IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 10, Issue 2 (Mar. - Apr. 2013), PP 82-86
www.iosrjournals.org

A Survey on Identification of Closed Frequent Item Sets Using Intersecting Algorithm for Transaction in Data Mining

Veenita Gupta[1] , Neeraj Kumar[2] , Praveen Kumar[3]
1*(Computer Science, Amity University Noida UP , India)*
2(Computer Science, Amity University Noida UP , India)
3(Computer Science, Amity University Noida UP , India)

Abstract: *Most known frequent item set mining approaches enumerate candidate item sets, determine their support, and prune candidates that fail to reach the user-specified minimum support. Apart from this scheme we can use intersection approach for identifying frequent item set. But the intersection approach of transaction is the less researched area and need attention and improvement to be applied. To the best of our knowledge, there are only two basic algorithms: a cumulative scheme, which is based on a repository with which new transactions are intersected, and the Carpenter algorithm, which enumerates and intersects candidate transaction sets. These approaches yield the set of so-called closed frequent item sets, since any such item set can be represented as the intersection of some subset of the given transactions.As the transactional database increases, the size of prefix tree also grows which make it difficult to handle. An improvement has been suggested to reduce the total number of branches in the prefix tree leading to reduction in its size.*
Keywords : *algorithm , closed item set , frequent item set mining, intersection ,transaction.*

I. Introduction

It is often the case that large collections of data, however well structured, conceal implicit patterns of information that cannot be readily detected by conventional analysis techniques Such information may often be usefully analyzed using a set of techniques referred to as knowledge discovery or data mining. These techniques essentially seek to build a better understanding of data, and in building characterizations of data that can be used as a basis for further analysis, extract value from volume. Data mining is essentially the computer-assisted process of information analysis. Most of the approaches in data mining enumerate candidate item sets, determine their support, and prune candidates that fail to reach the user-specified minimum support. Apart from this scheme we can use intersection approach for identifying frequent item set. But the intersection approach of transaction is the least research area and need attention and improvement to be applied. The main reason why the intersection approach is less researched is that it is often not competitive with the item set enumeration approaches, at least on standard benchmark data sets. Naturally, if there are few items, there are (relatively) few candidate item sets to enumerate and thus the search space of the enumeration approaches is of manageable size. In contrast to this, the more transactions there are, the more work an intersection approach has to do, especially, since it is not linear in the number of transactions like the support computation of the item set enumeration approaches.

1.1 Basic Definition

1.1.1 Association rule mining: - Association rule mining[1] is a type of mining used to identify certain kind of association (interconnection relation in term of probability) among the items of database. It aims is to extract interesting correlations, frequent patterns, associations or casual structures among sets of items in the transaction or other data repositories.
1.1.2 Support**: -** It specifies the fraction of transactions that contains an item sets.
1.1.3 Confidence: - It denotes the measure of trueness of the generated association rule true in probabilistic term.
1.1.4 Frequent item set:-A frequent item set is the item set whose support is greater than some user specified minimum support.
1.1.5 Closed item set:-A frequent item set is closed if there does not exist a super set that has the same support [2].

II. Frequent Item Set Mining

An important observation while mining frequent item sets is that the output is often huge and it may even exceed the size of the transaction database to mine. As a consequence, there are several approaches that try to reduce the output, if possible without any loss of information. The most basic of these approaches is to restrict the output to so-called closed or maximal frequent item sets [3]. Restricting the output of a frequent item set

mining algorithm to only the closed or even only the maximal frequent item sets can sometimes reduce it by orders of magnitude. However, little information is lost: From the set of all maximal frequent item sets the set of all frequent item sets can be reconstructed, since any frequent item set has at least one maximal superset. Therefore the union of all subsets of maximal item sets is the set of all frequent item sets. Closed frequent item sets even preserve knowledge of the support values. The reason is that each frequent item set has a uniquely determined closed superset with the same support. Hence the support of a frequent item set that is not closed can be computed as the maximum of the support values of all closed frequent item sets that contain it (the maximum has to be used, because no superset can have a greater support-the so-called apriori property[4]). As a consequence, closed frequent item sets are the most popular form of compressing the result of frequent item set mining.

2.1 Basic Notions and Notation

Formally, the task of frequent item set mining can be described as follows:

B:-Base Item Set.

1. T: - (t_1, t_2….. t_n) Transaction Database over an item base B.
2. K_T (I):- Cover K_T (I) of an item set I which is subset of B with respect to this database is set of indices of transactions that contain it. $K_T(I) = \{k \in \{1,.....,n\}\} \mid I \subseteq t_k$
3. S_T (I):- Support of an item set I which is subset of B is the size of cover.
4. We can write it as $S_T = | K_T(I)$ That is, the number of transactions in the database T it is contained in.
5. Given a user-specified minimum support S_{min} for an item set I, an item set I is called frequent in T if and only if S_T (I) >=S_{min}.

2.2 Item Set Enumeration Algorithms

A standard approach to find all frequent item sets w.r.t. a given database T and support threshold smin, which is adopted by basically all frequent item set mining algorithms (except those of the Apriori family), is a depth-first search in the subset lattice of the item base B. This approach can be interpreted as a simple divide-and-conquer scheme. For some chosen item i, the problem to find all frequent item sets is split into two subproblems: (1) find all frequent item sets containing the item i and (2) find all frequent item sets not containing the item i. Each subproblem is then further divided based on another item j: find all frequent item sets containing (1.1) both items i and j, (1.2) item i, but not j, (2.1) item j, but not i, (2.2) neither item i nor j, and so on. All subproblems that occur in this divide-and-conquer recursion can be de_ned by a conditional transaction database and a prefix. The prefix is a set of items that has to be added to all frequent item sets that are discovered in the conditional database. Formally, all sub problems are tuples S = (C; P), where C is a conditional transaction database and P is a prefix. The initial problem, with which the recursion is started, is S = (T;φ), where T is the transaction database to mine and the prefix[5] is empty.

2.3 Types of Frequent Item Sets

One of the first observations one makes when mining frequent item sets is that the output is often huge-it may even exceed the size of the transaction database to mine. As a consequence, there are several approaches that try to reduce the output, if possible without any loss of information. for example:

2.3.1 Closed Item Set **:** A frequent item set is called closed if there does not exist a superset that has the same support, or formally

$$I \subseteq B \text{ is closed} \Leftrightarrow S_T(I) \geq S_{\min} \wedge \forall_i \in B - I : S_T(I \cup \{i\}) < S_T(I) \quad \text{equation 1}$$

2.3.2 Maximal Item Set**:** A frequent item set is called maximal if there does not exist any superset that is frequent, or formally:

$$I \subseteq B \text{ is maximal} \Leftrightarrow S_T(I) \geq S_{\min} \wedge \forall_i \in B - I : S_T(I \cup \{i\}) < S_{\min}. \quad \text{equation 2}$$

From the set of all maximal frequent item sets the set of all frequent item sets can be reconstructed, since any frequent item set has at least one maximal superset. Therefore the union of all subsets of maximal item sets is the set of all frequent item sets. Closed frequent item sets even preserve knowledge of the support values. Note that closed item sets are closely related to perfect extensions: an item set is closed if it does not have a perfect extension. However, using perfect extension pruning does not mean that the output is restricted to closed item sets, because in the search not all possible extension items are considered (conditional databases do not contain all items).

III. Intersecting Transactions

We discuss two ways of implementing the intersection approach: enumerating transaction sets as it is done in the Carpenter algorithm [6] and a cumulative scheme [7].

3.1 Enumerating Transaction Sets

The Carpenter algorithm [6] implements the intersection approach by enumerating sets of transactions (or, equivalently, sets of transaction indices) and intersecting them. Technically, the task to enumerate all transaction index sets is split into two sub-tasks: (1) enumerate all transaction index sets that contain the index 1 and (2) enumerate all transaction index sets that do not contain the index 1. These sub-tasks are then further divided w.r.t. the transaction index 2: enumerate all transaction index sets containing (1.1) both indices 1 and 2, (1.2) index 1, but not index 2, (2.1) index 2, but not index 1, (2.2) neither index 1 nor index 2, and so on.

3.2 Prefix Tree Implementation

Prefix tree, is an ordered tree data structure that is used to store an associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node, instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that correspond to keys of interest. In the example shown, keys are listed in the nodes and values below them. Each complete English word has an arbitrary integer value associated with it. A prefix tree can be seen as a deterministic finite automaton, although the symbol on each edge is often implicit in the order of the branches.

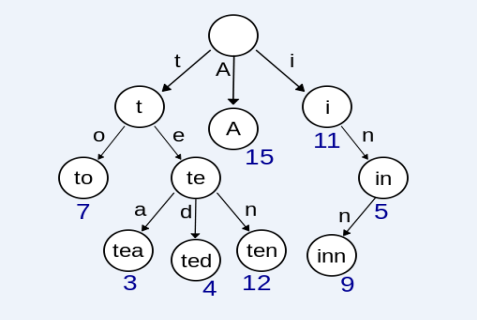

Fig-1 Prefix Tree

It is not necessary for keys to be explicitly stored in nodes. (In the figure, words are shown only to illustrate how the prefix tree works.) Prefix tree need not be keyed by character string. The same algorithms can easily be adapted to serve similar functions of ordered lists of any construct, e.g., permutations on a list of digits or shapes. In particular, a bitwise prefix tree is keyed on the individual bits making up a short, fixed size of bits such as an integer number or pointer to memory.

IV. Intersecting Algorithm

In the intersection approach after adding the transaction to the prefix tree we need to perform the intersection of currently added transaction with all the existing transaction in the prefix tree. The proposed algorithm for performing the intersection is defined as:-

```
void isect (NODE_ node, NODE **ins)
{                                                          //intersect with transaction
    int i;                                                  // buffer for current item
    NODE *d;                                                // to allocate new nodes
    while (node)
    {                                                       // traverse the sibling list
        i = node->item;                                    // get the current item
        if (trans[i])
            {                                              // if item is in intersection
                while ((d = *ins) && (d->item > i))
                ins = &d->sibling;                          //find the insertion position
                if (d                                  // if an intersection node with
                && (d!item == i))                           // the item already exists
                    {
```

```
                                        if (d->step >= step) d->supp--;
                                        if (d->supp < node->supp)
                                        d->supp = node->supp;
                                        d->supp++; // update intersection support
                                        d->step = step;
                                    }                   // and set current update step
                                    else
                                    {                   // if there is no corresp. node
                                    d = malloc(sizeof(NODE));
                                    d->step = step;     // create a new node and
                                    d->item = i;                // set item and support
                                    d->supp = node->supp+1;
                                    d->sibling = *ins;
                                    *ins = d;
                                    d->children = NULL;
                                    }                           // insert node into the tree
                            if (i <= imin) return;                  // if beyond last item, abort
                            isect(node->children, &d->children);
                            }
                        else
                        {                                       // if item is not in intersection
                        if (i =< imin) return;                      // if beyond last item, abort
                        isect(node->children, ins);
                        g
                        }                                           // intersect with subtree
                node = node->sibling;                           // go to the next sibling
                }                                                   // end of while (node)
}                                                                   // isect()
```

In more detail, the procedure works as follows: whenever the item in the current sibling equals an item in the transaction (that is, whenever trans[i] is true) and thus the item is in the intersection, it is checked whether the sibling list starting at *ins contains a node with this item, because this node has to represent the extended intersection. If such a node exists, its support is updated. For this update the step field and variable are vital, because they allow us to determine whether the current transaction was already counted for the support in the node or not. If the value of the step field in the node equals the current step, the node has already been updated and therefore the transaction must be discounted again before taking the maximum. The maximum is taken, because we have to determine the support from the largest set of transactions containing the item set represented by the node. If a node with the intersection item does not exist, a new node is allocated and inserted into the tree at the location indicated by ins. In both cases (with the current item in the transaction or not, that is, with trans[i] true or false) the sub-tree is processed recursively, unless the item of the current node is not larger than the minimum item in the current transaction.

The only difference between the recursive calls is that in the case where the current item is in the intersection, the insertion position is advanced to the children of the current node.

4.1 Reporting of Closed Item Set

Finally, after all transactions have been processed, the closed frequent item sets have to be reported. This is done with the help of report function. Not every node of the prefix tree generates output. In the first place, item sets that do not reach the user-specified minimum support must be discarded. In addition, however, we must also check whether a child has the same support. If this is the case, the item set represented by a node is not closed and must not be reported. In order to take care of this, the function first traverses the children and determines their maximum support. Only if the nodes own support exceeds this maximum (and thus we know that the item set represented by it is closed), it is reported.

4.2 Item and Transaction Orders

It is usually most efficient to assign the item codes with respect to ascending frequency in the database (the rarest item has code 0, the next code 1 etc.) and to process the transactions in the order of increasing size (number of contained items). The order of transactions of the same size seems to have very little influence. We use a lexicographical order of the transactions based on a descending order of items in each transaction. An intuitive explanation why this scheme is fastest is that it manages to have few and small closed item sets and thus small prefix trees at the beginning, so that many transactions can be processed fast. With the reverse

processing order for the transactions the prefix tree becomes fairly large already after few transactions, which slows down the processing for all later transactions [8].

V. Conclusions

According to the studies made the size of prefix tree grows larger and thus making its handling difficult and this lead to more memory to be used. Smaller prefix tree take less time in searching for any particular node and also makes its handling easier. The main fact behind the reduction in size is that items with higher initial support value have more probability of occurring in intersection. Arranging the items in different order of support value could allow the higher support items to exit at the first level in prefix tree. These higher support items exist more often in intersections and while we search for intersecting items in the tree they can be found at first level of prefix tree thus it could prevents extra branches to be added in the tree and thus the number of branches in the prefix tree reduces.Further the experimental results could be carried out in future to predict correct ordering of items in prefix tree and further compact the size of prefix tree and reduce its complexity.

References

[1] T. Uno, M. Kiyomi, and H. Arimura. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In Proc. Workshop Frequent Item Set Mining Implementations (FIMI 2004, Brighton, UK), Aachen, Germany, 2004. CEUR Workshop Proceedings 126.

[2] G. Grahne and J. Zhu. Reducing the main memoryconsumptions of FPmax* and FPclose. In Proc. Workshop Frequent Item Set Mining Implementations (FIMI 2004, Brighton, UK), Aachen, Germany, 2004. CEUR Workshop Proceedings 126.

[3] Calders T, Garboni C, Goethals B (2010) Efficient pattern mining of uncertain data with sampling. In: Proceedings of the 14th Pacific-Asia conference on knowledge discovery and data mining (PAKDD 2010, Hyderabad, India), vol I. Springer, Berlin, pp 480–487

[4] B. Goethals and M. Zaki, editors. Proc. Workshop Frequent Item Set Mining Implementations (FIMI 2004, Brighton, UK), Aachen, Germany, 2004. CEUR Workshop Proceedings 126.

[5] C. Borgelt and X. Wang. SaM: A split and merge algorithm for fuzzy frequent item set mining. In Proc. 13th Int. Fuzzy Systems Association World Congress and 6th Conf. of the European Society for Fuzzy Logic and Technology (IFSA/EUSFLAT'09, Lisbon, Portugal), Lisbon, Portugal, 2009. IFSA/EUSFLAT Organization Committee.

[6] F. Pan, G. Cong, A. Tung, J. Yang, and M. Zaki. Carpenter: Finding closed patterns in long biological datasets. In Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2003, Washington, DC), pages 637{642, New York, NY, USA, 2003. ACM Press.

[7] F. Pan, A. Tung, G. Cong, and X. Xu. Cobbler: Combining column and row enumeration for closed pattern discovery. In Proc. 16th Int. Conf. on Scienti_c and Statistical Database Management (SSDBM 2004, Santori Island, Greece), page 21_, Piscataway, NJ, USA, 2004. IEEE Press.

[8] G. Cong, K.-I. Tan, A. Tung, and F. Pan. Mining frequent closed patterns in microarray data. In Proc. 4th IEEE International Conference on Data Mining (ICDM 2004, Brighton, UK), pages 363{366, Piscataway, NJ, USA, 2004. IEEE Press.