed itemset is one for which no superset has the same support count.
    - For each itemset, check if any of its supersets have the same support count. If not, it is a closed itemset.
4. **Filtering by K-th Greatest Support:**
    - From the list of closed itemsets, create a multi-set of their support counts.
    - Find the K-th greatest support in this multi-set.

- Filter the closed itemsets, keeping only those with support greater than or equal to this K-th greatest support.

(ii) Illustration with K = 2:

Given transactions:

```
A B C D
1 0 0 1
0 1 0 0
1 0 1 1
1 0 1 1
```

1. **Generate FP-tree:**
    - Frequencies: A=3, D=3, C=2, B=1.
    - FP-tree is built with this order in mind.
2. **Mining and Identifying Closed Itemsets:**
    - Potential closed itemsets might include {A}, {D}, {C}, {A,D}, {A,C}, {D,C}, {A,D,C}, with their respective supports calculated from the FP-tree mining process.
3. **Finding K-th Greatest Support:**
    - For simplicity, let's say we identify {A}, {D}, and {A,D} as closed itemsets after the check for supersets with equal support.
    - The support counts are 3, 3, and 2, respectively. The second greatest support (K=2) is 2.
4. **Filtering:**
    - Closed itemsets with support ≥ 2 are {A}, {D}, and {A,D}.

(iii) Illustration with K = 3:

Following the same dataset and assuming a similar process, the third greatest support might not exist or be lower, depending on the actual closed itemsets found and their supports. If, for instance, the only supports are 3, 3, and 2, with 3 being the highest and 2 the next, there's no "third greatest" in a strict sense. We'd consider all itemsets with support ≥ 2, which would be the same as in the K=2 case for this example.

(b)

Advantages:

- **Reduced Set:** Closed itemsets provide a compact representation of frequent itemsets, potentially reducing the number to analyze.
- **No Information Loss:** They retain complete information about itemset frequencies, allowing full recovery of the original frequent itemsets.
- **Efficiency:** Can be more efficient in both storage and computational resources when dealing with large datasets.

Disadvantages:

- **Complexity:** Finding closed itemsets, especially with added constraints like the K-th greatest support, can be more complex than traditional frequent itemset mining.
- **Post-processing:** Requires additional steps to filter and verify the closedness of itemsets, adding to the computational overhead.

- **Understanding:** For some applications, interpreting closed itemsets and their significance can be less intuitive than dealing directly with all frequent itemsets.

This adaptation and comparison outline how FP-growth can be extended beyond its original purpose, leveraging its efficiency for more nuanced analyses in data mining.

3.

(a)

(i)

```
Cluster 1:
  Centroid: [49.28571429 49.57142857]
  Points: [[55, 50], [43, 50], [55, 52], [43, 54], [58, 53], [41, 47], [50, 41]]
Cluster 2:
  Centroid: [50. 70.]
  Points: [[50, 70]]
```

(ii)

```
Cluster 1:
  Centroid: [44.25 48.  ]
  Points: [[43, 50], [43, 54], [41, 47], [50, 41]]
Cluster 2:
  Centroid: [54.5  56.25]
  Points: [[55, 50], [55, 52], [58, 53], [50, 70]]
```

(iii)

```
Cluster 1:
  Centroid: [54.5 49. ]
  Points: [[55, 50], [55, 52], [58, 53], [50, 41]]
Cluster 2:
  Centroid: [50. 70.]
  Points: [[50, 70]]
Cluster 3:
  Centroid: [42.33333333 50.33333333]
  Points: [[43, 50], [43, 54], [41, 47]]
```

(iv)

```
Cluster 1:
  Centroid: [50. 41.]
  Points: [[50, 41]]
Cluster 2:
  Centroid: [50. 70.]
  Points: [[50, 70]]
Cluster 3:
  Centroid: [42.33333333 50.33333333]
  Points: [[43, 50], [43, 54], [41, 47]]
Cluster 4:
  Centroid: [56.         51.66666667]
  Points: [[55, 50], [55, 52], [58, 53]]
```

(b)

Original k-means clustering processes all data points together, updating cluster means after considering all points in the dataset. **Sequential k-means clustering** updates cluster means as each point is observed, which is useful for streaming data.

Are O1 and O2 always equal? No, they are not guaranteed to be equal. Sequential k-means can lead to different results because the order of data points affects the cluster means' updates. Early points have a larger influence on the mean calculation than later points, which might not significantly affect the means if they're stable.

Example:

Original points are:

```
[1, 2], [1.5, 1.8], [5, 8],[8, 8], [1, 0.6], [9, 11],[8, 2], [10, 2], [9, 3]
```

Initial centroids:

```
[1, 1], [2, 2]
```

K-means:

```
[1.16666667 1.46666667]
[8.16666667 5.66666667]
```

sequential k-means:

```
[1.     1.2   ]
[6.5625 4.725 ]
```

Code for sequential k-means:

```
points = np.array([
    [1, 2], [1.5, 1.8], [5, 8],
    [8, 8], [1, 0.6], [9, 11],
    [8, 2], [10, 2], [9, 3],
])
def sequential_k_means(points, k, initial_centroids):
    centroids = np.array(initial_centroids, dtype=float)
    counts = [1] * k  # Initialize counts to avoid division by zero

    for point in points:
        # Find the nearest centroid
        distances = np.sqrt(((point - centroids) ** 2).sum(axis=1))
        nearest = np.argmin(distances)

        # Update the nearest centroid
        counts[nearest] += 1
        rate = 1 / counts[nearest]
        centroids[nearest] = centroids[nearest] + (point - centroids[nearest]) *
rate

    return centroids
```

```
# Initial centroids - chosen arbitrarily for demonstration
initial_centroids = [[1, 1], [2, 2]]

seq_centroids = sequential_k_means(points, 2, initial_centroids)
print("Sequential k-means centroids:")
print(seq_centroids)
```

4

(a)

Here is the code:

```
from itertools import combinations
import numpy as np

# Example distance matrix for a small set of points (symmetric, with zero
diagonal)
distances = np.array([
    [0, 11, 5, 12, 7, 13, 9, 11],
    [11, 0, 13, 2, 17, 4, 15, 20],
    [5, 13, 0, 14, 1, 15, 12, 12],
    [12, 2, 14, 0, 18, 5, 16, 21],
    [7, 17, 1, 18, 0, 20, 15, 17],
    [13, 4, 15, 5, 20, 0, 19, 22],
    [9, 15, 12, 16, 15, 19, 0, 30],
    [11, 20, 12, 21, 17, 22, 30, 0]
])

def find_best_division(distance_matrix):
    n = len(distance_matrix)
    all_indices = set(range(n))
    best_division = None
    max_between_group_distance = -np.inf

    # Generate all non-empty, non-complete subsets of the indices for one group,
    # the other group is the complement
    for r in range(1, n):
        for subset in combinations(range(n), r):
            subset_complement = all_indices - set(subset)

            # Calculate the maximum distance between these two groups
            group_distance = max(
                distance_matrix[i, j]
                for i in subset
                for j in subset_complement
            )
            # Update the best division if this one has a larger between-group
distance
            if group_distance > max_between_group_distance:
                max_between_group_distance = group_distance
                best_division = (subset, subset_complement)

    return best_division, max_between_group_distance
```

```
# Perform the divisive algorithm step
best_division, division_distance = find_best_division(distances)
print(f"Best division: {best_division}")
print(f"Division distance: {division_distance}")
```

Result:

```
Best division: ((7,), {1, 2, 3, 4, 5, 6, 8})
```

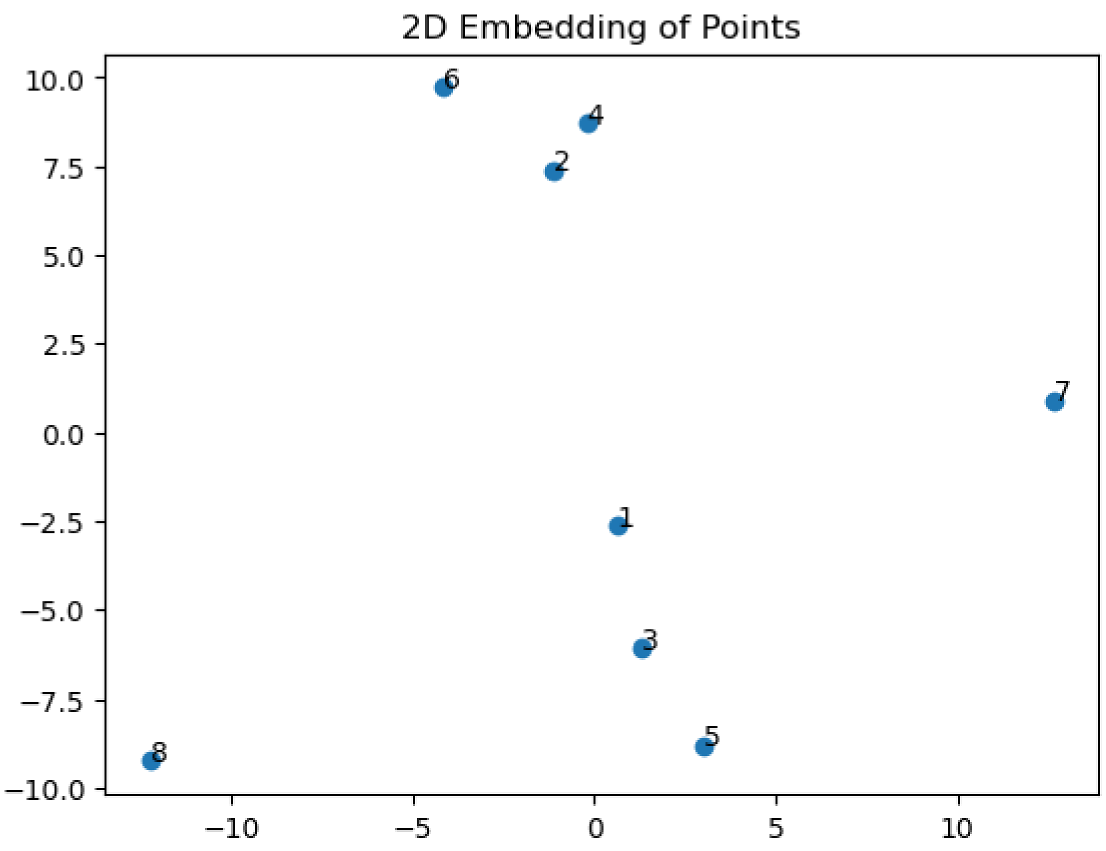
```
Division distance: 30
```

(b)

Here is the code:

```
import numpy as np
from sklearn.manifold import MDS
import matplotlib.pyplot as plt

# The given distance matrix
distances = np.array([
    [0, 11, 5, 12, 7, 13, 9, 11],
    [11, 0, 13, 2, 17, 4, 15, 20],
    [5, 13, 0, 14, 1, 15, 12, 12],
    [12, 2, 14, 0, 18, 5, 16, 21],
    [7, 17, 1, 18, 0, 20, 15, 17],
    [13, 4, 15, 5, 20, 0, 19, 22],
    [9, 15, 12, 16, 15, 19, 0, 30],
    [11, 20, 12, 21, 17, 22, 30, 0]
])

def find_embedding_with_mds(dimension):
    """
    Attempts to find a d-dimensional embedding of the points given their pairwise
distances.
    Returns the embedded coordinates and the stress of the embedding.
    """
    mds = MDS(n_components=dimension, dissimilarity="precomputed",
random_state=42, max_iter=3000, eps=1e-9)
    coords = mds.fit_transform(distances)
    stress = mds.stress_
    return coords, stress

# Try to embed in 2D
coords_2d, stress_2d = find_embedding_with_mds(2)
print(f"2D Stress: {stress_2d}")
# If the stress is too high, it indicates a poor fit, so you might try higher
dimensions

# Visualization for 2D, if applicable
if stress_2d < 100:  # This threshold is arbitrary; adjust based on your
understanding of the acceptable stress level
    plt.scatter(coords_2d[:, 0], coords_2d[:, 1])
    for i, txt in enumerate(range(1, 9)):
        plt.annotate(txt, (coords_2d[i, 0], coords_2d[i, 1]))
    plt.title("2D Embedding of Points")
    plt.show()
else:
    # If 2D embedding is not satisfactory, try higher dimensions iteratively
    for dim in range(3, 10):  # Example: trying up to 9 dimensions
        coords, stress = find_embedding_with_mds(dim)
        print(f"{dim}D Stress: {stress}")
        if stress < 100:  # Again, this stress threshold is arbitrary
            print(f"Found a satisfactory embedding in {dim} dimensions.")
            print("Coordinates:", coords)
            break
```

Here is the result:



5

(a)

```
Algorithm DensityBasedClustering
Input: Distance matrix D, ϵ (epsilon), MinPts
Output: Clusters with core and border points, and identification of noise points

1. Initialize all points as unclassified.
2. For each point p in the dataset:
   a. If p is already classified, skip to the next point.
   b. Find N(p), the ϵ-neighborhood of p, using D.
   c. If |N(p)| < MinPts, label p as a noise point (temporarily).
   d. Otherwise:
      i. Create a new cluster C, and add p to C as a core point.
      ii. For each point q in N(p):
          - If q is a noise point, add q to C as a border point.
          - If q is unclassified, add q to C.
          - Find N(q), and if |N(q)| >= MinPts, label q as a core point and merge
N(q) into N(p).
          - Continue until no more points can be added to C.

3. Label all temporarily labeled noise points as border points if they are in the
ε-neighborhood of any core point, following Principle 3.

4. Assign each border point to a cluster if it is not already assigned, based on
its ε-neighborhood core points (arbitrarily choose one if multiple options).

5. Finalize clusters, excluding noise points as per Principle 4.

End Algorithm
```

(b)

```
Processing point: 1
Point 1 is a core point, forming cluster 1.
Point 3 added to cluster 1.
Point 5 added to cluster 1.
Point 7 added to cluster 1.
Processing point: 2
Point 2 is a core point, forming cluster 2.
Point 4 added to cluster 2.
Point 6 added to cluster 2.
Processing point: 3
Processing point: 4
Processing point: 5
Processing point: 6
Processing point: 7
Processing point: 8
Point 8 labeled as noise (temporarily).
Final cluster assignments: [1. 2. 1. 2. 1. 2. 1. 0.]
```

(c)

The key operations to consider are:

1. **Finding Neighbors for Each Point**: For each of the n points, the algorithm calculates the ε-neighborhood by comparing it to every other point. This involves checking the distance between a point and all other points, which has a complexity of $O(n)$ per point, leading to an overall complexity of $O(n^2)$ for this step across all points.
2. **Processing Each Point's Neighbors**: Once the neighbors of a point are found, the algorithm may need to check the neighbors of those neighbors (for core points), potentially traversing the entire dataset in the worst case. The complexity of processing the neighbors depends on the distribution of points and the density threshold. In the worst-case scenario, where many points are densely connected, this could lead to examining a significant portion of the dataset for each core point, adding another layer of $O(n)$ complexity in the context of the entire dataset. However, this is heavily dependent on the data distribution and could be significantly less in practice.

3. **Assigning Points to Clusters**: As part of processing the neighbors, points are assigned to clusters. This operation itself is relatively simple and can be considered $O(1)$ for each assignment. However, since assignments are made within the neighbor processing loop, the overall complexity of this step is absorbed into the complexity of processing the neighbors.

Combining these considerations, the worst-case time complexity of the algorithm can be approximated as $O(n^2)$

Space Complexity:
The space complexity of the algorithm is also worth considering. It needs to store the distance matrix, which requires $O(n^2)$ space, and various arrays for bookkeeping (e.g., labels for each point), which require $O(n)$ space. Therefore, the overall space complexity is $O(n^2)$ due to the distance matrix.

In summary, the theoretical worst-case time complexity of the proposed density-based clustering algorithm is $O(n^2)$, with a space complexity of $O(n^2)$ as well. In practice, the complexity can vary significantly based on the data's characteristics and the parameters (ε and MinPts) chosen.

(d)

Conceptual Foundation

- **Density-Based Clustering**: This approach focuses on identifying clusters as regions of high density separated by regions of low density. Points in sparse areas are typically considered noise or outliers. DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a well-known example of this type of clustering.
- **Hierarchical Clustering**: Instead of a single partitioning of the data into clusters, hierarchical clustering creates a tree of clusters called a dendrogram. It can be agglomerative (bottom-up) where each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy, or divisive (top-down), starting with one cluster that is successively split into smaller clusters.

Clustering Process

- **Density-Based Clustering**: Starts with an arbitrary point, finds all points within a specified distance (ε), and creates a cluster if the number of points meets a minimum threshold (MinPts), considering the density criteria. The process iterates until all points are classified as part of a cluster or marked as noise.
- **Hierarchical Clustering**: Builds clusters step by step. In agglomerative hierarchical clustering, each point starts as a separate cluster, and the algorithm merges the closest pairs of clusters until all clusters have been merged into one. In divisive, the process starts with all points in one cluster, which is then split recursively until each point is in its own cluster.

Output

- **Density-Based Clustering**: Produces a set number of clusters based on the data's density distribution, distinguishing between core points, border points, and noise. It does not impose a cluster structure and can handle clusters of arbitrary shapes and sizes.
- **Hierarchical Clustering**: Yields a dendrogram representing the nested grouping of objects and their sequential merging or splitting. It allows for an analysis at various levels of granularity but does not specify an optimal number of clusters unless an additional criterion is applied.

Parameters

- **Density-Based Clustering**: Requires parameters such as ε (the radius around a point to search for nearby neighbors) and MinPts (the minimum number of points required to form a dense region). The choice of these parameters significantly affects the clustering outcome.
- **Hierarchical Clustering**: Does not usually require predefined parameters to begin with (except for the choice of distance metric and linkage criterion). However, deciding the cut-off point on the dendrogram, which determines the number of clusters, can be subjective.

Sensitivity to Parameters and Scalability

- **Density-Based Clustering**: The results can be sensitive to the parameters ε and MinPts. Choosing these parameters can be challenging but crucial for the performance and relevance of the clustering. It can scale well to large datasets, especially with optimized implementations.
- **Hierarchical Clustering**: It is generally not scalable to very large datasets, especially for agglomerative clustering, due to its computational complexity (often $O(n^3)$ time and $O(n^2)$ space complexities). It doesn't require initial parameters like ε and MinPts but choosing the level of granularity (number of clusters) from the dendrogram can be subjective.

Handling Noise and Cluster Shape

- **Density-Based Clustering**: Effectively handles noise and is able to identify clusters of arbitrary shapes and sizes due to its reliance on local density estimates.
- **Hierarchical Clustering**: Can be sensitive to noise and outliers. It generally assumes spherical or well-separated cluster shapes, depending on the linkage criterion used (e.g., single, complete, average).