Mining Top-K Non-Redundant Association Rules

Philippe Fournier-Viger[1] and Vincent S. Tseng[2]
[1]Dept. of Computer Science, University of Moncton, Canada
[2] Dept. of Computer Science and Info. Engineering, National Cheng Kung University, Taiwan
philippe.fournier-viger@umoncton.ca, tsengsm@mail.ncku.edu.tw

Abstract. Association rule mining is a fundamental data mining task. However, depending on the choice of the thresholds, current algorithms can become very slow and generate an extremely large amount of results or generate too few results, omitting valuable information. Furthermore, it is well-known that a large proportion of association rules generated are redundant. In previous works, these two problems have been addressed separately. In this paper, we address both of them at the same time by proposing an approximate algorithm named TNR for mining top-*k* non redundant association rules.

Keywords: association rules, top-k, non-redundant rules, algorithm

1. Introduction

Association rule mining [1] consists of discovering associations between sets of items in transactions. It is one of the most important data mining tasks. It has been integrated in many commercial data mining software and has numerous applications [2].

The problem of *association rule mining* is stated as follows. Let $I = \{a_1, a_2, \ldots a_n\}$ be a finite set of items. A transaction database is a set of transactions $T=\{t_1,t_2\ldots t_m\}$ where each transaction $t_j \subseteq I$ ($1\leq j \leq m$) represents a set of items purchased by a customer at a given time. An itemset is a set of items $X \subseteq I$. The *support of an itemset* X is denoted as $sup(X)$ and is defined as the number of transactions that contain X. An association rule $X\rightarrow Y$ is a relationship between two itemsets X, Y such that X, $Y \subseteq I$ and $X\cap Y=\emptyset$. The *support of a rule* $X\rightarrow Y$ is defined as $sup(X\rightarrow Y) = sup(X\cup Y) / |T|$. The confidence of a rule $X\rightarrow Y$ is defined as $conf(X\rightarrow Y) = sup(X\cup Y) / sup(X)$. The *problem of mining association rules* [1] is to find all association rules in a database having a support no less than a user-defined threshold *minsup* and a confidence no less than a user-defined threshold *minconf*. For instance, Figure 1 shows a transaction database (left) and some association rules found for *minsup* = 0.5 and *minconf* = 0.5 (right).

Despite that much research has been done on association rule mining, an important issue that has been overlooked is how users should choose the *minsup* and *minconf* thresholds to generate a desired amount of rules [3, 4, 5, 6]. This is an important problem because in practice users have limited resources (time and storage space) for analyzing the results and thus are often only interested in discovering a certain amount of rules, and fine tuning the parameters is time-consuming. Depending on the choice of the thresholds, current algorithms can become very slow and generate an extremely large amount of results or generate none or too few results, omitting valuable information. To address this problem, it was proposed to replace the task of *association rule mining* with the task of *top-k association rules mining*, where *k* is the number of association rules to be found, and is set by the user [3, 4, 5, 6]. Several top-*k* rule mining algorithms were proposed [3, 4, 5, 6]. However, a major problem remains with these algorithms. It is that top-*k* association rules often contain a large proportion of redundant rules. i.e. rules that provide information that is redundant to the user. This problem has been confirmed in our experimental study (cf. section 4). We found that up to 83 % of top-*k* association rules are redundant for datasets commonly used in the association rule mining literature. This means that the user has to analyze a large proportion of redundant rules.

ID	Transactions
t_1	$\{a, b, c, e, f, g\}$
t_2	$\{a, b, c, d, e, f\}$
t_3	$\{a, b, e, f\}$
t_4	$\{b, f, g\}$

$\rightarrow$

ID	Rules	Support	Confidence
r_1	$\{a\} \rightarrow \{e, f\}$	0.75	1
r_2	$\{a\} \rightarrow \{c, e, f\}$	0.5	0.6
r_3	$\{a, b\} \rightarrow \{e, f\}$	0.75	1
r_4	$\{a\} \rightarrow \{c, f\}$	0.5	0.6

Fig. 1. (a) A transaction database and (b) some association rules found

The problem of redundancy in association rule mining has been studied extensively and various definitions of redundancy have been proposed [7, 8, 9, 10]. However, it remains an open challenge to combine the idea of mining a set of non-redundant rules with the idea of top-*k* association rule mining, to propose an efficient algorithm to mine top-*k* non-redundant association rules. The benefit of such an algorithm would be to present a small set of *k* non-redundant rules to the user. However, devising an algorithm to mine these rules is difficult. The reason is that eliminating redundancy cannot be performed as a post-processing step after mining the top-*k* association rules, because it would result in less than *k* rules. The process of eliminating redundancy has therefore to be integrated in the mining process.

In this paper, we undertake this challenge by proposing an approximate algorithm for mining the top-*k* non redundant association rules that we name TNR (*T*op-*k* *N*on-redundant *R*ules). It is based on a recently proposed approach for generating association rules that is named "rule expansions", and adds strategies to avoid generating redundant rules. An evaluation of the algorithm with datasets commonly used in the literature shows that TNR has excellent performance and scalability.

The rest of the paper is organized as follows. Section 2 reviews related work and presents the problem definition. Section 3 presents TNR. Section 4 presents an experimental evaluation. Section 5 presents the conclusion.

2. Related Work and Problem Definition

Top-k association rule mining. Several algorithms have been proposed for top-*k* association rule mining [3, 4, 5]. However, most of them do not use the standard definition of an association rule. For instance, KORD [3, 4] finds rules with a single item in the consequent, whereas the algorithm of You et al. [5] mines association rules from a stream instead of a transaction database. To the best of our knowledge, only TopKRules [5] discovers top-*k* association rules based on the standard definition of an association rule (with multiple items, in a transaction database). TopKRules takes as parameters *k* and *minconf*, and it returns the *k* rules with the highest support that meet the *minconf* threshold. The reason why this algorithm defines the task of mining the top k rules on the support instead of the confidence is that *minsup* is much more difficult to set than *minconf* because *minsup* depends on database characteristics that are unknown to most users, whereas *minconf* represents the minimal confidence that users want in rules and is therefore generally easy to determine. TopKRules defines the problem of top-k association rule mining as follows [5]. The *problem of top-k association rule mining* is to discover a set L containing k rules in T such that for each rule $r \in L \mid conf(r) \geq minconf$, there does not exist a rule $s \notin L \mid conf(s) \geq minconf \wedge sup(s) > sup(r)$.

Discovering Non-Redundant Association Rules. The problem of redundancy has been extensively studied in association rule mining [7, 8, 9, 10]. Researchers have proposed to mine several sets of non-redundant association rules such as the Generic Basis [7], the Informative Basis [7], the Informative and Generic Basis [8], the Minimal Generic Basis [9] and Minimum Condition Maximum Consequent Rules [10]. These rule sets can be compared based on several criteria such as their compactness, the possibility of recovering redundant rules with their properties (support and confidence), and their meaningfulness to users (see [10] for a detailed comparison). In this paper, we choose to base our work on Minimum Condition Maximum Consequent Rules (MCMR). The reason is that the most important criteria in top-k association rule mining is the meaningfulness of rules for users. MCMR meet this goal because it defines non-redundant rules as rules with a minimum antecedent and a maximum consequent [10]. In other words, MCMR are the rules that allow deriving the maximum amount of information based on the minimum amount of information. It is argued that these rules are the most meaningful for several tasks [10]. In the context of top-k association rule mining, other criteria such as the compactness and recoverability of redundant rules are not relevant because the goal of a top-k algorithm is to present a small set of k meaningful rules to the user. MCMR are defined based on the following definition of redundancy [10]. An association rule $r_a : X \rightarrow Y$ is *redundant with respect to another rule* $r_b : X_1 \rightarrow Y_1$ if and only if $conf(r_a) = conf(r_b) \wedge sup(r_a) = sup(r_b) \wedge X_1 \subseteq X \wedge Y \subseteq Y_1$. **Example.** Consider the association rules presented in the right part of Figure 1. The rule $\{a\} \rightarrow \{c, f\}$ is redundant with respect to $\{a\} \rightarrow \{c, e, f\}$. Moreover, the rule $\{a, b\} \rightarrow \{e, f\}$ is redundant with respect to $\{a\} \rightarrow \{e, f\}$.

Problem Definition. Based on the previous definition of redundancy and the definition of top-k association rule mining, we define the problem of *top-k non redundant association mining* as follows. The *problem of mining top-k non-redundant association rules* is to discover a set L containing k association rules in a transaction database. For each rule $r_a \in L \mid conf(r_a) \geq minconf$, there does not exist a rule $r_b \notin L \mid conf(r_b) \geq minconf \wedge sup(r_b) > sup(r_a)$, otherwise r_b is redundant with respect to r_a. Moreover, $\nexists r_c, \mathrm{r}_d \in L$ such that r_c is redundant with respect to $\mathrm{r_d}$.

3. The TNR Algorithm

To address the problem of top-k non-redundant rule mining, we propose an algorithm named TNR. It is based on the same depth-first search procedure as TopKRules. The difference between TNR and TopKRules lies in how to avoid generating redundant rules. The next subsection briefly explains the search procedure of TopKRules.

3.1 The search procedure

To explain the search procedure, we introduce a few definitions**.** A rule $X \rightarrow Y$ is of *size* $p*q$ if $|X| = p$ and $|Y| = q$. For example, the size of $\{a\} \rightarrow \{e, f\}$ is 1*2. Moreover, we say that a rule of size $p*q$ is *larger* than a rule of size $r*s$ if $p > r$ and $q \geq s$, or if $p \geq r$ and $q > s$. An association rule r is *valid* if $sup(r) \geq minsup$ and $conf(r) \geq minconf$.

The search procedure takes as parameter a transaction database, an integer k and the *minconf* threshold. The search procedure first sets an internal *minsup* variable to 0 to ensure that all the top-k rules are found. Then, the procedure starts searching for rules. As soon as a rule is found, it is added to a list of rules L ordered by the support. The list is used to maintain the top-k rules found until now and all the rules that have the same support. Once k valid rules are found in L, the internal *minsup* variable is raised to the support of the rule with the lowest support in L. Raising the *minsup* value is used to prune the search space when searching for more rules. Thereafter, each time that a valid rule is found, the rule is inserted in L, the rules in L not respecting *minsup* anymore are removed from L, and *minsup* is raised to the support of the rule with the lowest support in L. The algorithm continues searching for more rules until no rule are found. This means that it has found the top-k rules in L. The top-k rules are the k rules with the highest support in L.

To search for rules, the search procedure first scans the database to identify single items that appear in at least *minsup* transactions. It uses these items to generate rules of size 1*1 (containing a single item in the antecedent and a single item in the consequent). Then, each rule is recursively grown by adding items to its antecedent or consequent (a depth-first search). To determine the items that should be added to a rule, the search procedure scans the transactions containing the rule to find single items that could expand its antecedent or consequent. The two processes for expanding rules are named *left expansion* and *right expansion*. These processes are applied recursively to explore the search space of association rules. Left and right expansions are formally defined as follows. A *left expansion* is the process of adding an item i to the left side of a rule $X \rightarrow Y$ to obtain a larger rule $X \cup \{i\} \rightarrow Y$. A *right expansion* is the process of adding an item i to the right side of a rule $X \rightarrow Y$ to obtain a larger rule $X \rightarrow Y \cup \{i\}$.

The search procedure described above is correct and complete for mining top-k association rules [5]. It is very efficient because the internal *minsup* variable is raised during the search. This allows pruning large part of the search space instead of generating all association rules. This pruning is possible because the support is monotonic with respect to left and right expansions (see [5] for details).

3.2 Adapting the search procedure to find top-*k* non-redundant rules

We now explain how we have adapted the search procedure to design an efficient top-k non-redundant rule mining algorithm. These modifications are based on the following observation.

Property 3. During the search, if only non-redundant rules are added to L, then L will contain the top-k non-redundant rules when the search procedure terminates. **Rationale**. The search procedure is correct and complete for mining the top-k association rules [5]. If only non-redundant rules are added to L instead of both redundant and non-redundant rules, it follows that the result will be the top-*k* non-redundant rules instead of the top-*k* association rules.

Based on this observation, we have aimed at modifying the search procedure to ensure that only non-redundant rules are added to *L*. This means that we need to make sure that every generated rule r_a is added to *L* only if $sup(r_a) \geq minsup$ and r_a is not redundant with respect to another rule. To determine if r_a is redundant with respect to another rule, there are two cases to consider.

The *first case* is that r_a is redundant with respect to a rule r_b that was generated before r_a. By the definition of redundancy (cf. Definition 2), if r_a is redundant with respect to r_b, then $sup(r_b) = sup(r_a)$. Because $sup(r_a) \geq minsup$, it follows that $sup(r_b) \geq minsup$, and that $r_b \in L$. Therefore, the first case can be detected by implementing the following strategy.

Strategy 1. For each rule r_a that is generated such that $sup(r_a) \geq minsup$, if $\exists\ r_b \in L \mid sup(r_b) = sup(r_a)$ and r_a is redundant with respect to r_b, then r_a is not added to *L*. Otherwise, r_a is added to *L*.

The *second case* is that r_a is redundant with respect to a rule r_b that has not yet been generated. There are two ways that we could try to detect this case. The first way is to postpone the decision of adding r_a to *L* until r_b is generated. However, this would not work because it is not known beforehand if a rule r_b will make r_a redundant and r_b could appear much later after r_a. The second way is to scan transactions to determine if item(s) could be added to r_a to generate a rule r_b such tat r_a redundant is redundant with respect to r_b. However, this approach would also be inefficient because the confidence is non-monotonic with respect to left/right expansions [5]. This mean that it is possible that r_b may contain several more items than r_a.. For this reason, it would be too costly to test all the possibilities of adding items to r_a to detect the second case.

Because the second case cannot be checked efficiently, our solution is to propose an approximate approach that is efficient and to prove that this approach can generate exact results if certain conditions are met. The idea of this approach is the following. Each rule r_a that satisfy the requirements of Strategy 1 is added to *L* without checking the second case. Then, eventually, if a rule r_b is generated such that the rule r_a is still in *L* and that r_a is redundant with respect to r_b, then r_a is removed from *L*. This idea is formalized as the following strategy.

Strategy 2. For each rule r_b that is generated such that $sup(r_b) \geq minsup$, if $\exists\ r_a \in L \mid sup(r_b) = sup(r_a)$ and r_a is redundant with respect to r_b, then r_a is removed from *L*.

By incorporating Strategy 2, the algorithm becomes approximate. The reason is that each rule r_a that is removed by Strategy 2 previously occupied a place in the set *L*. By its presence in *L*, the rule r_a may have forced raising the internal *minsup* variable. If that happened, then the algorithm may have missed some rules that have a support lower than r_a but are non-redundant.

Given that the algorithm is approximate, it would be desirable to modify it to be able to increase the likelihood that the result is exact. To achieve this, we propose to add a parameter that we name Δ that increase by Δ the number of rules *k* that is necessary to raise the internal *minsup* variable. For example, if the user sets k = 1000 and Δ = 100, it will now be required to have $k + \Delta$ = 1100 rules in L to raise the internal *minsup* variable instead of just k=1000. This means that up to 100 redundant rules can be at the same time in L and the result will still be exact. This latter observation is formalized by the following property.

Property 4. If the number of redundant rules in L is never more than Δ rules, then the algorithm result is exact and the k rules in L having the highest support will be the top-k non redundant rules. **Rationale**. As previously explained, the danger is that too many redundant rules are in L such that they would force to raise *minsup* and prune part of the search space containing top-k non-redundant rules. If there is no more than Δ redundant rules at the same time in L and that $k + \Delta$ are needed to raise *minsup*, then redundant rules cannot force to raise *minsup*.

The previous property proposes a condition under which Strategy 2 is guaranteed to generate an exact result. Based on this property, an important question is "Would it be efficient to integrate a check for this condition in the algorithm?". The answer is that it would be too costly to verify that no groups of more than Δ redundant rules are present at the same time in L. The reason is that rules are only known to be redundant when they are removed by Strategy 2. Therefore, checking Property 4 would be very expensive to perform. For this reason, we choose to utilize the following weaker version of Property 4 in our implementation.

Property 5. If the number of redundant rules removed by Strategy 2 during the execution of the algorithm is less or equal to Δ, then the final result is exact and the first k rules of L will be the top-k non redundant rules. **Rationale**. It can be easily seen that if Property 5 is met, Property 4 is also met.

Property 5 can be easily incorporated in the algorithm. To implement this functionality, we have added a counter that is incremented by 1 after every rule removal from L by Strategy 2. Then, when the algorithm terminates, the counter is compared with Δ. If the counter value is lower or equal to Δ, the user is informed that the result is guaranteed to be exact. Otherwise, the user is informed that the result may not be exact. In this case, the user has the option to rerun the algorithm with a higher Δ value. In the experimental study presented in section 4 of this paper, we will address the question of how to select Δ.

The previous paragraphs have presented the main idea of the TNR algorithm. Due to space limitation, we do not provide the pseudo-code of TNR. But the Java source code of our implementation can be downloaded freely from http://www.philippe-fournier-viger.com/spmf/.

Note that in our implementation, we have added a few optimizations that are used in TopKRules [5] and that are compatible with TNR. The first optimization is to try to generate the most promising rules first when exploring the search space of association rules. This is because if rules with high support are found earlier, the algorithm can raise its internal *minsup* variable faster to prune the search space. To perform this, an internal variable R is added to store all the rules that can be expanded to have a chance of finding more valid rules. This set is then used to determine the rules that are the most likely to produce valid rules with a high support to raise *minsup* more quickly and prune a larger part of the search space [5]. The second optimization is to use bit vectors as data structure for representing the set of transactions that contains each rule (*tidsets)* [11]. The third optimization is to implement L and R with data structures supporting efficient insertion, deletion and finding the smallest element and maximum element. In our implementation, we used a Red-black tree for L and R.

4. Experimental Evaluation

We have carried several experiments to assess the performance of TNR and to compare its performance with TopKRules under different scenarios. For these experiments, we have implemented TNR in Java. For TopKrules, we have obtained the Java implementation from their authors. All experiments were performed on a computer with a Core i5 processor running Windows 7 and 2 GB of free RAM. Experiments were carried on four real-life datasets commonly used in the association rule mining literature, namely *Chess*, *Connect*, *Mushrooms* and *Pumsb* (available at: http://fimi.ua.ac.be/data/). The datasets' characteristics are summarized in Table 1.

Experiment 1: What is the percentage of redundant rules? The goal of the first experiment was to assess the percentage of rules returned by TopKRules that are redundant to determine how important the problem of redundancy is. For this experiment, we ran TopKRules on the four datasets with *minconf* = 0.8 and *k* = 2000, and then examined the rules returned by the algorithm. We chose *minconf* = 0.8 and *k* = 2000 because these values are plausible values that a user could choose. The results for *Chess*, *Connect*, *Mushrooms* and *Pumsb* are that respectively, 13.8 %, 25.9 %, 82.6 % and 24.4 % of the rules returned by TopKRules are redundant. These results indicate that eliminating redundancy in top-*k* association rule mining is a major problem.

Table 1. Datasets' Characteristics

Datasets	Number of transactions	Number of distinct items	Average transaction size
Chess	3,196	75	37
Connect	67,557	129	43
Mushrooms	8,416	128	23
Pumsb	49,046	7,116	74

Experiment 2: How many rules are discarded by Strategy 1 and Strategy 2 for Δ = 0**?** The second experiment's goal was to determine how many rules were discarded by Strategy 1 and Strategy 2 for each dataset. To perform this study, we set *minconf* = *0.8, k* = 2000 and Δ = 0. We then recorded the number of rules discarded. Results are shown in Table 2. From this experiment, we can see that the number of discarded rules is high for all datasets and that it is especially high for dense datasets (e.g. *Mushrooms*) because they contains more redundant rules compared to sparse datasets (e.g. *Pumsb*).

Experiment 3: What if we use the Δ parameter? The next experiment consisted of using the $\boldsymbol{\Delta}$ parameter to see if an exact result could be guaranteed by using Property

5. For this experiment, we used *k = 2000* and *minconf* = 0.8. We then set **Δ** to values slightly larger than the number of rules discarded by Strategy 2 in Experiment 2.

For example, for the *Pumsb* dataset, we set **Δ** = 4000. The total runtime was 501 s and the maximum memory usage was around 1.3 GB. The number of rules eliminated by Strategy 1 and Strategy 2 was respectively 3454 and 16,066. Because these values are larger than **Δ**, the result could not be guaranteed to be exact. Moreover, the execution time and memory requirement significantly increased when setting **Δ** = 4000 (with **Δ** = 0, the runtime is 125 s and the maximum memory usage is 576 MB). Furthermore, we tried to continue raising **Δ** and still got similar results.

We did similar experiments with *Chess*, *Connect*, *Mushrooms* and observed the same phenomena. Our conclusion from this experiment is that in practice using the Δ parameter does not help to guarantee an exact result. This means that although the algorithm is guaranteed to find *k* rules that are non-redundant, there rules are not guaranteed to be the *top-k* non-redundant rules.

Table 2. Rules discarded by each strategy for *minconf = 0.8, k=2000* and Δ = 0

Dataset	# rules discarded by Strategy 1	# rules discarded by Strategy 2
Chess	961	10454
Connect	2732	15275
Mushrooms	39848	38627
Pumsb	803	3629

Experiment 4: Performance comparison with TopKRules. The next experiment consisted of comparing the performance of TNR with TopKRules. The parameter *minconf* and *k* were set to 0.8 and 2000 respectively. The execution times and maximum memory usage of both algorithms are shown in Table 3. The results show that there is a significant additional cost for using TNR. The reason is that checking Strategy 1 and Strategy 2 is costly. Moreover, because a large amount of rules are discarded as shown in the second experiment, the algorithm needs to generate much more rules before it can terminate.

Table 3. Performance comparison

Datasets	Runtime (s)		Maximum Memory Usage (MB)	
	TNR	TopKRules	TNR	TopKRules
Chess	8	1.49	269	72.12
Connect	283	25.51	699	403.38
Mushrooms	105	3.46	684	255
Pumsb	125	46.39	576	535

Experiment 5: Influence of the number of transactions. Next, we ran TNR on the datasets while varying the number of transactions in each dataset. We used *k*=500, *minconf*=0.8. We varied the database size by using 70%, 85 % and 100 % of the transactions in each dataset. Results are shown in Figure 2. Globally we found that for all datasets, the execution time and memory usage increased more or less linearly for TNR. This shows that TNR has good scalability.

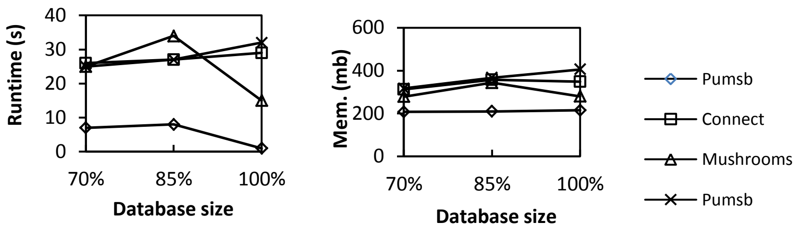

Fig. 2. Influence of the number of transactions

Discussion. Our conclusion from these experiments is that TNR is more costly than TopKRules. But it provides the benefit of eliminating redundancy.

5. Conclusion

Two important problems with classical association rule mining algorithm are that (1) it is usually difficult and time-consuming to select the parameters to generate a desired amount of rules and (2) there can a large amount of redundancy in the results. Previously, these two problems have been addressed separately. In this paper, we have addressed them together by proposing an approximate algorithm named TNR for mining the top-*k* non-redundant association rules. The algorithm is said to be approximate because it is guaranteed to find non-redundant rules. But the rules found may not be the *top-k* non redundant rules. We have compared the performance of TNR with TopKRules and found that TNR is more costly than TopKRules. However, it provides the benefit of eliminating a great deal of redundancy. Source code of TNR and TopKRules can be downloaded at http://www.philippe-fournier-viger.com/spmf/ as part of the SPMF data mining platform.

References

[1] R. Agrawal, T. Imielminski and A. Swami, "Mining Association Rules Between Sets of Items in Large Databases," Proc. ACM Intern. Conf. on Management of Data, ACM Press, June 1993, pp. 207-216.

[2] J. Han, and M. Kamber, Data Mining: Concepts and Techniques, 2nd ed., San Francisco:Morgan Kaufmann Publ., 2006.

[3] Fournier-Viger, P., Wu, C.-W., Tseng, V. S. "Mining Top-K Association Rules," Proc. 25th Canadian Conf. on Artificial Intelligence (AI 2012), Springer, 2012, pp. 61-73.

[4] G. I. Webb and S. Zhang, "k-Optimal-Rule-Discovery," Data Mining and Knowledge Discovery, vol. 10, no. 1, 2005, pp. 39-79.

[5] G. I. Webb, "Filtered top-k association discovery," WIREs Data Mining and Knowledge Discovery, vol.1, 2011, pp. 183-192.

[6] Y. You, J. Zhang, Z. Yang and G. Liu, "Mining Top-k Fault Tolerant Association Rules by Redundant Pattern Disambiguation in Data Streams," Proc. 2010 Intern. Conf. Intelligent Computing and Cognitive Informatics, March 2010, IEEE Press, pp. 470-473.

[7] Bastide, Y., Pasquier, N., Taouil, R., Lakhal, L. and Stumme, G., "Mining minimal non-redundant association rules using frequent closed itemsets," Proc. of the Intern. Conf. DOOD'2000, 2000, Springer, pp. 972-986.

[8] G. Gasmi, S. BenYahia, E. M. Nguifo and Y. Slimani, "A new informative generic base of association rules," Proc. Of PAKDD-05, 2005, Springer, pp. 81-90.

[9] C. L. Cherif, W. Bellegua, S. Ben Yahia and G. Guesmi, "VIE_MGB: A Visual Interactive Exploration of Minimal Generic Basis of Association Rules," Proc. of the Intern. Conf. on Concept Lattices and Application (CLA 2005), 2005, pp. 179-196.

[10] Kryszkiewicz, M., "Representative Association Rules and Minimum Condition Maximum Consequence Association Rules,"

[11] C. Lucchese, S. Orlando and R. Perego, "Fast and Memory Efficient Mining of Frequent Closed Itemsets," IEEE Trans. Knowl. and Data Eng., vol. 18, no. 1, 2006, pp. 21-36.