An Adaptive Algorithm for Incremental Mining of Association Rules

N.L. Sarda

N. V. Srinivas

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai, India

nls@cse.iitb.ernet.in

Abstract

The association rules represent an important class of knowledge that can be discovered from data warehouses. Current research efforts are focused on inventing efficient ways of discovering these rules from large databases. As databases grow, the discovered rules need to be verified and new rules need to be added to the knowledge base. Since mining afresh every time the database grows is inefficient, algorithms for incremental mining are being investigated. Their primary aim is to avoid or minimize scans of the older database by using the intermediate data constructed during the earlier mining. In this paper, we present one such algorithm. We make use of large and candidate itemsets and their counts in the older database, and scan the increment to find which rules continue to prevail and which ones fail in the merged database. We are also able to find new rules for the incremental and updated database. The algorithm is adaptive in nature, as it infers the nature of the increment and avoids altogether, if possible, multiple scans of the incremental database. Another salient feature is that it does not need multiple scans of the older database. We also indicate some results on its performance against synthetic data.

1. Introduction

Data mining, which is also referred to as knowledge discovery in databases, is a process of nontrivial extraction of implicit, previously unknown, and potentially useful information from data in a database. Data mining has recently attracted considerable attention from database user community as they realize that this information, locked inside the large organizational databases built over many years, can provide information and knowledge for enhancing their organization's effectiveness and competitiveness. The process of data mining provides knowledge in the form of rules and patterns based on statistical analysis of data. The process is challenging because the source databases from which the knowledge is extracted are large and growing. The knowledge itself is time-varying, as some rules and patterns may hold now but not in future, or vice-versa. The mining techniques must scale well to handle very large and growing databases, and should permit efficient maintenance of extracted knowledge.

One of the most studied data mining problems is mining for association rules. Given a collection of items and a set of records (i.e, transactions), each of which contain some number of items from the given collection, the association rules indicate affinities that exist among the collection of items. These affinities can be expressed by rules such as "62 % of all the records that contain items A, B and C also contain items D and E." The specific percentage of occurrences is called the confidence factor of the rule. A database may throw up a very large number of association rules.

Much work has been done in the field of finding association rules [1] [2] [3] [8] [6]. These efforts are directed at devising algorithms to mine the rules efficiently in large databases. They commonly require multiple scans of the given database. As databases grow over time, there is a need to undertake mining again for maintaining (i.e., verifying) rules discovered earlier and also for discovering new rules. However, it has been realized that applying the proposed algorithms on the updated database (i.e, the older and the incremental database together) may be too costly. Researchers are now investigating ways by which rule maintenance can be done by processing the incremental part separately, and scanning the older database only if necessary. To achieve this, the incremental mining algorithms generally plan to use intermediate information collected during earlier mining process. This strategy for mining is also re-

ferred to as 'incremental' mining. Comparatively, there is not enough work in the field of incremental mining [4] [5] [7].

Maintaining sets of discovered rules poses several challenges because the underlying data changes over time and therefore the current rules may become invalid while new rules may exist undetected. New rules may be valid only for the incremental database or for the whole database. If the purpose of the incremental mining is just to validate the existing rules, then it is a trivial process which can be done in one pass of the incremental database (we just have to count the support of all the large itemsets of the original database in the incremental database). However, there is always a need for finding new rules both in the increment and the updated database. Since mining for association rules is costly, we would like to somehow use the information that was found during the earlier mining process, both to maintain and generate the rules.

An Adaptive algorithm is proposed in this paper that makes effective use of the information discovered during the earlier process of mining. Moreover, the knowledge of the *type* of incremental database is also used by the algorithm for improving its performance further. Experiments have been conducted to study the performance of the Adaptive algorithm.

The remaining of the paper is organized as follows. A brief review of the related work is given in Section 2. Different types of incremental data are identified in Section 3. Section 4 contains description of the proposed Adaptive algorithm. The test data generation and performance studies are given in Section 5. Finally, Section 6 contains concluding remarks.

2. Background and Related Work

The task of mining association rules [2] can be formally stated as follows : Let $I = \{ i_1, i_2, \ldots, i_m \}$ be a set of literals, called items. Let *D* be a set of transactions where each transaction *T* is a set of items such that $T \in I$. We say that a transaction *T* contains *X*, a set of some items in *I*, if $X \in T$. An *association rule* is an implication of the form $X \Rightarrow Y$, where $X \subset I$, $Y \subset I$ and $X \cap Y = \phi$. The rule $X \Rightarrow Y$ holds in the transaction set *D* with *confidence c* if *c %* of transactions in *D* that contain X also contain Y. The rule $X \Rightarrow Y$ has *support s* in the transaction set *D* if *s %* of the transactions in *D* contain $X \cup Y$. The problem of mining association rules is to generate all association rules that have support and confidence greater than the user-specified minimum support and minimum confidence respectively.

In all the different algorithms given in the literature, the problem of discovering association rules is decomposed into two subproblems [2].

- Find all sets of items (*itemsets*) that have support above the minimum support. The *support* for an itemset is the number of transactions that contain the itemset. Itemsets with minimum support are called *large* itemsets.
- Use the large itemsets to generate the desired rules. For every large itemset *l*, find all non-empty subsets of *l*. For every such subset *a*, output a rule of the form $a \Rightarrow$ (*l*-*a*) if the ratio of support (*l*) to support (*a*) is at least *minconf*.

Since it is easy to generate association rules if the large itemsets are available, major research efforts have been focused on finding efficient algorithms to compute large itemsets.

An itemset is called *k-itemset* if the number of items in the itemset is *k*. We denote as L_k the set of large k-itemsets. Candidate itemsets are those which are potentially large itemsets. Let C_k denote the set of candidate k-itemsets.

The Apriori algorithm [3] makes multiple passes over the database to find all large itemsets. In the first pass, the algorithm simply counts item occurrences to determine large 1-itemsets. A subsequent pass, say pass k, consists of two phases. First, the large itemsets L_{k-1} found in the (k-1)th pass are used to generate the candidate itemsets C_k, using the *apriorigen()* function. This function first joins L_{k-1} with itself, the joining condition being that the lexicographically ordered first k-2 items are the same. Next, it deletes all those itemsets from the join result that have some (k-1) subset that is not in L_{k-1}, yielding C_k. The algorithm now scans the database. For each transaction, it determines which of the candidates in C_k are contained in the transaction and increments the count of those candidates. To efficiently determine which of the candidates in C_k are contained in a transaction, it uses a data structure called *hash tree*.

Toivonen [8] has proposed a sampling based algorithm for mining association rules. They take a random sample of the database, find large itemsets, and then verify the results with the whole database. They make use of *negative border* to find candidate itemsets. Since we also use this concept, we will define it below.

Negative border [8] : Given R as a set of items and a collection $S \subseteq P(R)$ of sets (P(R) represents the powerset of R), closed with respect to the set inclusion relation, the negative border $B^-(S)$ of S consists of the minimal itemsets $X \subseteq R$ not in S. For example, let R = A,B, . . ., F and assume the collection S of frequent itemsets (in some relation *r* and some specified minimum support *s*) is {A}, {B}, {C}, {F}, {A,B}, {A,C}, {A,F}, {C,F}, {A,C,F}. The negative border for the above collection $B^-(S)$ = { {B,C}, {B,F}, {D}, {E} }. The intuition behind the concept of negative border is that given a closed collection *S* of sets that are frequent, the negative border contains the "closest" itemsets

that could be frequent, too.

Let *DB* be the original database and *db* be the incremental database. The problem of incremental mining is to find the set L_{DB+db} of large itemsets in the updated database *DB* + *db*. We may also be interested in finding the set L_{db} of large itemsets in *db*.

In [4], Cheung et al describe the Fast Update Algorithm, FUP, for incrementally maintaining association rules from large databases. Basically, the framework of FUP is similar to that of Apriori. It contains a number of iterations. We will briefly review FUP algorithm so that we can compare it with the algorithm proposed in this paper.

Let L_k denote the set of all size-*k* large itemsets in *DB*, and L'_k represent the set of all large *k*-itemsets in $DB \cup db$. C_k is the set of size-*k* candidate itemsets in the *k*-th iteration of FUP. Let $X.support_D$, $X.support_d$ and $X.support_{UD}$ denote support counts of an itemset *X* in *DB*, *db* and $DB \cup db$, respectively. The main steps in the FUP algorithm are :

1. Scan *db* for all itemsets $X \in L_1$, and update its support count $X.support_{UD}$. Remove those itemsets X for whom the condition $X.support_{UD} < s \times (D + d)$ is true. In the same scan, the candidate set C_1 is created to store all size-one itemsets which are not in L_1. Their support in *db* is also found in the same scan. The itemsets in C_1 whose $X.support_d$ is less then $s \times db$ are deleted as they cannot be large in the updated database.

2. Scan *DB* to update the support of each $X \in C_1$. By checking their support count, new large itemsets from C_1 are found. By combining these with those identified in L_1, the set of all size-one large itemsets, L'_1, is generated.

3. The above steps are repeated for all the later iterations until no large itemset is found. In iteration *k*, C_k is generated by first using the join-based apriori-gen function [3] on L'_{k-1} and then pruning C_k by removing those itemsets which are already present in L_k.

Although FUP algorithm makes multiple scans of db and DB, it achieves significant efficiency because it avoids recomputations for itemsets which were already found large during miming of DB, and it significantly reduces number of new candidate sets through a simple pruning strategy. It also prunes the given databases by removing transactions containing items that do not have a chance of appearing in large itemsets.

The performance studies on the FUP algorithm [4] show significant improvement compared to the association algorithms applied to the updated database afresh. For small support levels, the performance improvement is 3 to 6 times, and for large itemsets, it is 2 to 3 times. The studies also show that there is 98% to 95% reduction in number of candidate itemsets at each iteration. Another observation is that the performance gain reduces when the increment db is large in size. In [5], the FUP algorithm has been extended to handle insertions as well as deletions from the original database.

A temporal windowing technique for incremental maintenance of association rules is proposed in [7]. Their approach is based on the premise that transactions outside a user-defined *time window* are too old to contribute towards association rules of current interest. They also define a *strong support threshold* and *near strong threshold* levels (and also the corresponding strong and near strong confidence levels) for mining strong and near-strong association rules. These near-strong rules have the potential to become strong association rules during the next time window. Consequently, their update algorithm *retires* old and outdated transactions and carries out mining using the incremental database only. As they note in their conclusions, the time window and the deviations from strong support (for defining near string support) may be hard to define. Their technique, where applicable, can lead to substantial performance improvement in maintaining the contemporary association rules.

3. Types of Incremental databases

The database used in mining for knowledge discovery is dynamic in nature. The existing data may be updated and new transactions may be added over time (we will, however, assume, that the database represents history, and hence changes are of append type only). The knowledge discovered from these databases is also dynamic. The changes represent changing policies of the organization (which,in fact, may be based on knowledge obtained from past mining), or changes in behavior of the customers, or due to seasonal nature of demands. The objective of incremental mining is to avoid re-learning of rules from the old data [7] and utilize knowledge that has already been discovered.

The new transactions, representing new happenings in the organization, are accumulated over a period of time. The incremental database db, collected during time period p, is periodically merged with the original database DB, representing history up to time T. The period p will be chosen such that sufficient number of new transactions are accumulated to indicate changing patterns, if any.

Let A_T and A_{T+p} represent rules discovered from DB and the updated database DB+db up to time T and T+p for a given level of support and confidence. A_{T+p} may contain some new rules (called *winners*). Some rules from A_T, called *losers*, may be absent in A_{T+p}. However, we must note that the increment db may itself contain some interesting patterns, which may be a consequence of new business strategies or seasonal customer demands. We also need to find the rules A_p which hold only in the increment. Note

that these may be missed in A_{T+p}.

Thus, for rulebase maintenance, we need an approach where both the increment database db and the updated database DB+db are mined. Our objective in mining is still to avoid multiple scans of the two databases by utilizing intermediate information obtained in the earlier mining of DB.

The cost of maintaining the rules can be reduced considerably once we know the type of incremental database being mined. We visualize the following types of incremental databases :

1. The incremental database can be considered as a sample of the of the original database. In this case, we do not expect any significant differences between the patterns in the original and the updated databases. An example of such a database is the Treatment database describing drugs used for curing diseases. Here, changes in treatments are infrequent.

2. The incremental database has more or less similar patterns to that of the original database. Essentially the original patterns may still exist but there may be a few new patterns. An example could be the Point of Sale database. During the mining of the original database, we may have observed a few associations. Subsequently, we may have laid out new strategies. These might have resulted in new associations. We expect the updated database to reflect them.

3. The incremental database may not necessarily be a good sample of the original database. The patterns found in the incremental and the original database may be entirely different. An example of such an incremental database is one containing patterns of seasonal nature.

In the next Section, we will describe an algorithm for incremental mining which adapts itself for better performance by observing the type of incremental data that it is mining.

4. Adaptive Algorithm

4.1. Definitions and Notations

- The set of candidate itemsets in the original database is denoted by $\mathcal{C}^{DB}$ where $\mathcal{C}^{DB} = \bigcup_{i=1}^{i=n} C_i^{DB}$.
- The set of large itemsets in the original database is denoted by $\mathcal{L}^{DB}$ where $\mathcal{L}^{DB} = \bigcup_{i=1}^{i=n} L_i^{DB}$.
- The set of itemsets which are candidate itemsets but not large in the original database is denoted by $\mathcal{C}\_\mathcal{L}$.
- Set of itemsets which are large in the incremental database is denoted by $\mathcal{L}^{db}$ where $\mathcal{L}^{db} = \bigcup_{i=1}^{i=p} L_i^{db}$.

4.2. Approach

Initially, we assume the incremental database to be a good sample of the original database (i.e., the incremental database is of type 1). We start counting the support of all the candidate itemsets, $\mathcal{C}^{DB}$, of the original database. $\mathcal{C}^{DB}$ is obtained by combining the large itemsets in DB with its negative border. After counting the support of the itemsets in $\mathcal{C}^{DB}$, which gives us $\mathcal{L}^{db}$, the algorithm applies the following rules to find out whether more scans of the incremental database are required.

- **Rule 1 :** If the set of large itemsets in the incremental database, $\mathcal{L}^{db}$, is a subset of $\mathcal{L}^{DB}$, then no more scans of either the incremental database or the original database is required.

- **Rule 2 :** If any of the local large itemsets, $X \in \mathcal{L}^{db}$, is in $\mathcal{C}\_\mathcal{L}$ then there is a possibility that we have missed out some local large itemsets. In this case we have to scan the incremental database again.

Let us look at an example: Let us suppose {A}, {B}, {C}, {D}, {E}, {AB}, {AC}, {BC}, {ABC} are the candidate itemsets in the original database of which {A}, {B}, {C}, {AB}, {AC} and {BC} are large in the original database. As noted above, the candidate itemsets include large itemsets and the negative border. Now, if the the set of local large itemsets $\mathcal{L}^{db}$ is a subset of $\mathcal{L}^{DB}$, then we don't have to scan anymore, either the incremental database or the original database. But, instead, if {D} turns out to be large in the incremental database, then there is a possibility that we have missed out some large itemsets like {AD}, {BD}, {CD} etc. To find out the missed local large itemsets, we have to rescan the incremental database.

This is the time we can infer the nature of incremental database. If the local large itemsets belonging to $\mathcal{C}\_\mathcal{L}$ are not too many, we can generate all possible itemsets which could be large in the incremental database and count their support in one more scan. Thus, we may totally require two scans of the incremental database. For instance, in the above example, if only D turns out to be the local large itemset, then we will count the support of the following in the next scan: {AD}, {BD}, {CD}, {ABD}, {ACD}, and {BCD}. Instead, if the local large itemsets belonging to $\mathcal{C}\_\mathcal{L}$ are plenty in number, then we switch to level wise algorithm to mine the incremental database for remaining large itemsets. For example, in the above case, if both {D} and {E} turn out to be local large itemset, and the candidate itemsets generated by these itemsets is above some threshold, then we switch to the level wise algorithm; i.e., in the kth pass (or, scan) of the incremental database, support for candidate k itemsets is counted. (Here, we can optimize by not counting candidate k itemsets whose support has already been counted.) Thus,

in the second pass of the incremental database, support for {AD}, {BD}, {CD}, {AE}, {BE}, {CE}, {DE} is counted. If {AD} and {BD} turn out to be large locally then, in the third pass, support for {ABD} is counted and so on.

4.3. Algorithm

Based on the above discussion, the adaptive algorithm is presented as follows.

Input : (1) DB : the original database of size D, (2) L_k^{DB}: the set of all large *k*-itemsets in DB, where *k* = 1, ..., *r*, and N_k, the negative border of L_k^{DB}; (3) *db* : an increment database of size o *d*; (4) *s* : the minimum support threshold, and (5) *z* : threshold on number of new candidate itemsets.

Output : (1) The set of large-itemsets which are large when both the original and the incremental databases are taken into consideration. (2) The set of large-itemsets which are large only in the incremental database.

Steps

1. Build level-wise hash trees for $\mathcal{C}^{DB}$, ie, for large itemsets in DB and their negative border.
2. Count support for itemsets in hash trees in the incremental database db.
3. Find those which are large, ie, $\mathcal{L}^{db}$.
4. If $\mathcal{L}^{db} \subseteq \mathcal{L}^{DB}$ then output $\mathcal{L}^{db}$ as no new large itemsets found in db, and exit.
5. Generate candidate itemsets upto the threshold *z*, build hash trees for them, and in one scan of db, obtain large itemsets in db, ie, $\mathcal{L}^{db}$. Output these and go to step 7 if the threshold was not crossed.
6. Starting from the level at which we stopped in the above step, for each level until no new large itemsets are found, compute candidate itemsets using *apriori-gen()*, scan db, and obtain large itemsets. We obtain $\mathcal{L}^{db}$ in multiple scans of db. Output these.
7. Scan DB to get further counts of itemsets in $\mathcal{L}^{db}$ to find new large itemsets in the combined database.

5. Comparison and Performance Studies

In Section 2, a technique proposed by Cheung et al [4] for incrementally maintaining association rules was described. Their approach also utilizes what has already been learnt about large itemsets to reduce the cost of updating the discovered rules. However, in their approach, all the rules that are found are those which are applicable when both the original and the incremental databases are taken to-gether. In our approach, apart from finding such rules, we also find rules which are applicable in the incremental database alone. Moreover, we achieve both the tasks in (generally) two scans of the incremental database and only one scan of the original database. Note that we need only one scan of the incremental database when the increment has similar patterns as the original database.

Let $\mathcal{L}$ be the set of large itemsets in the time period $T + t_0$ and *l* be the set of large itemsets in the time period t_0 alone. The set *l* of large itemsets can be a result of new business strategies. The set l may continue to be large in the time period $t_1, t_2, \ldots$. In this case, the set *l* will remain undetected by Cheung et al's approach because the itemsets in *l* do not have enough support when both the original and the incremental database are taken into consideration.

The advantages of our approach when compared to the one proposed by Cheung et al [4] are as follows :

- Since the process of finding rules in the updated database is separated from finding rules that are applicable for incremental database alone, we can postpone the scan of the original database.
- Each rule can be time-stamped with the interval of the incremental database to indicate its temporal validity.
- When we decide to retire some old data, we do not have to re-learn the rules afresh as we have the rules valid for the incremental databases.

5.1. Generation of test data

The original database is generated using the same technique introduced in [3]. The incremental database can be generated in the following different ways :

1. A random sample of the original database is taken.
2. A set of itemsets, *S*, is chosen as *would-be-large-itemsets*. The itemsets in *S* are not large in DB but it is intended to make these itemsets large in db. After choosing *S*, a sample of transactions db is taken from DB. To make the itemsets in *S* large in db, items present in *S* are added to the transactions in db with some probability (that depends on how many of these items a transaction already contains).
3. A database of size (*D* + *d*) is first generated using the technique given in [3]. The first *D* transactions are used as DB and the next *d* transactions form db.

5.2. Performance of the Adaptive algorithm

Experiments have been performed with the original database of about 70,000 transactions and incremental databases of size 1k, 5k, 10k. The performance of the adaptive algorithm was compared with the Apriori algorithm

[3]. The Apriori was run on the database of (*D+d*) transactions and also on *d* transactions. The time taken in the two cases was added up. Since we are finding large itemsets in (*D+d*) as well as in *d*, the above comparison is justified. The *Speedup ratio S* was calculated as follows :

$$S = \frac{(time\ for\ Apriori_{D+d} + time\ for\ Apriori_d)}{time\ for\ Adaptive_{D+d}}$$

In our experiments, the speed-up ratios ranged from 4.2 to 7.6 when the test data was generated using method 1 above, and from 3.0 to 6.3 when method 2 was used.

It was observed in the experiments that the size of the largest itemset was 7 while the number of scans of the incremental database required was only 4. The number of scans of the original database was 0 or 1. The speed-up ratios obtained in our simulation results may appear lower than those reported for the FUP [4] algorithm. The reason could be the nature of test data and the details of coding. We also note that we did not include pruning of the databases in our algorithm. We stress here that since we do not scan original database more than once, and that only the incremental database is scanned (generally twice only; it will be scanned multiple times only when it is significantly different from the original database in the rules it contains), we expect to have better performance for the adaptive algorithm.

6. Conclusions

The real-world databases, from which useful patterns and rules are mined, are dynamic in nature. Periodically, the organizational database is updated, and it may become necessary to carry out the mining process again on the updated database. Since mining is a costly activity, typically requiring multiple database scans, process of incremental mining is proposed by researchers to maintain the rules discovered during previous mining processes.

In this paper, we have proposed an adaptive algorithm for incremental mining. We have tried to categorize different types of increments. Primarily, the increments could be representing similar business trends as before or significantly different trends. The algorithm adapts itself by first finding the nature of the increment. It then decides whether to scan the original database for updating the rules which were obtained in earlier mining processes. The algorithm not only updates rules discovered from the original database, but also mines rules which may be present in the increment alone. These rules may be due to new business decisions, or due to changing customer preferences, or seasonal trends. It is important to extract them explicitly as new trends to help in business decisions. It must be noted that these rules found from the increment alone may not have the required support in the updated database.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. In *IEEE Trans. Knowledge and Data Engineering*, 1993.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databas es. In *Proc. ACM SIGMOD*, 1993.

[3] R. Agrawal and R. Srikanth. Fast algorithms for mining association rules in large databases. In *Proc. 20th Int'l Conf. Very Large Data Bases*, 1994.

[4] David W. Cheung, Jiawei Han, Vincent T. Ng, and C.Y.Wong. Maintenance of discovered association rules in large databases: An incremental updating approach. In *12th IEEE International Conference on Data Engineering*, 1996.

[5] David W. Cheung, S.D. Lee, and B. Kao. A general incremental technique for maintaining discovered association rules. In *Proceedings of Database Systems for Advanced Applications, DASFAA'97*, Melbourne, Australia, pp. 185-194, 1997.

[6] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of ACM SIGMOD conference*, San Jose, California, 1995.

[7] Chris P. Rainsford, Mukesh K. Mohania, and John F. Roddick. A temporal windowing technique for the incremental maintenance of association rules. In *8th International Database Workshop, Data Mining, Data Ware housing and Client/Server Databases*, 1997.

[8] H. Toivonen. Sampling large databases for association rules. In *Proceedings of Very Large Data Bases Conference*, Mumbai (India), pp. 134-145, 1996.