An Implementation of Apriori and FP-growth Algorithms

Xiaoyu Wan

Georgen Institute of Data Science, University of Rochester†

ABSTRACT

Apriori is the most famous frequent pattern mining method. This method scans dataset repeatedly and generate item sets by bottom-top approach. FP-growth algorithm is currently one of the fastest approaches to frequent item set mining. This approach follows the rules of the Apriori and it is based on a prefix tree representation of the given database of transactions (called an FP-tree), which can save considerable amounts of memory for storing the transactions. In this paper, I describe the process of implementation of the Apriori algorithm using an open dataset, and one possible optimization of the algorithm using. Then, I discuss the process of implementation of FP-tree and FP-growth algorithm. The results showed the FP-growth is more effective than Apriori and the improvement on time consuming with similar accuracy.

CCS CONCEPTS

• **Information systems → Information system applications → Data Mining → Association Rules**

KEYWORDS

Apriori; Apriori Improvement; FTP Growth

1 PROJECT DESCRIPTION

This project aims to implement the Apriori and FP-growth tree algorithm. The dataset in this project was extracted by Barry Becker from the 1994 Census database. The dataset included 15 attributes:age,workclass,fnlwgt,education,education_num, marital_status, occupation, relationship, race, sex, capital_gain, capital_loss, hours_per_week,and native-country, and income. In this project, each tuple is a piece of transaction and each value in the dataset represents an item.

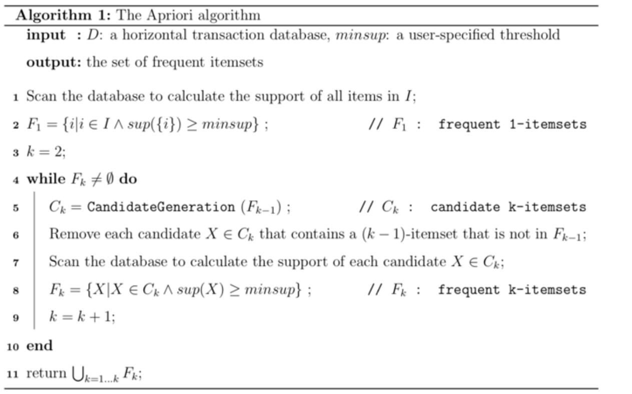
2 DATA PRE-PROCESSING

After previewing the dataset, I dropped the continuous attribute fnlwgt, which is irrelevant to the data analysis, and because of this variable included around 25,000 items, which will largely increase the computational cost.

For the rest of the continuous variables: age, education_num,capital_loss, capital_gain and hours_per_week, I adopted data discretization methods to smooth the continuous variables into bins. For The variable “age” was categorized into 4 bins: youth (10-30), middle-age(30-50), older(50-70) and senior (70-90); The same applies to education_num, meaning the years of education. It was binned to four bins: “1-5”,”6-10”, “11-15”,”16-20”. Capital_loss and capital gains are binned into two categories, labelled as “low” and “high”. Finally, the variable hours_per_week was binned into four categories: "part-time", "full-time","extra","workholic".

3 APRIORI ALGORITHM

After processing the data, the apriori algorithm is implemented to extract the frequent patters. The psedocode for the Apriori algorithm is listed as follow. Apriori takes a transaction database and the minsup threshold as input. Apriori uses a standard database representation, as shown in Table 1, also called a horizontal database. The pseudocode of Apriori is given in Algorithm 1. Apriori first scans the database to calculate the support of each item, i.e. 1-itemset (line 1). Then, Apriori uses this information to identify the set of all frequent items, denoted as F1 (line 2). Then, Apriori performs a breadth-first search to find larger frequent itemsets (line 4 to 10). During the search, Apriori uses the frequent itemsets of a given length k − 1 (denoted as Fk−1) to generate potentially frequent itemsets of length k (denoted as Ck). This is done by combining pairs of items of length k that share all but one item (line 5). For example, if the frequent 1-itemsets are {a}, {b}, {c} and {e}, Apriori will combine pairs of these itemsets to obtain the following candidate 2-itemsets: {a, b}, {a, c}, {a, e}, {b, c}, {b, e}, and {c, e}. After generating candidates of length k, Apriori checks if the (k − 1)-subsets of each candidate are frequent. If a candidate itemset X has an infrequent (k − 1)-subset, X cannot be frequent (it would violate the downward-closure property) and it is thus removed from the set of candidate k-itemsets. Then, Apriori scans the database to calculate the support of all remaining candidate itemsets in Ck (line 7). Each candidate having a support not less than minsup is added to the set Fk of frequent k-itemsets (line 8). This process is repeated until no candidates can be generated. The set of all frequent itemsets is then returned to the user (line 11).



```
Algorithm 1: The Apriori algorithm
  input  : $D$: a horizontal transaction database, $minsup$: a user-specified threshold
  output: the set of frequent itemsets

1  Scan the database to calculate the support of all items in $I$;
2  $F_1 = \{i | i \in I \wedge sup(\{i\}) \geq minsup\}$ ;          // $F_1$ :  frequent 1-itemsets
3  $k = 2$;
4  while $F_k \neq \emptyset$ do
5      $C_k$ = CandidateGeneration ($F_{k-1}$) ;          // $C_k$ :  candidate k-itemsets
6      Remove each candidate $X \in C_k$ that contains a $(k-1)$-itemset that is not in $F_{k-1}$;
7      Scan the database to calculate the support of each candidate $X \in C_k$;
8      $F_k = \{X | X \in C_k \wedge sup(X) \geq minsup\}$ ;          // $F_k$ :  frequent k-itemsets
9      $k = k + 1$;
10 end
11 return $\bigcup_{k=1...k} F_k$;
```

In this project, I first construct the one-item set I1. The Item_Scanner function is to collect all the candidate items, and generate dictionaries for all the items, with the item name as the key and the count as value. Then we traverse to store all items and their count from the transactions Ck. And then, item_scanner will traverse the items and calculate the support for all items in the Ck and store all items that meet the min_support to result list. The function will output the result list and the support of each item.

The apriori algorithm has two functions. The apriori_generator is to creates the candidate k-item sets, Ck. The algorithm inputs the frequent itemsets Lk and the item number K and outputs candidate item list Ck. In order to decrease the repetitive items for generating itemsets, the algorithm merge the sets that has the same K-2 item. The apriori_generator serves for apriori algorithm. The algorithm included a dataset, the min_support and will generate the candidate item sets for K items, such as K1, K2, K3.and use the while loop to find all frequent items that meets the min_support requirement, until the K items becomes an empty set.

3.1 Apriori Implementation

In implementing the apriori algorithm to our dataset, I found 55 5 frequent items for K items, where the largest K=7. The executio n time for apriori algorithm is user 39.8 s, sys: 267 ms, total: 40.1 s. In using the algorithm to implement the test dataset, I found 77 frequent itemsets for K items, where the largest K=4. The executi on time is user 5.69 s, sys: 63.1 ms, total: 5.76 s.

3.2 Apriori Algorithm Improvement

In order to improve the regular apriori algorithm, I adopted a method that trade off the time complexity and space com plexity[1].The algorithm can be described as follows.

The improvement of algorithm can be described as follows:

```
//Generate items, items support, their transaction ID
(1)  $L_1$ = find_frequent_1_itemsets (T);
(2)  For (k = 2; $L_{k-1} \neq \Phi$; k++) {
//Generate the Ck from the $L_{K-1}$
(3)  $C_k$ = candidates generated from $L_{k-1}$;
//get the item $I_w$ with minimum support in $C_k$ using $L_1$, $(1 \leq w \leq k)$.
(4)  x = Get _item_min_sup($C_k$, $L_1$);
// get the target transaction IDs that contain item x.
(5)  Tgt = get_Transaction_ID(x);
(6)  For each transaction t in Tgt Do
(7)  Increment the count of all items in $C_k$ that are found in Tgt;
(8)  $L_k$= items in $C_k \geq$ min_support;
(9)  End;
(10)  }
```

Firstly, the algorithm generate all items and their supports and its corresponding tranction ID. The first frequent item sets are generated by filtering the items that meets the min_support. Second, all candidate items Ck are generated from Lk-1, and the algorithm get the item I2 that meets all the minimum support. Then, each items and its corresponding tranction ID Tgt that contain item are recorded. And to traverse each item and their tranction ID, to count all items in Ck that are also found in Tgt. Fiter all the items that in Ck meets the minimun support.

For code implementation, I adapted the previous algorithm, and used three functions: trasanction_scanner, item_generator, and adapted apriori algorithm.

In implementing the impapriori algorithm to our dataset, I fo und 664 frequent items for K items, where the largest K=7. The e xecution time for improved apriori algorithm is user 13.9 s, user 1 3.9 s, sys: 584 ms, total: 14.5 s. In using the algorithm to impleme nt the test dataset, I found 661 frequent itemsets for K items, wher e the largest K=4. The execution time is user 6.92 s, sys: 180 ms, t otal: 7.1 s.

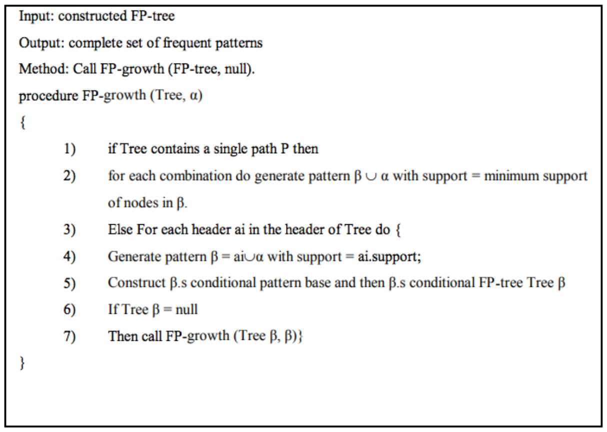
4 FP-GROWTH

4.1 Algorithm

The basic idea of the FP-growth algorithm can be described as a recursive elimination scheme: in a preprocessing step delete all items from the transactions that are not frequent individually, i.e., do not appear in a user-specified minimum number of transactions. Then select all transactions that contain the least frequent item (least frequent among those that are frequent) and delete this item from them. Recurse to process the obtained reduced (also known as projected) database, remembering that the item sets found in the recursion share the deleted item as a prefix. On return, remove the processed item also from the database of all transactions and start over, i.e., process the second frequent item etc. In these processing steps the prefix tree, which is enhanced by links between the branches, is exploited to quickly find the transactions containing a given item and also to remove this item from the transactions after it has been processed.

FP-growth is mainly used for mining frequent item sets without candidate generation. The two major steps are Step1- It firstly compresses the database showing frequent item set in to FP-tree. FP-tree is built using 2 passes over the dataset. Step2: It divides the FP-tree in to a set of conditional database and mines each database separately, thus extract frequent item sets from FP-tree directly.

The pseudocode for construct a FP-tree is as follow:



```
Input: constructed FP-tree
Output: complete set of frequent patterns
Method: Call FP-growth (FP-tree, null).
procedure FP-growth (Tree, α)
{
        1)  if Tree contains a single path P then
        2)  for each combination do generate pattern $\beta \cup \alpha$ with support = minimum support
            of nodes in β.
        3)  Else For each header ai in the header of Tree do {
        4)  Generate pattern $\beta = ai \cup \alpha$ with support = ai.support;
        5)  Construct β.s conditional pattern base and then β.s conditional FP-tree Tree β
        6)  If Tree β = null
        7)  Then call FP-growth (Tree β, β)}
}
```

3.1 FP-Growth Implementation

In the code implementation, I first build the Treenode that supports to create a tree. The other function includes the FP-tree construction and headertable update and identify the frequent item sets by mining the tree.

In implementing the FP-algorithm to the test dataset, I found 21
57 frequent items. The execution time for FP-growth algorithm is
user 726 ms, sys: 65 ms, total: 791 ms. In using the algorithm to i
mplement the adult dataset, I found 3290 frequent itemsets for.
The execution time is user 2.35 s, sys: 113 ms, total: 2.46 s.

4 CONCLUSIONS

In conclusion, the FP-growth algorithm is much faster than the Apriori algorithm to find the frequent item sets in terms of the time efficiency. It mostly relied on the times of scanning the itemsets. FP-growth only scans twice the transactions and thus saves the time. However, FP-growth traded the time efficiency with the space efficiency.

REFERENCES

[1] Najadat, Hassan M., Mohammed Al-Maolegi, and Bassam Arkok. 2013. An improved Apriori algorithm for association rules. *International Research Journal of Computer Science and Application*, 1,1(Jun,2013), 01-08.