CSE304 Introduction to Data Mining
Group #10
Contact point karybekova3355@unist.ac.kr

Assignment #1

1 Apriori Algorithm

1.1 Introduction

The Apriori algorithm, introduced by Agrawal and Srikant in 1994 [1], is a foundational technique in data mining and association rule learning. It addresses the challenge of identifying significant associations between items in transactional data, aiming to uncover patterns that reveal underlying relationships or dependencies. The algorithm's importance extends beyond computer science, finding applications in various domains such as market basket analysis, bioinformatics, and recommendation systems.

The key innovation of the Apriori algorithm lies in the exploitation of the "apriori" property, which states that any subset of a frequent itemset must also be frequent. This property enables the algorithm to prune the search space efficiently, reducing computational complexity and enabling scalable mining of frequent itemsets from large transaction databases.

Over the years, the Apriori Algorithm has undergone significant evolution, driven by researchers' endeavors to improve its efficiency and scalability. Key contributions include the development of novel data structures, optimization techniques, and parallelization strategies to handle increasingly large datasets. Notable milestones include the introduction of hash-based methods, partitioning strategies, and parallel computing paradigms, all aimed at accelerating the algorithm's performance[2].

1.2 Problem Statement

This report addresses the challenge of mining frequent itemsets within transactional databases, aiming to uncover patterns of co-occurrence among items. Consider a transactional database $\mathcal{D}$ comprising sets of items, denoted as $T_1, T_2, \ldots, T_n$, where each transaction T_i consists of a distinct collection of items. The primary objective is to identify all itemsets that meet a user-defined minimum support threshold minsup.

Definition 1: An association rule is represented as $X \rightarrow Y$, where X and Y denote itemsets that do not overlap ($X \cap Y = \emptyset$). Here, $X \subseteq Z$ and $Y \subseteq D$, where $Z = \{i_1, i_2, \ldots, i_m\}$ represents a set of literals, referred to as items. Each transaction T in D comprises a subset of items from Z, and each transaction is associated with a unique identifier, termed its Transaction ID (TID). The confidence of the rule $X \rightarrow Y$ in transaction set D is determined by the percentage of transactions containing X that also contain Y. Moreover, the support of this rule in D represents the percentage of transactions containing both X and Y.

Theorem 1: Apriori Principle

Proof: The Apriori principle posits that if an itemset exhibits sufficient support (i.e., surpasses minsup), then all its subsets must also demonstrate adequate support. Let X denote a frequent itemset in transactional database D, with $\text{support}(X) > \text{minsup}$. According to the Apriori principle, all subsets of X must also be frequent to meet the support threshold.

Hence, the challenge of mining association rules entails developing efficient algorithms to:

- Traverse the transactional database D to compute the support of each itemset.
- Identify itemsets with support exceeding minsup.

1.3 Algorithm Description and Analysis

Algorithm 1: Apriori Algorithm

```
Input  : Dataset, minsup
Output: Frequent itemsets
1  $L_1 \leftarrow$ Large 1-itemsets;
2  $k \leftarrow 2$;
3  while $L_{k-1} \neq \emptyset$ do
4      $C_k \leftarrow \text{aprioriGen}(L_{k-1})$;
5      foreach transaction $t$ in dataset do
6          $C_t \leftarrow \text{subset}(C_k, t)$;
7          foreach candidate $c$ in $C_t$ do
8              $c.\text{count}++$;
9          end
10     end
11     $L_k \leftarrow \{c \in C_k \mid c.\text{count} \geq \text{minsup}\}$;
12     $k \leftarrow k + 1$;
13 end
14 return $\cup_k L_k$;
```

Definitions:

- **k-itemset**: An itemset consisting of k items.
- **Set of large k-items (L_k)**: Contains itemsets with minimum support. Each member has two fields: itemset and support count.
- **Set of candidate k-items (C_k)**: Potential large itemsets. Each member has two fields: itemset and support count.

The Apriori Algorithm aims to identify frequent itemsets within a dataset given a minimum support threshold (minsup) as input and outputs these frequent itemsets. It begins by initializing with the identification of large 1-itemsets (L_1) (Line 1). The iteration variable k is set to 2 (Line 2). The algorithm iterates over increasing sizes of itemsets until no more frequent itemsets can be found (Line 3). Within each iteration, candidate itemsets (C_k) are generated by invoking the aprioriGen function with the previous iteration's frequent itemsets (L_{k-1}) (Line 4). The algorithm then scans the dataset to count the support of candidate itemsets, incrementing their counts accordingly (Lines 5-9). For each transaction in the dataset, a subset (C_t) of candidate itemsets that exist in the transaction is identified. For each candidate itemset in C_t, its count is incremented. After processing all transactions, the algorithm retains itemsets from C_k whose counts meet or exceed the minimum support threshold, forming the new set of frequent itemsets (L_k) (Line 11). The iteration continues with incremented value of k (Line 12). Once no more frequent itemsets can be found, the algorithm returns the union of all sets of frequent itemsets discovered during the iterations (Line 14).

1.4 Example

$minsup = 2$

TID	Items
10	Apple, Carrot, Dragonfruit
20	Banana, Carrot, Eggplant
30	Apple, Banana, Carrot, Eggplant
40	Banana, Eggplant

$\xrightarrow{\text{1st scan}}$

Itemset	Sup
{Apple}	2
{Banana}	3
{Carrot}	3
{Dragonfruit}	1
{Eggplant}	3

$\longrightarrow$

Itemset	Sup
Apple	2
Banana	3
Carrot	3
~~Dragonfruit~~	~~1~~
Eggplant	3

$\longrightarrow$

Itemset	Sup
{Apple}	2
{Banana}	3
{Carrot}	3
{Eggplant}	3

$\longrightarrow$

Itemset
{Apple,Banana}
{Apple, Carrot}
{Apple, Eggplant}
{Banana, Carrot}
{Banana, Eggplant}
{Carrot, Eggplant}

$\xrightarrow{\text{2nd scan}}$

Itemset	Sup
~~Apple, Banana~~	~~1~~
{Apple, Carrot}	2
~~{Apple, Eggplant~~}	~~1~~
{Banana, Carrot}	2
{Banana, Eggplant}	3
{Carrot, Eggplant}	2

$\longrightarrow$

Itemset	Sup
{Apple, Carrot}	2
{Banana, Carrot}	2
{Banana, Eggplant}	3
{Carrot, Eggplant}	2

$\longrightarrow$

In this Apriori example, the process begins by establishing a minimum support threshold of 2. In the initial scan of the dataset, individual items such as apples, bananas,

Itemset
Banana, Carrot, Eggplant

$\xrightarrow{\text{3rd scan}}$

Itemset	Sup
Banana, Carrot, Eggplant	2

and carrots are identified as frequent itemsets if they appear in at least 2 transactions. Subsequently, pairs of frequent items, like apple and carrot combinations, are scrutinized in a second scan, with only those pairs occurring in at least 2 transactions retained for further analysis. Following this, the algorithm proceeds to examine triples of frequent items in a third scan, ensuring that each triple is also present in at least 2 transactions. Once these scans are completed, the result comprises a list of frequent itemsets along with their respective support counts, which can then be utilized for subsequent analysis or rule derivation. Throughout this iterative process, the Apriori algorithm efficiently prunes the search space, incrementally exploring and refining frequent itemsets until the support threshold is met or no further frequent itemsets can be identified.

1.5 Advantages and Disadvantages

1.5.1 Advantages

– **Scalability:** The Apriori algorithm is scalable and can efficiently handle large datasets, making it suitable for mining frequent itemsets in big data scenarios.

– **Interpretability:** The algorithm generates frequent itemsets that are easily interpretable, aiding in understanding patterns and associations within the data.

– **Simple Implementation:** The algorithm's logic is relatively straightforward, making it accessible for implementation in various programming languages.

1.5.2 Limitations

– **High Memory Usage:** The algorithm may require significant memory resources, especially when dealing with large datasets, due to the need to store candidate itemsets and transaction information.

– **Inefficient for Sparse Data:** In cases where the data is sparse, meaning there are few transactions compared to the total number of possible items, Apriori may incur unnecessary computational overhead by generating numerous candidate itemsets.

– **Prone to Generating Redundant Itemsets:** As the algorithm generates candidate itemsets based on the previous frequent itemsets, it may produce redundant itemsets, leading to longer execution times and potentially misleading results.

1.6 Experiment

Here is the link for the project: GitHub Link
First we started to parse a document and added transaction to transactionList ArrayList.

```
try (BufferedReader br = new BufferedReader(new FileReader(inputFile))) {
  String line;
  while ((line = br.readLine()) != null) {
     String[] values = line.split(",");
     // add sorted items
     Set<String> item = new HashSet<>(Arrays.asList(values));
      transactionList.add(item);
  }
}
```

After that we found frequent singletons. We implemented frequentSingletons method. It first finds unique items, then we started to check how many of them are greater than support value.

```
private static List<Item> frequentSingletons(ArrayList<Set<String>> items,
  int support) {
  // count the frequency of each item
  List<String> uniqueItems = new ArrayList<>();
  for (Set<String> item : items) {
    for (String i : item) {
       if (!uniqueItems.contains(i)) {
          uniqueItems.add(i);
       }
     }
  }
  // list with support value
  List<Item> itemsWithSupport = new ArrayList<>();
  for (String i : uniqueItems) {
    int count = 0;
    for (Set<String> item : items) {
      if (item.contains(i)) {
        count++;
      }
    }
    if (count >= support) {
      itemsWithSupport.add(new Item(new ArrayList<String>(Arrays.asList(i)),
          count));
    }
  }
  return itemsWithSupport;
}
```

Then we start main Apriori algorithm part

```
while (frequentItems.get(k - 1).size() > 0) {
  List<List<String>> candidatesK = Apriori_gen(frequentItems.get(k - 1), k);
  List<Item> itemsWithSupport = ...;
  for (Set<String> transaction : transactionList) {
    for (Item item : itemsWithSupport) {
      if (transaction.containsAll(item.itemsName)) {
        item.support++;
      }
    }
  }
  List<Item> f = new ArrayList<>();
  for (Item item : itemsWithSupport) {
    if (item.support >= support) {
      f.add(item);
    }
  }
  frequentItems.add(f);
  k++;
}
```

And in the of the code we just sort and print them out. Here we are using Apriori_gen function which will generate new itemset. It first writes down unique items in a list. And from that set it will try to pick elements with length k, after that it will check whether all subsets with length k-1 is frequent (i.e it appears in previous frequent itemset).

1.7 Conclusion and Future Directions

In conclusion, while the Apriori algorithm demonstrates effectiveness in mining frequent itemsets and revealing insights into transactional data, its limitations must be acknowledged. Future research directions could focus on optimizing the algorithm to reduce memory usage and improve efficiency, particularly for sparse datasets. Exploring alternative association rule mining algorithms or variations of Apriori may also lead to enhanced performance and scalability. Exciting new applications for the algorithm include market basket analysis for targeted marketing strategies and healthcare data mining for personalized healthcare interventions, highlighting its potential for diverse real-world applications.

2 FP-growth Algorithm

2.1 Background

The FP Growth Algorithm is a popular method for mining frequent itemsets in transactional databases. It addresses the need to efficiently identify recurring patterns in large datasets, which is essential for various applications such as market basket analysis, recommendation systems, and association rule mining.

2.2 History and Key Contributions

Introduced by Han et al. in 2000, the FP Growth Algorithm revolutionized frequent itemset mining by proposing a novel approach that eliminates the need for candidate generation. This innovation significantly improved the scalability and performance of frequent pattern mining algorithms.

2.3 Goals and Importance

The primary goal of the FP Growth Algorithm is to efficiently discover frequent itemsets from transactional data. This is important because it enables businesses to identify common purchasing patterns among customers, leading to better decision-making and targeted marketing strategies. Further research in this area is crucial for enhancing the algorithm's performance and applicability to diverse domains.

2.4 Problem Basics

The problem involves finding frequent itemsets in transactional datasets, where a frequent itemset is a set of items that appear together in a significant number of transactions.

2.5 Objectives

The main objective is to efficiently identify frequent itemsets that meet a specified minimum support threshold, facilitating meaningful analysis and insight generation from transactional data.

2.6 Algorithm Steps

The FP Growth Algorithm consists of four main steps:

- Scan the database to determine the frequency of each item.
- Sort items in descending order of frequency and remove infrequent items
- Construct the FP-tree, a compact data structure representing frequent itemsets.
- Mine frequent itemsets from the FP-tree using recursive traversal.

2.7 Implementation

- readCSV(String filePath): This function reads data from a CSV file specified by the filePath parameter. It reads each line of the CSV file, splits it into individual items separated by commas, trims whitespace from each item, and stores the items as transactions in a list of lists (List<List<String»). Each inner list represents a transaction, and each element within the inner list represents an item in that transaction.
- createFrequencyTable(List<List<String» transactions, double minSupport): This function creates a frequency table of items based on the transactions provided. It calculates the frequency of each item across all transactions, sorts the items in descending order of frequency, filters out infrequent items based on the minimum support threshold (minSupport), and returns a list of frequent items.
- arrangeItems(List<List<String» transactions, List<String> frequencyTable): This function arranges the items within each transaction based on the frequency table. It removes items that are not present in the frequency table, sorts the remaining items in each transaction based on their frequency, and returns a new list of transactions with items arranged accordingly.
- * main(String[] args): This is the main method where the execution of the program starts. It demonstrates the usage of the above functions by reading transactions from a CSV file, creating a frequency table, arranging transactions, building an FP-tree, and mining frequent itemsets from the FP-tree. * It first reads the transactions from the CSV file. * Then, it creates a frequency table of items using a minimum support threshold. * After that, it arranges the items within each transaction based on the frequency table. * Next, it builds an FP-tree using the arranged transactions.

* Finally, it mines frequent itemsets from the FP-tree and prints them along with their support values.

2.8 Advantages

- Efficiency: The FP Growth algorithm outperforms alternative frequent itemset mining methods like Apriori in terms of speed and memory usage, particularly on extensive datasets with many dimensions. Its advantage lies in creating frequent itemsets through the construction of the FP-Tree, which condenses the database and necessitates only two scans.
- Scalability: FP Growth algorithm improves as the database size and itemset dimensionality increase, rendering it a fitting choice for mining frequent itemsets in expansive datasets.
- Resistance to noise: The FP Growth algorithm exhibits greater resilience to data noise compared to other frequent itemset mining algorithms. It achieves this by exclusively generating frequent itemsets while disregarding infrequent ones, which could result from noise.
- Parallelization: The FP Growth algorithm lends itself well to parallelization, enabling its utilization in distributed computing setups and enabling efficient utilization of multi-core processors.

2.9 Limitations

- Memory Consumption: FP Growth is better on memory compared to other options for finding frequent items in data. However, it can still use a lot of memory for big datasets, because it needs to store two things: a tree structure and data breakdowns based on frequent items.
- Implementation: While FP Growth is powerful for finding frequent items in data, its inner workings are more intricate compared to other algorithms. This complexity can make it challenging to grasp and code.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In Proceedings of the 20th International Conference on Very Large Data Bases, pages 487–499. VLDB Endowment, 1994.

[2] P. Chugh and H. K. Verma. Various techniques to improve the efficiency of apriori algorithm: A review. In 2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC), pages 471–476, 2021. doi: 10.1109/ICSCCC51823.2021.9478133.

Name	**Sign**	**Individual Contribution**	**Percentage**
Abboskhon Sobirov	[illegible]	Apriori algorithm implementation	31%
Aiara Ashyrbekova		FPGrowth algorithm implementation	23%
Aizat Karybekova	[illegible]	FPGrowth algorithm implementation	23%
Makida Gebregiorgis Tesfaye		Writing a report	23%

Table 1: Percentage of the contribution must be 100% in total. For writing individual contribution, refer to `https://www.elsevier.com/researcher/author/policies-and-guidelines/credit-author-statement`