An Efficient Implementation of an Algorithm for Mining Locally Frequent Patterns

Fokrul Alom Mazarbhuiya
College of Computer Science & IT, Albaha University, Albaha, KSA
fokrul_2005@yahoo.com

ABSTRACT

Mining patterns from large dataset is an interested data mining problem. Many methods have been developed for this purpose till today. Most of the methods considered the time attributes as one of the normal attribute. However taking the time attribute into account separately the patterns can be extracted which cannot be extracted by normal methods. These patterns are termed as temporal patterns A couple of works have already been done in mining temporal patterns. A nice algorithm for mining locally frequent patterns from temporal datasets is proposed by *Anjana et al*. In this article, we propose a hash-tree based implementation of the algorithm. We also established the fact that the hash-tree based outperforms others.

Keywords

Data Mining, Frequent patterns, Temporal patterns, Locally frequent patterns.

1. INTRODUCTION

The problem frequent item set mining is well- researched field of data mining and is associated with association rule mining in market basket data [1]. There are a number of algorithms proposed till today for mining such datasets. A-priori algorithm [2], is one of the most popular algorithms. But market basket data are usually temporal in nature e.g. when a transaction happens the time of transaction is also recorded with the transaction. Considering the time features of such datasets, some interesting patterns can be extracted which otherwise cannot be extracted. In [3], Ale et al devised a method of extracting association rules which hold throughout the life-time of an itemset where the life-time of an item set is defined as the time period between the first transaction and last transaction containing the item set and it may not be same as that of dataset. Although the algorithm [3] extracts much more rules than normal A-priori algorithm, it has some limitations. For example, the method works well if the items are uniformly distributed in the transactions throughout its life-time. But in practice there may exist items, which may not be uniformly distributed in the transactions throughout their lifetime e.g. seasonal items like cold drinks. For such items if the items life-time is taken into consideration, they may not turn out to be frequent because of the large time period when the item is absent in the transaction or even if the items are frequent then they will have very small support value. Considering the seasonal behavior of certain items in the transactions, B. Ozden et al [4] put forward a method of finding cyclic association rules where they proposed algorithms to extract all such rules holding within a user-specified time period. Once the user chooses the time period it will be fixed throughout the execution of the algorithm. In [5, 6], authors tried to address the above limitations. The frequent itemsets extracted by [5, 6] is known as locally frequent itemsets. In this paper, our discussions are mostly emphasized on the implementation side of [5, 6]. We propose here a hash-based implementation of the algorithm [5, 6]. The nicety about the hash-tree based implementation is that it reduces the number of comparisons and store the candidate in a hash-tree. This paper is organized as follows. In section-2, we discuss about definitions and notations used in [5, 6]. In section-3, we discuss the algorithm of [5, 6]. In section-4, we discuss about the detailed implementation. Conclusion and Lines for future works are discussed in section-5.

2. TERMS, DEFINITIONS AND NOTATIONS USED

Let Let T = <to, t1…> be a sequence of time-stamps over which a linear ordering < is defined where ti < tj means ti denotes a time which is earlier than tj. Let I denote a finite set of items and the transaction dataset D is a collection of transactions where each transaction has a part which is a subset of the item set I and the other part is a time-stamp indicating the time in which the transaction had taken place. We assume that D is ordered in the ascending order of the time-stamps. For time intervals we always consider closed intervals of the form [t1, t2] where t1 and t2 are time-stamps. We say that a transaction is in the time interval [t1, t2] if the time-stamp of the transaction say t is such that t1 ≤ t ≤ t2.

We define the local support of an item set in a time interval [t1, t2] as the ratio of the number of transactions in the time interval [t1, t2] containing the item set to the total number of transactions in [t1, t2] for the whole dataset D. We use the notation $Supp_{[t_1,t_2]}$ (X) to denote the support of the item set X in the time interval [t1, t2]. Given a threshold σ we say that an item set X is frequent in the time interval [t1, t2] if $Supp_{[t_1,t_2]}$ (X) ≥ (σ/100)* tc where tc denotes the total number of transactions in D that are in the time interval [t1, t2].

3. FINDING LOCALLY FREQUENT ITEMSETS WITH ASSOCIATED TIME INTERVALS

Here for the sake of convenience, we discuss the algorithm used in [5, 6] for finding locally frequent itemsets. While constructing locally frequent sets, with each locally frequent set a list of time-intervals is constructed in which the set is frequent. Two thresholds minthd1 and minthd2 are used and these are given as input. During execution, while making a pass through the database, if for a particular item set the time gap between its current time-stamp and the time when it was last seen (before the current time-stamp) is less than the value of minthd1 then the current transaction is included in the current time-interval under consideration; otherwise a new time-interval is started with the current time-stamp as the starting point. The support count of the item set in the previous time interval is checked to see whether it is frequent in that interval or not and if it is then it is added to the list maintained for that set. Also for the locally frequent sets a minimum period length is given by the user as minthd2 and time intervals of length greater than or equal to this value are only kept. If minthd2 is not used than an item appearing once in the whole database will also become locally frequent.

Procedure to compute L1, the set of all locally frequent item sets of size 1.

For each item while going through the database we always keep a time-stamp called lastseen that corresponds to the time when the item was last seen. When an item is found in a transaction and the time-stamp is tm and the time gap between lastseen and tm is greater than the minimum threshold given, then a new time interval is started by setting start of the new time interval as tm and end of the previous time interval as lastseen. The previous time interval is added to the list maintained for that item provided that the duration of the interval and the support of the itemset in that interval are both greater than or equal to the minimum thresholds specified for each. Otherwise lastseen is set to tm, the counters maintained for counting transactions are increased appropriately and the process is continued. Following is the algorithm to compute L1, the list of locally frequent sets of size-1. Suppose the number of items in the dataset under consideration is n and we assume an ordering among the items.

Algorithm 1

```
C1 = {(ik,tp[k]) : k = 1,2,…..,n}
 where ik is the k-th item and tp[k] points to a list of time intervals initially
empty
for k = 1 to n do
    set lastseen[k], icount[k] ctcount[k] and ptcount[k]  to zero
for each transaction t in the database with time stamp tm do
  {for k = 1 to n do
     {if ({ik} ⊆ t) then
        {if(lastseen[k] == 0)
            {lastseen[k] = firstseen[k] = tm;
             icount[k] = ptcount[k]=ctcount[k] =1;
            }
         else  if (|tm – lastseen[k]| < minthd1)
               {lastseen[k]=tm; itemcount[k]++;
                 ctcount[k]++; ptcount[k]=ctcount[k];
```

Three support counts icount, ctcount and ptcount are maintained with each item. When an item is first seen then these are initialized to 1. For each item while making a pass through the dataset when a transaction containing the item is found then icount for that item is increased. To see whether an item is frequent in an interval the total number of transactions in that interval will have to be counted. For this with each item two counts ptcount and ctcount are kept. The value of ctcount increases with each transaction but ptcount changes its value only when a transaction containing an item is found within

```
               }
             else
               {if ((|lastseen[k] – firstseen[k]| ≥  minthd2)
                       &&(icount[k]/ptcount[k]*100 ≥ σ))
                    add(firstseen[k], lastseen[k]) to tp[k];
                icount[k] = ptcount[k] = ctcount[k]= 1;
                lastseen[k] = firstseen[k] = tm;
               }
         }
       else ctcount[k]++;
     }          // end of k-loop //
  }    // end of do loop //
for k = 1 to n do
  {if ((|lastseen[k] – firstseen[k]| ≥  mintdh2)  and
        (icount[k] /ptcount[k] * 100 ≥ σ))
           add  (firstseen[k], lastseen[k] ) to tp[k];
   if(tp[k] != 0) add (ik, tp[k]) to L1
 }
```

After this, A-priori candidate generation algorithm is used to find candidate frequent sets of size-2 and then pruning is applied. With each candidate frequent set of size-2 we associate a list of time intervals. In the candidate generation phase this list is empty. During the pruning phase this list is constructed. The procedure of construction is that when the first subset of an item set appearing in the previous level is found then that list is taken as the list of time intervals associated with the set. When subsequent subsets are found then the list is reconstructed by taking all possible pair wise intersection of subsets one from each list. If this list becomes empty at any point of time or when a particular subset of the item set under consideration is not found in the pervious level then the set is pruned. Pair-wise intersection of the interval lists are taken for the following reason. If in an interval say [t, t'] an item set say {A,B} is frequent then there exits two time periods [t1, t1'] and [t2, t2'] in which the item sets {A} and {B} are respectively frequent and $[t, t'] \subseteq [t1, t1'] \cap [t2, t2']$. Using this concept we describe below the modified A-priori algorithm for the problem under consideration.

Algorithm 2
Modified A priori

```
Initialize
  k = 1;
C1 = all item sets of size-1
L1 = { frequent item sets of size-1 where with each itemset {ik} a list
tp[k] is maintained       which gives all time intervals in which the set
is frequent}
L1 is computed using algorithm 3.1 */
for(k = 2; Lk-1 ≠ ϕ ; k++) do
      { Ck = apriorigen(Lk-1)
/* same as the candidate generation method of the A-priori algorithm
setting tp[i] to zero for all i*/
        prune(Ck);
        drop all lists of time intervals maintained with the sets in Ck
       Compute Lk from Ck.
//Lk can be computed from Ck using the same procedure used for
computing L1 //
      k = k + 1
      }
```

Answer = $\bigcup_k L_k$

Prune(Ck)
{Let m be the number of sets in Ck and let the sets be s1, s2,…, sm. Initialize the pointers tp[i] pointing to the list of time-intervals maintained with each set si to null

```
for i = 1 to m do
    {for each (k-1) subset d of si do
        {if d ∉ Lk-1 then
            {Ck = Ck - {si, tp[i]}; break;}
        else
            {if (tp[i] == null) then set tp[i] to point to the list of time intervals maintained for d
                else { take all possible pair-wise intersection of time intervals one from each list,
                one list maintained with tp[i] and the other maintained with d and take this as the list
                for tp[i]
                delete all time intervals whose size is less than the value of minthd2
                if tp[i] is empty then {Ck = Ck - {si,tp[i]};
                                        break;  }
                }
            }
        }
    }
}
```

4. IMPLEMENTATION

4.1 Data Structure Used

The candidate generation and the support counting processes require an efficient data structure. Hash-tree data structure is used in this purpose because it reduces the number of comparisons by storing the candidate in hash-tree and hence making the execution of the algorithm faster.

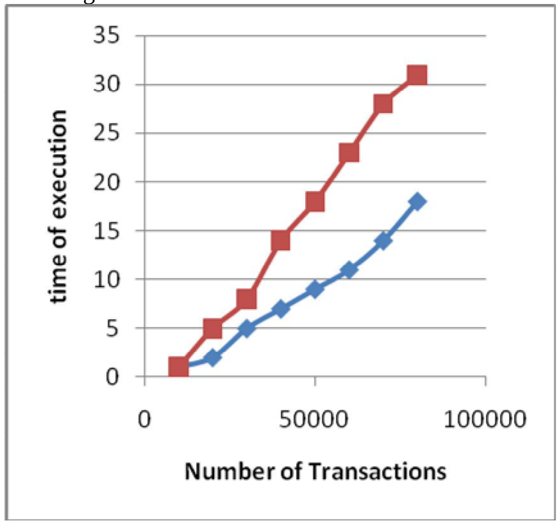

Figure 1: Graph of frequent itemsets for retail dataset

4.2 Datasets

For the experiments we performed here, we used two datasets with different characteristics. We have experimented using one retail dataset [7], and one synthetic dataset available at http://fimi.cs.helsinki.fi/testdata.html.

4.3 Analysis of Obtained Results

In this section we discuss about the comparative study of the performances of hash-based implementation and trie-based implementation. The results are described in the graphical form in figure 1 and figure 2.

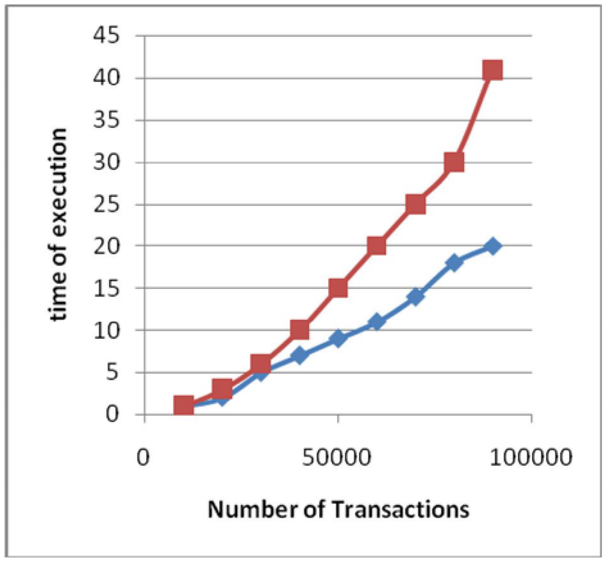

Figure 2: Graph of frequent itemsets for T10I4D100K dataset

In above two graphs the green lines are associated with execution of hash tree- based implementation and red lines are that of trie-based implementation. The two figures clearly show that the in hash tree-based implementation the execution is faster. It establishes the fact that hash tree-based implementation is more efficient than trie-based implementation.

5. CONCLUSIONS

In this paper, we present hash tree -based implementation of the algorithm [5, 6] and [3]. We made a comparative study with the performance of trie-based implementation. It is found that hash tree-based implementation is much faster than the trie-based implementation irrespective of the datasets under consideration by reducing the number comparisons in candidate generation. Thus it outperforms trie-based implementation. In future, we will go for other type of implementation like kd-tree based.

REFERENCES

[1] R. Agrawal, T. Imielinski and A. N. Swami, Mining association rules between sets of items in large databases, In Proc. of 1993 ACM SIGMOD Int'l Conf on Management of Data, Vol. 22(2) of SIGMOD Records, ACM Press, (1993), pp 207-216.

[2] R. Agrawal and R. Srikant; Fast Algorithms for Mining Association Rules, In Proc. of the 20th VLDB Conf., Santiago, Chile, 1994.

[3] J. M. Ale and G. H. Rossi; An approach to discovering temporal association rules, In Proc. of 2000 ACM symposium on Applied Computing (2000).

[4] B. Ozden, S. Ramaswamy and A. Silberschatz; Cyclic Association Rules, In Proc. of the 14th Int'l Conf. on Data Engineering, USA (1998), pp. 412-421.

[5] A. K. Mahanta, F. A. Mazarbhuiya and H. K. Baruah; Finding Locally and Periodically Frequent Sets and Periodic Association Rules, In Proc. of 1st Int'l Conf. on Pattern Recognition and Machine Intelligence, LNCS 3776 (2005), pp. 576-582.

[6] A. K. Mahanta, F. A. Mazarbhuiya and H. K. Baruah, Finding calendar-based periodic patterns, Pattern Recognition Letters, vol.29, no.9, pp.1274-1284, 2008.

[7] Tom Brijs, G. Swinnen, K. Vanhoof and G. Wets; using association rules for product assortment decisions: A case study. In Knowledge Discovery and Data Mining (1999), pp.254-260.