BIRENDRA MEMORIAL COLLEGE

|  |
| --- |
| Affiliated to Tribhuvan University  Institute of Science and Technology |
|

Practical Report

Of

Data Warehousing and Data Mining

(CSC 410)

**Submitted To**  **Submitted by:**

**Mr. Teju Basnet Sandesh Adhikari**

**Birendra Memorial College Roll no:21437/075**

**7th Semester**

Date:

**NAME OF EXPERIMENT:**

Implementation of Apriori algorithm for finding frequent itemset using Python.

**INTRODUCTION:**

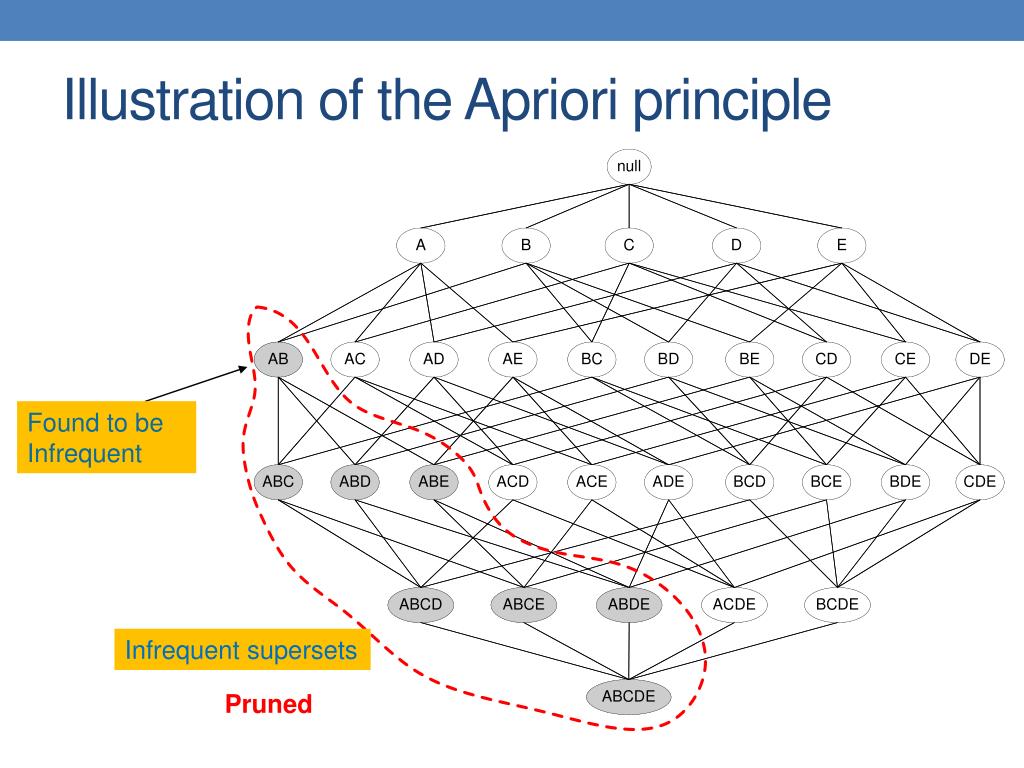
The Apriori algorithm is a sequence of steps used to find the most frequent itemset in a given database. It was developed by R. Agrawal and R. Srikant in 1994 for finding frequent itemset in a dataset for Boolean association rule.

Apriori property in the context of data cube states that “If there is any item set which is infrequent, it's superset should not be generated or tested”. We use the measures support, confidence, and lift to reduce the number of associations we need to analyze.

Support (S): Support of association rule **X =>Y** is the percentage of transactions in dataset that contain both items (**X & Y**).

Confidence (C): Confidence of association rule **X =>Y** is the percentage of transactions containing X that also contains Y.

Lift: It measures how many times more often X and Y occur together than expected if they were statically independent.



In the above figure, if the itemset {A, B} is infrequent then we do not need to consider all its supersets {A, B, C}, {A, B, D}, {A, B, E} and so on marked by red dots. This elimination helps to improve the itemset generation process.

**STEPS:**

This algorithm uses two steps “join” and “prune” to reduce the search space:

Step 1: Apriori employs an iterative approach known as level-wise search, where frequent K item sets are used to explore frequent K + 1 item sets.

Step 2: First, the set of frequent 1-itemset that satisfy minimum support is found by scanning the database. The resulting set is denoted by L1.

Step 3: Next, L1 is used to find L2 and L3 and so on until no more frequent K item sets can be found.

Step 4: The finding of each Lk requires one full scan of the database. Output all frequent item sets of size k(Lk) and stop.

**SOLUTION:**

|  |  |
| --- | --- |
| TID | **Itemset** |
| 10 | Milk, Butter, Egg |
| 20 | Bread, Butter, Cheese |
| 30 | Milk, Bread, Butter, Cheese |
| 40 | Bread, Cheese |

Use APRIORI algorithm to generate strong association rules from the following transaction database. Use min\_sup=2 and min\_confidence=75%.

🡪 Solution, here min\_sup=2 and min\_confidence=75%

Step 1: Create the itemset of the size 1 and calculate their support count.

|  |  |
| --- | --- |
| Itemset | **Support Count** |
| Milk | Checkmark outline2 |
| Bread | Checkmark outline3 |
| Butter | Checkmark outline3 |
| Egg | 1 |
| Cheese | Checkmark outline3 |

Now, we compare the support count of each itemset with min\_sup=2. As we can see, in above table D itemset does not meet the minimum criteria, thus it is discarded in the upcoming iterations, only **Milk, Bread, Butter, Cheese** meet the min\_sup count.

i.e., Support Count ≥ min\_sup

We have the frequent 1-itemset L1 as shown below:

|  |  |
| --- | --- |
| Itemset | **Support Count** |
| Milk | 2 |
| Bread | 3 |
| Butter | 3 |
| Cheese | 3 |

Step 2: Take 2-itemset and calculate the Support Count.

|  |  |
| --- | --- |
| Itemset | Support Count |
| Milk, Bread | 1 |
| Milk, Butter | Checkmark outline2 |
| Milk, Cheese | 1 |
| Bread, Butter | Checkmark outline2 |
| Bread, Cheese | Checkmark outline3 |
| Butter, Cheese | Checkmark outline2 |

Again, we compare the support count of each itemset with min\_sup=2. In above table, itemset {Milk, Bread} and {Milk, Cheese} does not meet the minimum criteria, thus it is deleted.

Step 3: Take 3-itemset and calculate the Support Count.

|  |  |
| --- | --- |
| Itemset | **Support Count** |
| Milk, Butter, Cheese | 1 |
| Bread, Butter, Cheese | Checkmark outline2 |

In the above table, only the itemset {Bread, Butter, Cheese} crosses the minimum criteria. From this frequent itemset we cannot further generate candidate itemset, so we stop here. Therefore, our final frequent itemset = {Bread, Butter, Cheese}

Step 4: Generating Association Rules from frequent itemset {Bread, Butter, Cheese} from above,

|  |  |  |
| --- | --- | --- |
| S. N | **Rules** | **Confidence** |
| 1. | {Bread, Butter} 🡪 {Cheese} | 2/2 = 1 |
| 2. | {Bread, Cheese} 🡪 {Butter} | 2/3 = 0.66 |
| 3. | {Butter, Cheese} 🡪 {Bread} | 2/2 = 1 |
| 4. | {Bread} 🡪 {Butter, Cheese} | 2/3 = 0.66 |
| 5. | {Butter} 🡪 {Bread, Cheese} | 2/3 = 0.66 |
| 6. | {Cheese} 🡪 {Bread, Butter} | 2/3 = 0.66 |

Here, the association rule number 1 and 3 is strong since it passes minimum confidence threshold 75%. So, output strong association rules are {Bread, Butter} 🡪 {Cheese} and {Butter, Cheese} 🡪 {Bread}.

**ADVANTAGES OF APRIORI ALGORITHM:**

1. Easy to understand.
2. Least memory consumption.
3. Easy implementation.
4. It uses Apriori property for pruning. Therefore, item-sets left for further support checking remain less.

**LIMITATIONS OF APRIORI ALGORITHM:**

1. It works slow compared to other algorithms.
2. It needs to generate a huge number of candidates sets if database is huge.
3. Apriori requires going through the database multiple times to check how many times each itemset appears and this can be very time-consuming and expensive.

**PROGRAM IMPLEMENTATION IN PYTHON:**

Requirement: Anaconda Navigator

**Source Code:**

**CONCLUSION:**

Hence, we have successfully implemented the Apriori algorithm in Python to find frequent itemset in given dataset.

**NAME OF EXPERIMENT:**

Implementation of K-Means Clustering in Python.

**INTRODUCTION:**

K-Means Algorithm is a centroid based clustering algorithm which groups the data into K clusters, where K is a predefined number. Each data point is assigned to the closed cluster centroid based on the distance between the point and centroid. The centroids are then recalculated until their convergence. It uses iterative relocation techniques.

**ALGORITHM:**

1. Randomly initialize K cluster centroids.
2. Calculate the Euclidean distance between each point and each centroid using formula:

d =

1. Assign each point to the cluster associated with the nearest centroid.
2. Recalculate the cluster centroids by taking the mean(x̅) of all data points assigned to each cluster. This will give you K new cluster centroids.
3. Repeat steps 3 and 4 until the cluster assignments of the data points no longer change, or until a maximum number of iterations is reached.

**SOLUTION:**

Clusters the following instances of given data with the help of K means algorithm (Take K=2).

|  |  |  |
| --- | --- | --- |
| **Instance** | **X** | **Y** |
| 1 | 31 | 31.5 |
| 2 | 31 | 34.5 |
| 3 | 32 | 31.5 |
| 4 | 32 | 33.5 |
| 5 | 33 | 32.5 |
| 6 | 33 | 34 |

→ Solution, here, K = 2.

Let the initial cluster centers are: C1 = (31, 31.5) and C2 = (32, 31.5).

**Iteration 1:** Calculate distance between cluster centers and each data point using Euclidean distance.

d(C1,2) = = 3

d(C2,2) = = 3.163

Here, d(C1,2) < d(C2,2). So, data point 1 belongs to cluster C1.

Similarly,

|  |  |  |  |
| --- | --- | --- | --- |
| **Points** | **Distance to** | | |
| **C1(31, 31.5)** | **C2(32, 31.5)** | **Clusters** |
| (31,34.5) | 3 | 3.163 | C1 |
| (32,33.5) | 2.236 | 2 | C2 |
| (33,32.5) | 2.236 | 1.414 | C2 |
| (33,34) | 3.2 | 2.7 | C2 |

Now, after iteration:

Cluster C1 = {(31, 31.5), (31, 34.5)}

Cluster C2 = {(32, 31.5), (32, 33.5), (33, 32.5), (33, 34)}

**Iteration 2:** New cluster centers are:

C1 = = (31,33)

C2 = ) = (32.5,32.8753)

Again, calculating distance between new cluster centers and each data point using Euclidean distance.

|  |  |  |  |
| --- | --- | --- | --- |
| **Points** | **Distance to** | | |
| **C1(31, 33)** | **C2(32.5, 32.875)** | **Clusters** |
| (31,31.5) | 1.5 | 2.035 | C1 |
| (31,34.5) | 1.5 | 2.22 | C1 |
| (32,31.5) | 1.8 | 1.463 | C2 |
| (32,33.5) | 1.12 | 0.8 | C2 |
| (33,32.5) | 2.06 | 0.625 | C2 |
| (33,34) | 2.236 | 0.718 | C2 |

Now, after iteration:

Cluster C1 = {(31, 31.5), (31, 34.5)}

Cluster C2 = {(32, 31.5), (32, 33.5), (33, 32.5), (33, 34)}

The cluster of data points obtained in the **2nd** iteration is same as of **1st**iteration. So, terminate.

**ADVANTAGES OF K-MEANS ALGORITHM:**

* It is easy to implement.
* It is scalable to huge data set and faster.

**DISADVANTAGES OF K-MEANS ALGORITHM:**

* It is sensitive to outliers.
* Choosing the K values manually is a tough job.

**PROGRAM IMPLEMENTATION IN PYTHON:**

Requirement: Anaconda Navigator

Graphical user interface, application

Description automatically generated

**CONCLUSION:**

Hence, we have successfully implemented the K-Means algorithm in Python to cluster the instance of given data.

**NAME OF EXPERIMENT:**

Implementation of Agglomerative Clustering in Python.

**INTRODUCTION:**

Agglomerative clustering is a type of hierarchical clustering algorithm that uses a “bottom-up” approach to build a hierarchy of clusters. It keeps on merging the objects or groups that are close to one another until all the groups are merged into one.

**ALGORITHM:**

1. Treat each data point as a single cluster and compute the distance matrix between all pairs of clusters using Euclidean distance.
2. Merge the two most similar clusters (less distance) together into a new cluster.
3. Update the distance matrix to reflect the new merged cluster.
4. Repeat steps 2-3 until k clusters remain.

**SOLUTION:**

Cluster the data points (91, 91), (91.5, 91.5), (95, 95), (93, 94), (94, 94) and (93, 93.5) into two clusters using single linkage Agglomerative clustering.

→ Solution, Assume A = (91, 91), B = (91.5, 91.5), C = (95, 95), D = (93, 94), E = (94, 94) and F = (93, 93.5).

Compute distance matrix using Euclidean distance between each data point.

AB = = 0.71

AC = = 5.65

AD = = 3.60

AE = = 4.24

AF = = 3.20

Similarly, the distance matrix is:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E | F |
| A | 0 |  |  |  |  |  |
| B | 0.71 | 0 |  |  |  |  |
| C | 5.65 | 4.94 | 0 |  |  |  |
| D | 3.60 | 2.91 | 2.23 | 0 |  |  |
| E | 4.24 | 3.53 | 1.41 | 1 | 0 |  |
| F | 3.20 | 2.5 | 2.5 | **0.5** | 1.11 | 0 |

In the above table, minimum value is 0.5 which is distance between D and F. Thus, merge D and F into the new cluster (D, F).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D, F | E |
| A | 0 |  |  |  |  |
| B | 0.71 | 0 |  |  |  |
| C | 5.65 | 4.94 | 0 |  |  |
| D, F | 3.20 | 2.5 | 2.23 | 0 |  |
| E | 4.24 | 3.53 | 1.41 | **1** | 0 |

Similarly, the minimum value is 1 which is the distance between A and B. Thus, merge A and B into the new cluster (A, B).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | A, B | C | D, F | E |
| A, B | 0 |  |  |  |
| C | 4.94 | 0 |  |  |
| D, F | 2.5 | 2.23 | 0 |  |
| E | 3.53 | 1.41 | **1** | 0 |

Again, the minimum value is 1 which is the distance between (D, F) and E. Thus, merge (D, F) and E into the new cluster (D, F), E).

|  |  |  |  |
| --- | --- | --- | --- |
|  | A, B | C | ((D, F), E) |
| A, B | 0 |  |  |
| C | 4.94 | 0 |  |
| ((D, F), E) | 2.5 | **1.41** | 0 |

In updated distance matrix, the minimum value is 1.41 which is distance between cluster C and (D, F), E). So, merge them into new cluster (((D, F), E), C).

|  |  |  |
| --- | --- | --- |
|  | A, B | (((D, F), E), C) |
| A, B | 0 |  |
| (((D, F), E), C) | 2.5 | 0 |

Constructing dendrogram,

D F E C A B

**Agglomerative Clustering**

**ADVANTAGES OF AGGLOMERATIVE ALGORITHM:**

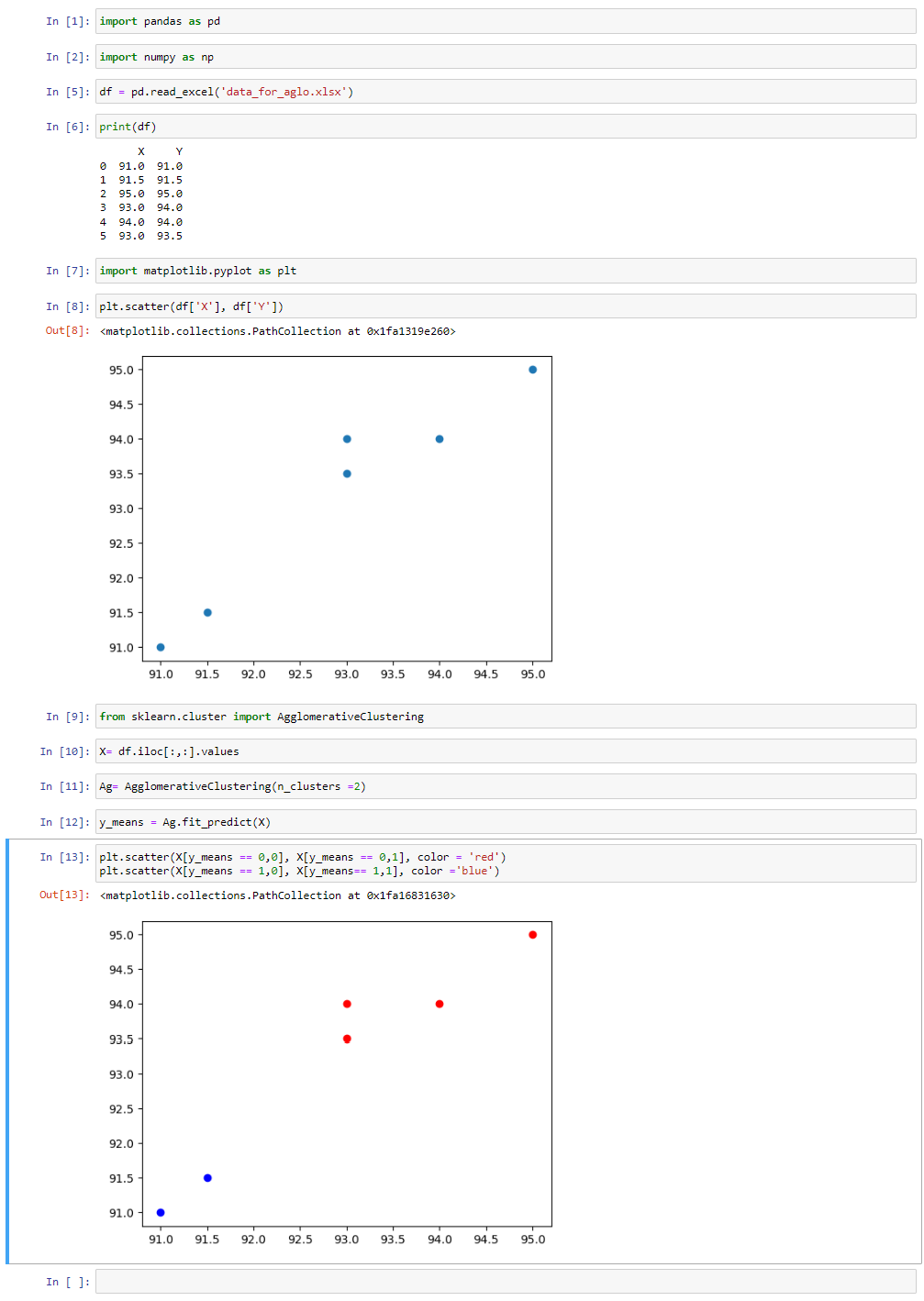
* Simple to implement and easy to interpret.
* No need to specify the number of clusters beforehand.

**DISADVANTAGES OF AGGLOMERATIVE ALGORITHM:**

* Less suitable for big data applications.
* Once clusters are merged, the process cannot be reversed.

**PROGRAM IMPLEMENTATION IN PYTHON:**

Requirement: Anaconda Navigator



**CONCLUSION:** Hence, we have successfully implemented Agglomerative Clustering in Python to cluster the given data points.