A Comparative Study of Iterative Algorithms in Association Rules Mining

Robert S. Győrödi

Department of Computer Science,

Faculty of Electrotehnics and Informatics,

University of Oradea,

ROMANIA

rgyorodi@rdsor.ro

Abstract: Data mining (also called knowledge discovery in databases) represents the process of extracting interesting and previously unknown knowledge (patterns) from data. An association rule expresses the dependence of a set of attribute-value pairs, called items, upon another set of items (itemset). This paper presents a comparative study of the most important iterative algorithms used in association rules mining. A performance study that shows the advantages and disadvantage of the iterative algorithms Apriori, Sampling and Partitioning is also presented.

Keywords: Data Mining, Iterative Association Rules Mining Algorithms, Apriori, Sampling, Partitioning, Comparison Framework

Robert S. Győrödi graduated in Computer Engineering the Polytechnic University of Timisoara in 1994. He is a PhD in *Computer Science*. His main research interests are in: combination of image processing, artificial intelligence, neural networks and database management systems in providing a safer operating experience within existing and future operating systems.

He participated in different TEMPUS financed projects and had many collaborations with private companies, most notably with the European Drinks Group, coordinating their IT projects. He is author/co-author of 3 books, and of over 30 published papers, many of them abroad. Mr. Győrödi is a lecturer at University of Oradea, Romania and also University of Limerick, Ireland.

1. Introduction

In recent years the continued growth in the size of databases has led to an increased interest in the automatic extraction of knowledge from data.

The term Data Mining, or Knowledge Discovery in Databases, has been adopted for a field of research dealing with the automatic discovery of implicit information or knowledge within databases [7]. High performance is a key factor for data mining systems in order to allow a sufficient exploration of data and to cope with the huge amounts of data. Different architectures for Data Mining were studied in [4].

The implicit information within databases, and mainly the interesting association relationships among sets of objects, that lead to association rules, may disclose useful patterns for decision support, financial forecast, marketing policies, even medical diagnosis and many other applications. This fact has attracted a lot of attention in recent data mining research [7]. Many authors focussed their work on efficient mining of association rules in databases [1, 2, 6, 7, 8].

A very influential association rule mining algorithm, Apriori [1], has been developed for rule mining in large transaction databases. Many other algorithms developed are derivative and/or extensions of this algorithm such as Sampling and Partitioning algorithms.

Recently alternative data structures were employed in order to improve on the efficiency of existing and new algorithms [10, 11]. In order to be able to compare new algorithms with existing algorithms a suitable comparison framework must be established. This paper presents a comparative study of three of the most important iterative algorithms used in association rules mining. This study will be further used and referenced in future works on developing and extending new and existing association rules mining algorithms.

The paper is organised as follows. Section II presents the problem definition. Section III presents the main aspects of Apriori algorithm and shown Sampling and Partitioning algorithms. Section IV shows a comparative study of the algorithms. Section V presents the results and conclusions.

2. Problem Definition

Association rule mining finds interesting association or correlation relationships among a large set of data items [8]. The association rules are considered interesting if they satisfy both a *minimum support* threshold and a *minimum confidence* threshold [3].

A more formal definition is the following [4]. Let $\Im = \{i_1, i_2, \ldots, i_m\}$ be a set of items. Let D, the task-relevant data, be a set of database transactions where each transaction T is a set of items such that $T \subseteq \Im$. Each transaction is associated with an identifier, called TID. Let A be a set of items. A transaction T is said to contain A if and only if $A \subseteq T$. An association rule is implication of the form $A \Rightarrow B$, where $A \subset \Im$, $B \subset \Im$, and $A \cap B = \varnothing$. The rule $A \Rightarrow B$ holds in the transaction set D with *support* s, where s is the percentage of transactions in D that contain $A \cup B$ (i.e., both A and B). This is taken to be the probability, $P(A \cup B)$. The rule $A \Rightarrow B$ has *confidence* c in the transaction set D if c is the percentage of transactions in D containing A that also contain B. This is taken to be the conditional probability, $P(B|A)$. That is,

$$support(A \Rightarrow B) = P(A \cup B) \quad (1)$$

$$confidence(A \Rightarrow B) = P(B \mid A) \quad (2)$$

The definition of a frequent pattern relies on the following considerations [5]. A set of items is referred to as an itemset (pattern). An itemset that contains k items is a k-itemset. The set {*name, semester*} is a 2-itemset. The occurrence frequency of an itemset is the number of transactions that contain the itemset. This is also known, simply, as the frequency, support count, or count of itemset. An itemset satisfies *minimum support* if the occurrence frequency of the itemset is greater than or equal to the product of *minimum support* and the total number of transactions in D. The number of transactions required for the itemset to satisfy *minimum support* is therefore referred to as the *minimum support count*. If an itemset satisfies *minimum support*, then it is a *frequent itemset* (*frequent pattern*).

The most common approach to finding association rules is to break up the problem into two parts [6]:

1. Find all frequent itemsets: By definition, each of these itemsets will occur at least as frequently as a pre-determined minimum support count [8].
2. Generate strong association rules from the frequent itemsets: By definition, these rules must satisfy minimum support and minimum confidence [8].

Additional interesting measures can be applied, if desired. The second step is the easier of the two. The overall performance of mining association rules is determined by the first step. As shown in [2], the performance, for large databases, is most influenced by the combinatorial explosion of the number of possible frequent itemsets that must be considered and also by the number of database scans that has to be performed.

3. Iterative Algorithms in Association Rules Mining

3.1. The Apriori Algorithm

Figure 1 gives an overview of the Apriori algorithm for finding all frequent itemsets, using the notation in Table 1.

Table 1. Notations used in the algorithms

Notation	Description
k-itemset	An itemset having k items.
L_k	Set of large k-itemsets (those with minimum support). Each member of this set has two fields: 1) itemset and 2) support count.
C_k	Set of candidate k-itemsets (potentially large itemsets). Each member of this set has two fields: 1) itemset and 2) support count.
$\overline{C_k}$	Set of candidate k-itemsets when the TIDs of the generating transactions are kept associated with the candidates.

The first pass of the algorithm simply counts item occurrences to determine the large 1-itemsets. A subsequent pass, say pass k, consists of two phases. First, the large itemsets L_{k-1} found in the $(k\text{-}1)^{th}$ pass are used to generate the candidate itemsets C_k, using the Apriori candidate generation function (apriori-gen) described below. Next, the database is scanned and the support of candidates in C_k is counted. For fast counting, we need to efficiently determine the candidates in C_k that are contained in a given transaction t. A hash-tree data structure [1] is used for this purpose.

```
L1 = {large 1-itemsets};
for ( k = 2; Lk-1 <> Ø; k++ ) do begin
  Ck = apriori-gen(Lk-1);      //New
candidates
  forall transactions t ∈ D do begin
    Ct = subset(Ck, t);        //Candidates
contained in t
    forall candidates c ∈ Ct do
      c.count++;
    end
    Lk = { c ∈ Ck | c.count >= minsup }
end
Answer = ⋃k Lk;
```

Figure 1. The Apriori Algorithm

Apriori Candidate Generation: The *apriori-gen* function takes as argument L_{k-1}, the set of all large (k-1)-itemsets. It returns a superset of the set of all large k-itemsets. The function works as follows. First, in the *join* step, we join L_{k-1} with L_{k-1}:

insert into C_k

select $p.\text{item}_1, p.\text{item}_2, \ldots, p.\text{item}_{k-1}, q.\text{item}_{k-1}$

from $L_{k-1}\ p, L_{k-1}\ q$

where $p.\text{item}_1=q.\text{item}_1, \ldots, p.\text{item}_{k-2}=q.\text{item}_{k-2},$

$p.\text{item}_{k-1} < p.\text{item}_{k-1}$;

Next, in the *prune* step, all itemsets $c \in C_k$ such that some (k-1)-subset of c is not in L_{k-1} are deleted:

forall itemsets $c \in C_k$ do

forall (k-1)-subset s of c do

if ($s \notin L_{k-1}$) then

delete c from C_k;

The subset function is based on the fact that the items in any itemset are ordered, and is described in detail in [1].

3.2. The Sampling Algorithm

To facilitate efficient counting of itemsets with large databases, sampling of the database may be used. The original sampling algorithm reduces the number of database scans to one in the best case and two in the worst case. The database sample is drawn so that it can be memory-resident. Then any algorithm, such as Apriori, is used to find the large itemsets for the sample. These are viewed as *potentially large* (*PL*) itemsets and used as candidates to be counted using the entire database. Additional candidates are determined by applying the *negative border* function, BD^-, against the large itemsets from the sample. The entire set of candidates is then $C = BD^-(PL) \cup PL$. The negative border function is a generalization of the Apriori-Gen algorithm. It is defined as the minimal set of itemsets that are not in *PL*, but whose subsets are all in *PL*.

Figure 2 shows the sampling algorithm [6]. Here the Apriori algorithm is shown to find the large itemsets in the sample, but any large itemset algorithm could be used. Any algorithm to obtain a sample of the database could be used as well. The set of large itemsets is used as a set of candidates during a scan of the entire database. If an itemset is large in a sample, it is viewed to be potentially large in the entire database. Thus, the set of large itemsets from the sample is called *PL*. in an attempt to obtain all the large itemsets during the first scan, however, *PL*, is expanded by its negative border. So the total set of candidates is viewed to be $C = BD^-(PL) \cup PL$.

During the first scan of the database, all candidates in C are counted. If all candidates that are large are in *PL*, then all large itemsets are found. If, however, some large itemsets are in the negative border, a second scan is needed. Think of $BD^-(PL)$ as a buffer area on the border of large itemsets. The set of all itemsets is divided into four areas: those that are known to be large, those that are known to be small, those that are on the negative border of those known to be large, and the others. The negative border is a buffer zone

between those known to be large and the others. It represents the smallest possible set of itemsets that could potentially be large. Because of the large itemset property, we know that if there are no large itemsets in this area then there can be none in the rest of the set.

Input:

I //itemsets
D //database of transactions
s //support

Output:

L //large itemsets

Algorithm:

D_s = Sample drawn from D;
PL = Apriori(I,D_s,$small_s$);
$C = BD^-(PL) \cup PL$
$L = \emptyset$;
for each $I_i \in C$ **do**
  $c_i = 0$; //initial counts for each itemset are 0;
for each $t_j \in D$ **do** //first scan count.
  for each $I_i \in C$ **do**
    if $I_i \in t_j$ **then** c_i ++;
for each $I_i \in C$ **do**
  if $c_i >= (s \times |D|)$ **do** $L = L \cup I_i$;
$ML = \{x \mid x \in BD^-(PL) \wedge x \in L\}$; //missing large itemsets.
if $ML <> \emptyset$ **then begin**
  $C = L$; //set candidates to be the large itemsets.
  repeat
    $C = C \cup BD^-(C)$ //expand candidate sets by neg border
  until no new itemsets are added to C;
  for each $I_i \in C$ **do**
    $c_i = 0$; //initial counts for each itemset are 0;
  for each $t_j \in D$ **do** //second scan count.
    for each $I_i \in C$ **do**
      if $I_i \in t_j$ **then** c_i ++;
  if $c_i >= (s \times |D|)$ **do**
    $L = L \cup I_i$;
end

Figure 2. The Sampling Algorithm

During the second scan, additional candidates are generated and counted. This is done to ensure that all large itemsets are found. Here ML, the missing large itemsets, are those in L that are not in PL. since there are some large itemsets in ML, there may be some in the rest of the set of itemsets. To find all the remaining large itemsets in the second scan, the sampling algorithm repeatedly applies the negative border function until the set of possible candidates does not grow further. While this creates a potentially large set of candidates (with many not large), it does guarantee that only one more database scan is required.

The algorithm shows that the application of the Apriori algorithm to the sample is performed using a support called $small_s$. Here $small_s$ can be any support value less than s. the idea is that by reducing the support when finding large itemsets in the sample, more of the true large itemsets from the complete database will be discovered.

3.3. The Partitioning Algorithm

Various approaches to generating large itemsets have been proposed based on a partitioning of the set of transactions. In this case, D is divided into p partitions D^1, D^2, ..., D^p.

Input:
I //itemsets
$D = \{ D^1, D^2, \ldots, D^p \}$ //divided database of transactions
s //support
Output:
L //large itemsets
Algorithm:
$C = \emptyset$;
//find large itemsets in each partition.
for $i = 1$ to p **do begin**
L^i = Apriori(I, D^i, s);
$C = C \bigcup L^i$;
end
$L = \emptyset$;
for each $I_i \in C$ **do**
$c_i = 0$; //initial counts for each itemset are 0;
// count candidates during second scan.
for each $t_j \in D$ **do**
for each $I_i \in C$ **do**
if $I_i \in t_j$ **then**
c_i ++;
for each $I_i \in C$ **do**
if $c_i >= (s \times |D|)$ **do**
$L = L \bigcup I_i$;

Figure 3. The Partitioning Algorithm

Partitioning may improve the performance of finding large itemsets in several ways [9]:

- By taking advantage of the large itemset property, we know that a large itemset must be in at least one of the partitions. This idea can help to design algorithms more efficiently than those based on looking at the entire database.
- Partitioning algorithms may be able to adapt better to limited main memory. Each partition can be created such that it fits into main memory. In addition, it would be expected that the number of itemsets to be counted per partition would be smaller than those needed for the entire database.
- By using partitioning, parallel and/or distributed algorithms can be easily created, where each partition could be handled by a separate machine.
- Incremental generation of association rules may be easier to perform by treating the current state of the database as one partition and treating the new entries as a second partition.

The basic partition algorithm reduces the number of database scans to two and divides the database into partitions such that each can be placed into main memory. When it scans the database, it brings that partition of the database into main memory and counts the items in that partition alone. During the first scan, the algorithm finds all large itemsets in each partition. Although any algorithm could be used for this purpose, the original proposal assumes that some level-wise approach, such as Apriori, is used. Here L^i represents the large itemsets from partition D^i. During the second scan, only those itemsets that are large in at least one partition are used as candidates and counted to determine if they are large across the entire database. The partitioning algorithm [9] is shown in Figure 3.

4. Comparative Study

The previous section presented the main aspects of three important iterative algorithms used in association rules mining. To be able to compare these (and other) algorithms a suitable comparison framework was established. This framework's design concentrated on its flexibility and extensibility, so that new algorithms could be anytime added and compared to existing ones. In order to achieve this, special interfaces and abstract classes were defined that provide abstracted access to different algorithms and also for different databases. The entire framework was implemented using Java 1.4.

```
public abstract class ARMiner {
  /** Used to store the minimum support of a frequent itemset in
percentage */
  private double minSupport;
  /** Used to store the minimum confidence of a rule in percentage */
  private double minConfidence;
  /** Used to store the frequent itemsets */
  protected ItemSets frequentItemSets;
  /** Used to store the generated rules */
  protected Rules rules;
  /** Default implementation of rules generation */
  public void generateRules() { ... };
  /** Declaration of the frequent itemsets mining algorithm */
  public abstract long mineFrequentItemsets();
  /* ... */
}
```

Figure 4. The abstract definition of an Association Rules Mining Algorithm

In order to evaluate the performances of the algorithms Apriori [1], Partitioning [9] and Sampling [6] and also study their performance on different databases, these were implemented according to the defined interfaces.

Other papers present reports using binary databases. In this paper, for a more realistic evaluation, databases stored on SQL servers (MS SQL 2000), accessed using standard ODBC drivers were used. Although due to the flexible design, any other database types could be used.

The machine used to perform the tests was a Pentium 4 with a 1.7GHz processor, 256 MRAM with Windows 2000 operating system.

To study the performance of the algorithms and the scalability, data sets with 10000 to 500000 transactions, were generated and support factors between 5% and 40% were used.

Table 2. Generation parameters

Parameter	Description
$\|D\|$	Number of transactions
$\|T\|$	Average size of the transactions
$\|L\|$	Number of maximal potentially large itemsets
N	Number of items

The real-life model shows that people tend to buy some items together, forming a set. Each of these sets is a potential frequent item-set. A transaction can contain more then one frequent item-set. The transactions can have a different number of items, and the dimension of the frequent item-sets can vary. Taking into account these facts we generated datasets depending on the number of items in a transaction, on the number of frequent item-sets, etc. The method used was similar to that described in [1]. The necessary parameters to generate the test dataset are described in Table 2.

The test datasets are generated setting the item number N = 100 the maximum number of frequent item-sets $|L| = 3000$. We also set the average dimension o a transaction $|T| = 10$. These transactions we generated, conforms to the transactions from retail commerce.

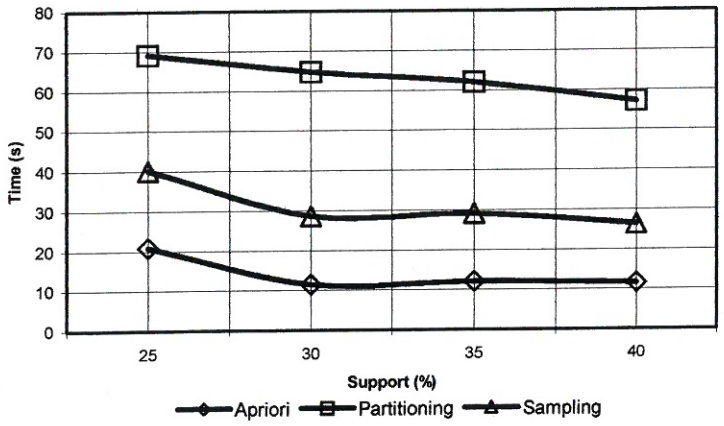

Figure 5. Scalability function of support for D1 40K database

In Figure 4 we can observe that the three algorithms have an almost constant value for the execution time when the support is greater than 30% on a database with a relatively small number of transactions (D1 = 40000), but the best performance is given by the Apriori algorithm. For a support less than 30% we notice that the execution time increases dramatically.

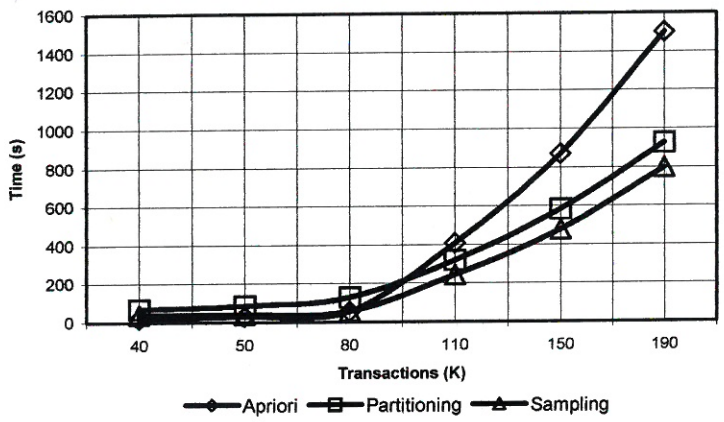

Figure 6. Scalability of the algorithms function of the number of transactions, for 25% support

The results from Figure 5 show that the Apriori algorithm has a good performance on a rather large database, 150000 transactions, only for a high value of the support (minsupport > 30%). For a value less or equal to 30% for the support, the execution time for the Apriori algorithm increases dramatically, while the Partitioning and Sampling algorithms have an execution time smaller than the Apriori algorithm for a support less than 30%. Thus, the Sampling and Partitioning algorithms have a better performance for a larger database with a small support (less than 30%).

We notice that the Apriori algorithm has a low performance for a small support, for example a support less than 10% even for small databases (50000 transactions). For a large database and a large support the performance is satisfactory. When we grow the dimension of the database the performance of the Apriori algorithm drops, even for a large support.

Figure 6 presents the scalability of the Apriori algorithm function of the number of transactions. The database has a varying number of transactions, between 10000 and 150000, and a support factor between 5% and 40%.

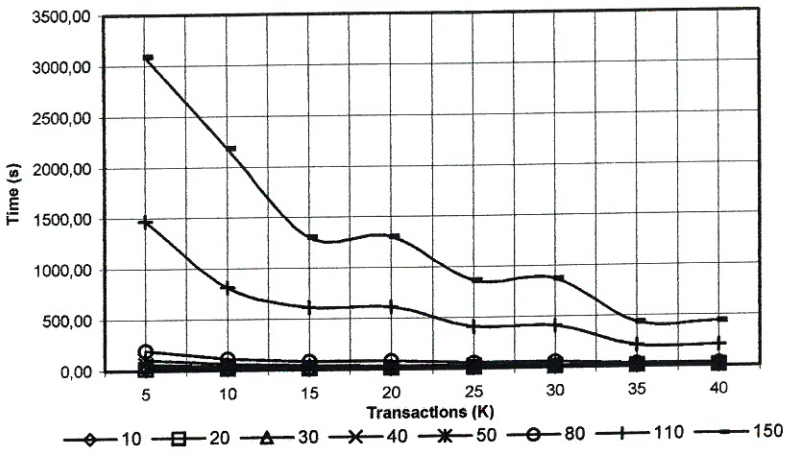

Figure 7. Scalability of the Apriori algorithm function of the number of transactions

Following these tests, the Apriori algorithm has a better performance on a database with a large number of transactions but for a large support (> 20%). For large databases with small support (< 5%) the performance of the Apriori algorithm drops dramatically, while the Sampling and Partitioning algorithms have a better performance. For example, for a database with a number of 150000 transactions and a support of 40% the execution time was 442 second, and for a support of 5% the time increased to 3098 seconds.

5. Conclusions

A comparison framework was developed to allow a flexible comparison between existing and new algorithms that conform to the defined algorithm interface. Using this framework, this paper presented a comparative study of three of the most important iterative algorithms used in association rules mining.

The Apriori algorithm has a low performance when executed on large databases with a large number of transactions. This low performance is due to the fact that the dimension of the frequent item-set is rather large. For example, for a frequent item-set with N>4, the Apriori algorithm needs N scans of the database to discover the item, this being very time consuming.

For large databases with large support (> 20%) the Apriori algorithm has a better performance than the Partitioning and Sampling algorithms, but for a small support (< 5%) the performance of the Apriori algorithm drops dramatically.

The Sampling and Partitioning algorithms reduce the number of scans of the database to two and have a better performance than the Apriori algorithm for large databases with small support (< 5%).

The Partitioning algorithm reduces the number of scans of the database to two and splits the database into partitions in order to be able to load each partition into memory. During the second scan, only the item-sets that are frequent in at least one partition are used as candidates and are counted to determine if they are frequent in the entire database, thus reducing the set of candidates.

As future work, the developed framework will be used for testing and comparing new and existing association rules mining algorithms.

REFERENCES

1. R. AGRAWAL, R. SRIKANT. **Fast algorithms for mining association rules in large databases**. Proc. of 20th Int'l conf. on VLDB: pp. 487-499, 1994.
2. J. HAN, J. PEI, Y. YIN. **Mining Frequent Patterns without Candidate Generation**. Proc. of ACM-SIGMOD, 2000.
3. C. GYORODI, R. GYORODI. **Mining Association Rules in Large Databases**. Proc. of Oradea EMES'02: pp. 45-50, Oradea, Romania, 2002.
4. R. GYORODI, C. GYORODI. **Architectures of Data Mining Systems**. Proc. of Oradea EMES'02: pp. 141-146, Oradea, Romania, 2002.

5. C. GYORODI, R. GYORODI, S. HOLBAN, M. PATER. **Mining Knowledge in Relational Databases**. Proc. of CONTI 2002, 5th International Conference on Technical Informatics: 1-6, Timisoara, Romania, 2002.

6. M. H. DUNHAM. **Data Mining. Introductory and Advanced Topics**. Prentice Hall, 2003, ISBN 0-13-088892-3.

7. U.M. FAYYAD, et al.: **From Data Mining to Knowledge Discovery: An Overview**, Advances in Knowledge Discovery and Data Mining: 1-34, AAAI Press/ MIT Press, 1996, ISBN 0-262-56097-6.

8. J. HAN, M. KAMBER, **Data Mining Concepts and Techniques**, Morgan Kaufmann Publishers, San Francisco, USA, 2001, ISBN 1558604898.

9. ASHOKA SAVASERE, EDWARD OMIENCINSKI, and SHAMKANT B. NAVATHE. **An Efficient Algorithm for Mining Association Rules in Large Databases**. In Proceedings of the 21st International Conference on Very Large Databases, pp. 432 - 444, 1995.

10. J. HAN, J. PEI, Y. YIN. **Mining Frequent Patterns without Candidate Generation**. In Proceedings of ACM-SIGMOD, 2000.

11. C. GYORODI, R. GYORODI, T. COFFEY, S. HOLBAN. **Mining Association Rules using Dynamic FP-trees**. In Proceedings of Irish Signals and Systems Conference 2003, Limerick, Ireland, pp. 76-81.