Mining N-most Interesting Itemsets

Ada Wai-chee Fu Renfrew Wang-wai Kwong Jian Tang

Department of Computer Science and Engineering
The Chinese University of Hong Kong, Hong Kong
{adafu, wwkwong}@cse.cuhk.edu.hk

Abstract. Previous methods on mining association rules require users to input a minimum support threshold. However, there can be too many or too few resulting rules if the threshold is set inappropriately. It is difficult for end-users to find the suitable threshold. In this paper, we propose a different setting in which the user does not provide a support threshold, but instead indicates the amount of results that is required.

1 Introduction

In recent years, there have been a lot of studies in association rule mining. An example of such a rule is :

$$\forall x \in persons, buys(x, \text{"}biscuit\text{"}) \Rightarrow buys(x, \text{"}orangejuice\text{"})$$

where x is a variable and *buy(x,y)* is a predicate that represents the fact that the item y is purchased by person x. This rule indicates that a high percentage of people that buy biscuits also buy orange juice at the same time, and there are quite many people buying both biscuits and orange juice.

Typically, this method requires the users to specify the minimum support threshold, which in the above example is the minimum percentage of transactions buying both biscuits and orange juice in order for the rule to be generated. However, it is difficult for the users to set this threshold to obtain the result they want. If the threshold is too small, a very large amount of results are mined. It is difficult to select the useful information. If the threshold is set too large, there may not be any result. Users would not have much idea about how large the threshold should be. Here we study an approach where the user can set a threshold on the amount of results instead of the threshold.

We observe that solutions to multiple data mining problems including mining association rules [2, 4], mining correlation [3], and subspace clustering [5], are based on the discovery of large itemsets, i.e. itemsets with support greater than a user specified threshold. Also, the mining of large itemsets is the most difficult part in the above methods. Therefore, we would like to mine the interesting itemsets instead of interesting association rules with the constraint on the number of large itemsets instead of the minimum support threshold value. The resulting interesting itemsets are the *N-most interesting itemsets* of size k for each $k \geq 1$.

2 Definitions

Similar to [4], we consider a database D with a set of transactions T, and a set of items $I = i_1, i_2, ..., i_n$. Each transaction is a subset of I, and is assigned a transaction identifier $< TID >$.

Definition 1. *A* **k-itemset** *is a set of items containing k items.*

Definition 2. *The* **support** *of a k-itemset (X) is the ratio of number of transactions containing X to the total number of transactions in D.*

Definition 3. *The* **N-most interesting k-itemsets** *: Let us sort the k-itemsets by descending support values, let S be the support of the N-th k-itemset in the sorted list. The N-most interesting k-itemsets are the set of k-itemsets having support $\geq S$.*

Given a bound m on the itemset size, we mine the N-most interesting k-itemsets from the transaction database D for each $k, 1 \leq k \leq m$.

Definition 4. *The* **N-most interesting itemsets** *is the union of the N-most interesting k-itemsets for each $1 \leq k \leq m$. That is, N-most interesting itemset = N-most interesting 1-itemset $\cup$ N-most interesting 2-itemset $\cup$ ... $\cup$ N-most interesting m-itemset. We say that an itemset in the N-most interesting itemsets is* **interesting**.

Definition 5. *A* **potential k-itemset** *is a k-itemset that can potentially form part of an interesting (k+1)-itemset.*

Definition 6. *A* **candidate k-itemset** *is a k-itemset that potentially has sufficient support to be interesting and is generated by joining two potential $(k-1)$-itemsets.*

A potential k-itemset is typically generated by grouping itemsets with support greater than a certain value. A candidate k-itemset is generated as in the apriori-gen function.

3 Algorithms

In this section, we propose two new algorithms, which are *Itemset-Loop* and *Itemset-iLoop*, for mining N-most interesting itemsets. Both of the algorithms have a flavor of the Apriori algorithm [4] but involve backtracking for avoiding any missing itemset. The basic idea is that we automatically adjust the support thresholds at each iteration according to the required number of itemsets. The notations used for the algorithm are listed below.

P_k	Set of potential *k*-itemsets, sorted in descending order of the support values.
$support_k$	The minimum support value of the *N*-th *k*-itemset in P_k.
$lastsupport_k$	The support value of the last *k*-itemset in P_k.
C_k	Set of candidate *k*-itemsets.
I_k	Set of interesting *k*-itemsets.
I	Set of all interesting itemsets. (*N*-most interesting itemsets)

3.1 Mining *N*-most Interesting Itemsets with *Itemset-Loop*

This algorithm has the following inputs and outputs.

Inputs : A database D with the transaction T, the number of interesting itemsets required (N), the bound on the size of itemsets (m).

Outputs : *N*-most Interesting k itemsets for $1 \leq k \leq m$

Method : In this algorithm, we would find some *k*-itemsets that we call the *potential* *k*-itemsets. The potential *k*-itemsets include all the *N*-most interesting *k*-itemsets and also extra *k*-itemsets such that two potential *k*-itemsets may be joined to form interesting $(k+1)$-itemsets as in the Apriori algorithm.

First, we find the set P_1 of potential *1*-itemsets. Suppose we sort all 1-itemset in descending order of support. Let S be the support of the N-th 1-itemset in this ordered list. Then P_1 is the set of 1-itemsets with support greater than or equal to S. At this point P_1 is the N-most interesting 1-itemsets. The candidate *2*-itemsets (C_2) are then generated from the potential 1-itemsets.

The potential *2*-itemsets P_2 are generated from candidate *2*-itemsets. P_2 is the N-most interesting 2-itemsets among the itemsets in C_2. If $support_2$ is greater than $lastsupport_1$, it is unnecessary for looping back. This is the pruning effect. If $support_2$ is less than or equal to $lastsupport_1$, it means that we have not uncovered all 1-itemsets of sufficient support that may generate a 2-itemset with support greater than $support_2$. The system will loop back to find new potential 1-itemsets whose supports are not less than $support_2$. P_1 is augmented with these 1-itemsets, and the value of $lastsupport_1$ is also updated. C_2 is generated again from P_1. The new potential *1*-itemsets may produce candidate potential *2*-itemsets having support $\geq$ the value of $support_2$ in the above. P_2 is generated again from C_2, it now contains the N-most interesting 2-itemsets from C_2. The values of $support_2$ and $lastsupport_2$ are updated.

For mining potential *3*-itemsets, the system will find the candidate *3*-itemsets from P_2 with the Apriori-gen algorithm. After finding *3*-itemsets, $support_3$, and $lastsupport_3$, it will compare $support_3$ and $lastsupport_1$.

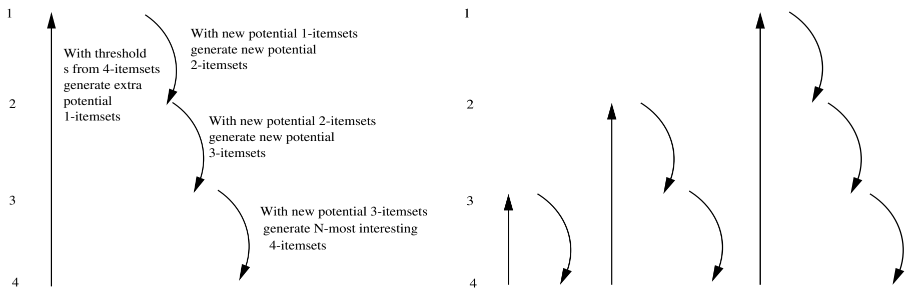
```
Algorithm 1 : Itemset-Loop
var: 1 < k ≤ m, support_k, lastsupport_k, N, C_k, P_k, D

(P_1,support_1,lastsupport_1) = find_potential_1_itemset(D,N);
C_2 = gen_candidate(P_1);
for (k=2;k < m;k++){
   (P_k,support_k,lastsupport_k) = find_N_potential_k_itemset(C_k,N,k);
   if k < m then C_{k+1} = gen_candidate(P_k); }
I_k = N-most interesting k-itemsets in P_k;
I = ∪_k I_k;
return (I);

find_N_potential_k_itemset(C_k,N,k)
{
   (P_k,support_k,lastsupport_k)=find_potential_k_itemset(C_k,N);
   newsupport = support_k;

   for(i=2;i <= k;i++) updated_i = FALSE;
   for(i=1;i < k;i++) {
      if (i = 1) {
         if (newsupport ≤ lastsupport_i) {
            (P_i,support_i,lastsupport_i) = find_potential_1_itemsets_with_support(D,newsupport);
            if i < k then C_{i+1} = gen_candidate(P_i);
            if C_{i+1} is updated then updated_{i+1} = TRUE; } }
      else {
         if (newsupport ≤ lastsupport_i or updated_i = TRUE) {
            (P_i,support_i,lastsupport_i) = find_potential_k_itemsets_with_support(C_i,newsupport);
            if i < k then C_{i+1} = gen_candidate(P_i);
            if C_{i+1} is updated then updated_{i+1} = TRUE; } }
      if (no. of k-itemsets < N and i = k and k = m) {
         newsupport = reduce(newsupport);
         for(j=2;j <= k;j++) updated_j = FALSE;
         i = 1; } }
   return(P_k,support_k,lastsupport_k);
}
```

Fig. 1. Itemset-Loop



(a) Itemset-Loop (b) Itemset-iLoop

Fig. 2. Sketch of the iterations in the step for mining N-most interesting 4-itemsets

- If $lastsupport_1$ is greater than $support_3$, it means that there may be some relevant *1*-itemsets missing. P_1 will be augmented by including 1-itemsets whose supports are $\geq support_3$. The value of $lastsupport_1$ is updated accordingly. The set C_2 candidate *2*-itemsets will be generated from P_1 again. After that P_2 is generated from C_2 including all itemsets with support $\geq support_3$. $lastsupport_2$ is updated accordingly.
- If $lastsupport_1$ is not greater than $support_3$, $support_3$ will be compared with $lastsupport_2$ of P_2. similar processing is applied to update P_2, C_3 and P_3.

This process is iterated with larger and larger itemsets and stops at the user specified bound m on the itemset size. Figure 2 (a) illustrates the idea. Next we describe the functions used.

find_potential_1_itemset(D,N) : This function finds the N-most interesting *1*-itemsets and returns these itemsets as the potential *1*-itemsets together with their supports. The itemsets are sorted in descending order of the supports and are placed in P_1. In order to obtain the support values, this function scans all the transaction records in the database.

The minimum support among the return itemsets is recorded as $support_1$ and also $lastsupport_1$.

gen_candidate(P_k) : This function generates the candidate (*k+1*)-itemsets from potential *k*-itemsets using the Apriori-gen function [4]. It will also scan the database to count the support for the newly generated candidate itemsets. A hash tree is used in this process as in [4].

find_N_potential_k_itemset(C_k,N,k) : This function finds the N-most interesting *k*-itemsets. The system will first compare $support_k$ with $lastsupport_1$. If $support_k \leq lastsupport_1$, the potential *1*-itemset is updated by adding all 1-itemsets with support $\geq support_k$. Then candidate 2-itemsets C_2 will be updated if necessary. The process is repeated with l-itemsets for $2 \leq l \leq k$.

find_potential_k_itemset(C_k,N) : This function finds potential k-itemsets from the candidate k-itemsets in C_k. The N-most interesting k-itemsets in C_k is returned. The values of $support_k$ and $lastsupport_k$ are also returned.

find_potential_1_itemset_with_support(D,$newsupport$) : This function finds all potential *1*-itemsets with the support $\geq newsupport$. All itemsets with sufficient support are stored into the potential *1*-itemset (P_1). These itemsets are returned together with $lastsupport_1$ and $support_1$.

find_potential_k_itemsets_with_support(C_i,$newsupport$) : This function finds the potential *k*-itemsets with the $newsupport$ value and the candidate *k*-itemsets. The candidates in C_i are scanned and those having support $\geq newsupport$ are returned. These are returned as P_k, the values of $lastsupport_k$, and $support_k$ are also updated and returned.

reduce($newsupport$) : This function reduces the $newsupport$ value for mining N potential *k*-itemsets if there are no enough N potential *k*-itemsets.

Correctness: The correctness of the algorithm is based on the downward closure of large itemsets : If a k-itemset $X = \{X_1, ..., X_k\}$ is large, then a $(k-1)$-itemset $Y \subset X$ must also be large. When we compute the N largest k-itemsets, and discovers the smallest support of the itemsets is S, then for a $(k-1)$-itemset, if the support is less than S, it cannot form part of an interesting k-itemset. Hence if we have considered all the $(k-1)$-itemsets with support $\geq S$ in the generation of candidate k-itemsets, we have not missed any interesting k-itemsets. Otherwise, the algorithm loops back to uncover all the smaller itemsets to uncover all l-itemsets $l < k$ which have support $\geq S$.

3.2 Second Algorithm : *Itemset-iLoop*

The first approach requires loop back in the k-th iteration to generating itemsets of size 1, 2, ..., $k-1$ in that order, using a support bound S generated at the k-itemsets. One alternative is the following : we loop back first to generate extra $(k-1)$-itemsets using S, then using these extra $(k-1)$-itemsets, we may generate more k-itemsets. With the newly generated k-itemsets, if any, we may be able to to come up with a support bound S' greater than S. With S', we may require the generation of less itemsets of size less than $k-1$. This process can be repeated with itemsets of size $k-2$, $k-3$, ... 1. Hence we propose a second algorithm based on this technique. The second proposed algorithm is similar to the first algorithm except that at the k-th iteration, instead of loop backing to the generation of potential 1-itemsets, we loop back first to examine $(k-1)$-itemsets. The algorithm is called *Itemset-iLoop*. This algorithm has the same inputs and outputs as Algorithm *itemset-Loop*.

Method : The functions in the algorithm are the same as the corresponding functions in *Itemset-Loop* algorithm except for the following:

find_N_potential_k_itemset(C_k,N,k) : This function finds n potential k-itemsets given the candidate k-itemsets C_k and a new support, $support_k$. If $support_k \geq lastsupport_{k-1}$, it is not necessary to update P_{k-1}. If $support_k < lastsupport_{k-1}$, the potential $(k-1)$-itemsets (P_{k-1}) will be updated. The missing $(k-1)$-itemsets, which have support greater than or equal to $support_k$, will be inserted into (P_{k-1}). Then candidates C_k and P_k with $support_k$, and $lastsupport_k$ will be updated. After this, the system will compare $support_k$ with $lastsupport_{k-2}$, the potential $(k-2)$-itemsets (P_{k-2}) may be updated in a similar manner. Then the potential $(k-1)$-itemsets, $support_{k-1}$, $lastsupport_{k-1}$, the potential k-itemsets, $support_k$, and $lastsupport_k$ will be updated accordingly. This is repeated with $lastsupport$ for indices $k-3$, $k-4$, ... 1. In each case, we compare $support_k$ with all $lastsupport_i$ where $i < k$, and update P_i if necessary. P_j may be updated at every pass, where $j > i$, if P_i is updated.

Note that the first two iterations are the same as that in Algorithm *Itemset-Loop*. Figure 2 (b) is a sketch of the iterations for mining potential *4*-itemset.

```
Algorithm 2 : Itemset-iLoop
find_N_potential_k_itemset(C_k,N,k)
{
  (P_k,support_k,lastsupport_k)=find_potential_k_itemset(C_k,N);
  newsupport = support_k;
  for(i=k − 1;i ≥ 1;i=i − 1) {
    if(newsupport ≤ lastsupport_i) {
      for(j=i;j ≤ k;j++) {
        if(j = 1) {
          P_j = find_potential_1_itemset_with_support(D,newsupport); }
        else {
          P_j = find_potential_k_itemset_with_support(C_j,newsupport); }
        if(j = k) {
          newsupport = support_k; }
        if(j ≠ k) {
          C_{j+1} = gen_candidate(P_j);
    } } }
    if (no. of k-itemsets < N and i = 1 and k = m) {
      newsupport = reduce(newsupport);
      i = k − 1; } }
  return(P_k,support_k,lastsupport_k);
}
```

Fig. 3. Itemset-iLoop

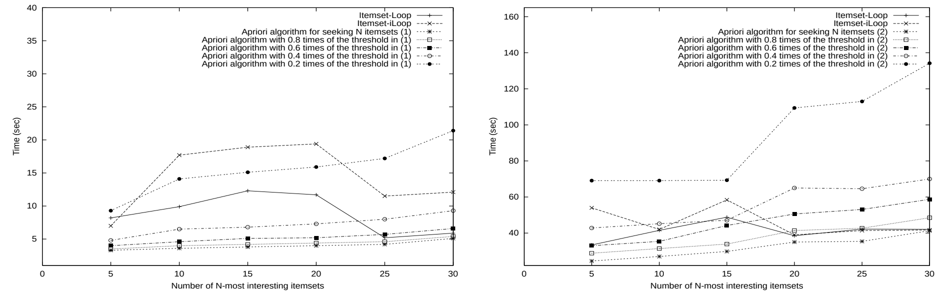
4 Experimental Results

In this section, we present the performance analysis of the algorithms *Itemset-Loop* and *Itemset-iLoop* and comparison with the Apriori algorithm [4]. All experiments were carried out on a SUN ULTRA 5_10 machine running SunOS 5.6. The workstation has 128MB memory. The hash-tree data structure [4] is used for keeping candidate itemsets. Both synthetic datasets and real datasets were used.

The real data comes from census of United States 1990. The US census database is available at the web site of IPUMS-98[1]. The experiments are based on two sets of real data: a small database with 5577 tuples and 77 different items, and a large database with 57972 tuples and 77 different items. For each database, we investigate the performance under different values of N in the N-most interesting itemsets. The different values of N are 5, 10, 15, 20, 25, and 30. We mine itemsets up to size 4, hence k-itemsets are mined for $1 \leq k \leq 4$. For the function *reduce(newsupport)* in our proposed algorithms, we choose a factor of 0.8, meaning that when the function is called, the value of newsupport is reduced to be 0.8 times its original value.

In Figure 4(a) and 4(b), we show the performance of the *Itemset-Loop* algorithm, the *Itemset-iLoop* algorithm, and the Apriori algorithm with different support thresholds for the small and the large databases respectively. We perform the algorithms *Itemset-Loop* and *Itemset-iLoop* first and take the minimum support thresholds under every N, where N are 5, 10, 15, 20, 25, and 30 after mining *4*-itemsets. And we use the notations *minsup* to represent these thresholds.

[1] The URL of IPUMS-98 is http://www.ipums.umn.edu/.



(a) small database (b) large database

Fig. 4. Performance with the growth of the number of N-most interesting itemsets

For the tiny database, the thresholds are found to be $\{0.097, 0.069, 0.062, 0.06, 0.058, 0.054\}$ for $N = 5, 10, 15, 20, 25, 30$, respectively. For the large database, the thresholds are found to be $\{0.22, 0.22, 0.22, 0.14, 0.13, 0.11\}$. [2] We apply the Apriori algorithm with these thresholds to measure the execution time. We also apply the Apriori algorithm with 0.8, 0.6, 0.4, and 0.2 of these thresholds, which we call $minsup_{0.8}$, $minsip_{0.6}$, $minsup_{0.4}$, and $minsup_{0.2}$, respectively.

In general, the performance of *Itemset-Loop* algorithm is better than that of *Itemset-iLoop algorithm*. This is because the *Itemset-Loop* algorithm loops back to the *1*-itemset first every time and updates k-itemset for $k > 1$ if necessary. On the other hand, the *Itemset-iLoop* algorithm loops back to check (k-*1*)-itemsets first and does comparisons. Then it loops back to check (k-*2*)-itemsets and updates (k-*1*)-itemsets and k-itemset if necessary, and so on so for. It may involve more back-tracking than the *Itemset-Loop* algorithm. The Apriori algorithm can provide the optimum results if the user knows the exact maximum support thresholds that can generate the N-most interesting results. We refer to this threshold as the optimal threshold. Otherwise, the proposed algorithms perform better.

We have studied the execution time for every pass using the *Itemset-Loop* and the *Itemset-iLoop* algorithms. Since we only record N or a little bit more for each itemsets for every k-itemset at the first step, it may be necessary to loop back for updating the result in both algorithms proposed. In general the increase of N leads to the increase of execution time. However, sometimes less looping back is necessary for a greater value of N and a decrease in execution time is recorded.

Table 1 shows the total number of unwanted itemsets generated by the Apriori algorithm in the large database when the guess of the thresholds is not optimal. The thresholds of $minsup_i$, where i=0.8, 0.6, 0.4 and 0.2, are used,

[2] Notice that the optimal thresholds can vary by orders of magnitude from case to case, and it is very difficult to guess the optimal thresholds.

N	$minsup_{0.8}$	$minsup_{0.6}$	$minsup_{0.4}$	$minsup_{0.2}$
5	251	395	583	1236
10	255	379	567	1220
15	105	250	437	947
20	372	541	921	1624
25	374	587	957	1854
30	467	800	1016	2322

Table 1. Number of unwanted itemsets generated by Apriori (large database)

$minsup_i$ is i times the optimal minimum support thresholds. We can see that the unwanted information can increase very dramatically with the deviation from the optimal thresholds.

We have also carried out another set of experiments on synthetic data. The results are similar in that the proposed method is highly effective and can outperform the original method by a large margin if the guess of the minimum support threshold is not good. For the interest of space the details are not shown here.

5 Conclusion

We proposed two algorithms for the problem of mining *N-most interesting k*-itemsets. We carried out a number of experiments to illustrate the performance of the proposed techniques. We show that the proposed methods do not introduce much overhead compared to the original method even with the optimal guess of the support threshold. For thresholds that are not optimal by a small factor, the proposed methods have much superior performance in both efficiency and the generation of useful results.

References

1. N. Megiddo, R. Srikant : Discovering Predictive Association Rules. Proc. of the 4th Int'l Conf. on Knowledge Discovery in Databases and Data Mining (1998)
2. J. Han, Y. Fu : Discovery of Multiple-Level Association Rules from Large Databases. Proc. of the 21st Int'l Conf. on Very Large Data Bases (1995) 420-431
3. S. Brin, R. Motwani, C. Silverstein : Beyond Market Baskets: Generalizing Association Rules to Correlations. Proc. of the 1997 ACM SIGMOD International Conference on Management of Data (1997) 265-276
4. R. Agrawal, R. Srikant : Fast Algorithms for Mining Association Rules. Proc. of the 20th Int'l Conf. on Very Large Data Bases (1994) 487-499
5. R. Agrawal, J. Gehrke, D. Gunopulos, P. Raghavan : Automatic Subspace Clustering of High Dimensional Data for Data Mining Application. Proc. of the 1996 ACM SIGMOD Int'l Conf. on Management of Data (1998) 94-105