Name: Devyani Khabiya

Class: TE9-B-27

Subject: DWM

EXPERIMENT NO. 5

Title: Implementation of Association Rule Mining algorithm (Apriori)

Aim: To implement the Apriori algorithm for finding frequent itemsets based on a minimum support threshold.

Introduction: Association Rule Mining (ARM) is a fundamental technique in data mining that identifies relationships between variables in large datasets. The Apriori algorithm is one of the most widely used ARM techniques that efficiently discovers frequent itemsets and generates association rules based on minimum support and confidence thresholds. This experiment implements the **Apriori algorithm** in Python to find frequent itemsets from a given set of transactions.

Procedure:

1. **Import Necessary Libraries:**
   - Use itertools.combinations to generate candidate itemsets.
2. **Define Function to Find Frequent 1-Itemsets:**
   - *Iterate through each transaction and count the frequency of individual items.*
   - *Filter out items that do not meet the minimum support threshold.*
3. **Generate Candidate Itemsets of Size k:**
   - Use the combinations() function to generate larger itemsets from previously found frequent itemsets.
4. **Implement the Apriori Algorithm:**
   - Start with 1-itemsets and find frequent items that meet the support threshold.
   - Iteratively generate larger itemsets (*k=2, k=3,* …) and compute their support.
   - Continue the process until no more frequent itemsets are found.
5. **Run the Apriori Algorithm on Sample Transactions:**
   - Define a list of transactions.
   - Set a minimum support threshold.
   - Call the apriori() function and display the frequent itemsets.
6. **Analyze and Interpret Results:**
   - Print the frequent itemsets obtained at each level.
   - Use them to extract association rules (if required).

Program Code:

```
from itertools import combinations

# Function to get frequent itemsets based on minimum support
def get_frequent_itemsets(transactions, min_support):
    itemsets = {} for transaction in
    transactions:    for    item    in
    transaction:    if    item    in
    itemsets:

                itemsets[item] += 1
            else:
                itemsets[item] = 1

    # Filter itemsets to only include those that meet or exceed the minimum support
    frequent_itemsets = {item: support for item, support in itemsets.items() if support >= min_support}
    return frequent_itemsets

# Function to generate candidate itemsets of size k
def get_candidate_itemsets(frequent_itemsets, k):
    candidates = []
    frequent_items = list(frequent_itemsets.keys()) for
    combination in combinations(frequent_items, k):
```

```
            candidates.append(combination)
    return candidates

# Apriori algorithm to find all frequent itemsets
def apriori(transactions, min_support):
    k = 1
    # Initial set of frequent itemsets
    frequent_itemsets = get_frequent_itemsets(transactions, min_support)
    all_frequent_itemsets = [frequent_itemsets]

    # Iterate to find larger itemsets
    while frequent_itemsets:
        k += 1
        # Generate candidate itemsets of size k
        candidates = get_candidate_itemsets(frequent_itemsets, k)
        candidate_supports = {candidate: 0 for candidate in candidates}

        # Calculate support for each candidate itemset
        for transaction in transactions: for candidate
        in candidates:
                if set(candidate).issubset(set(transaction)):
                    candidate_supports[candidate] += 1

        # Filter candidate itemsets to only include those that meet or exceed the minimum support
        frequent_itemsets = {itemset: support for itemset, support in candidate_supports.items() if support >= min_support}
        if frequent_itemsets:
            all_frequent_itemsets.append(frequent_itemsets)

    return all_frequent_itemsets

# Example usage
transactions = [
    ['milk', 'bread', 'butter'],
    ['bread', 'butter'],
    ['milk', 'bread'],
    ['milk', 'butter'],
    ['bread', 'butter'],
    ['milk', 'bread', 'butter']
]

min_support = 2
frequent_itemsets = apriori(transactions, min_support)

# Display the title before output
from IPython.display import display, Markdown

display(Markdown("**Implementation/Output snap shot:**"))
print(frequent_itemsets)
```

Implementation/Output snap shot:

```
[{'milk': 4, 'bread': 5, 'butter': 5}, {('milk', 'bread'): 3, ('milk', 'butter'): 3, ('bread', 'butter'): 4}]
```

> **Conclusion:** The Apriori algorithm efficiently finds frequent itemsets in transactional data by iteratively generating and filtering itemsets based on minimum support. This experiment successfully implemented the algorithm in Python, demonstrating its ability to uncover useful patterns in datasets. The results can be applied in areas like market basket analysis and recommendation systems.

> **Review Questions:**
>
> 1. **What is the Apriori algorithm in Association Rule Mining?**
>
> **Ans.** The **Apriori algorithm** is a fundamental technique in **Association Rule Mining (ARM)** that identifies relationships between items in large datasets. It follows a **bottom-up approach**, where frequent itemsets are iteratively expanded to generate larger itemsets. The algorithm relies on the **Apriori principle**, which states that if an itemset is frequent, then all its subsets must also be frequent. By applying this principle, Apriori efficiently reduces the search space and eliminates infrequent itemsets early in the process. The algorithm is widely used in applications such as **market basket analysis**, where it helps uncover patterns like "Customers who buy bread are likely to buy butter."
>
> 2. **What is the significance of support, confidence, and lift in Apriori?**
>
> **Ans.** These three measures help in evaluating the strength of association rules:
>
> - **Support**: Measures how frequently an itemset appears in the dataset. It helps in identifying frequent patterns.
> - **Confidence**: Indicates the likelihood of consequent items appearing when the antecedent is present. It is used to determine the strength of association rules.

- **Lift**: Measures the strength of an association rule compared to random occurrence. A lift value greater than 1 indicates a strong positive correlation between itemsets.

github link:https://github.com/Pralix20/DWMexp

1}What is a Decision Tree Classifier, and How Does it Work?

A Decision Tree is a supervised learning algorithm used for both classification and regression. It works by recursively splitting the dataset into smaller subsets based on feature values to form a tree-like structure.

How it works:

1. Root Node: Represents the entire dataset.
2. Splitting: The algorithm chooses the best feature and threshold to split the data based on a criterion like Gini Index or Information Gain.
3. Internal Nodes: Represent decisions (feature-based splits).
4. Leaf Nodes: Represent the predicted class labels.
5. The process continues until a stopping condition is met (e.g., maximum depth, minimum number of samples in a node).

Explain the Naïve Bayes Algorithm and Its Underlying Assumptions

Naïve Bayes is a classification algorithm based on Bayes' Theorem, which calculates the probability of a class given a set of features.

Bayes' Theorem:

$P(C|X) = P(X)P(X|C) \cdot P(C)$ Where:

- $P(C|X)P(C|X)P(C|X)$: Posterior probability of class CCC given input features XXX
- $P(X|C)P(X|C)P(X|C)$: Likelihood of features XXX given class CCC
- $P(C)P(C)P(C)$: Prior probability of class CCC
- $P(X)P(X)P(X)$: Probability of the features

Assumptions:

- Features are independent of each other given the class label.
- It assumes equal importance of each feature and no interaction between them, which is often not realistic but works well in practice.

2}Compare the Working Principles of Decision Tree and Naïve Bayes Classifiers

Aspect	Decision Tree	Naïve Bayes
Type	Non-probabilistic, tree-based model	Probabilistic model based on Bayes' Theorem
Learning Approach	Greedy, recursive partitioning	Statistical estimation
Interpretability	Easy to interpret and visualize	Less intuitive due to probabilities
Assumption	No assumptions about data distribution	Assumes conditional independence of features
Handling Non-linear	Good at capturing non-linear relationships	Assumes feature independence, so less flexible
Performance	Can overfit without pruning	Robust, especially with small datasets

3}What Are the Different Types of Decision Tree Splitting Criteria?

- Gini Impurity: Measures the frequency of a randomly chosen element being misclassified.
- $Gini = 1 - \sum p_i^2Gini = 1 - \sum p_i^2Gini=1−∑pi2$
- Information Gain (Entropy): Measures the reduction in entropy before and after a split.
- $Entropy = -\sum p_i \log_2(p_i)Entropy = - \sum p_i \log_2(p_i)Entropy=−∑pilog2(pi)$
- Gain Ratio: A normalized version of information gain to reduce bias toward features with many levels.
- Chi-square: Statistical test to determine the relevance of a split.
- Reduction in Variance: Used for regression trees to reduce variance in the target variable after a split.