Mining Sequential Rules Common to Several Sequences with the Window Size Constraint

Philippe Fournier-Viger[1], Cheng-Wei Wu[2], Vincent S. Tseng[2], Roger Nkambou[3]

[1]Dept. of Computer Sciences, University of Moncton. Camada
[2] Dep. of Computer Science and Information Engineering, National Cheng Kung University
[3]Dept. of Computer Sciences, University of Quebec in Montreal. Camada
philippe.fv@gmail.com, silvemoonfox@hotmail.com, tsengsm@mail.ncku.edu.tw, nkambou.roger@uqam.ca

Abstract. We present an algorithm for mining sequential rules common to several sequences, such that rules have to appear within a maximum time span. Experimental results with real-life datasets show that the algorithm can reduce the execution time, memory usage and the number of rules generated by several orders of magnitude compared to previous algorithms.

Keywords: sequential rules, sequence database, sliding-window, time

1 Introduction

In several domains, information is stored in sequence databases (e.g. stock market data, biological data and customer data). Discovering sequential rules [2-8] in databases is important, as it can provide a better understanding of the data. A sequential rule indicates that if some event(s) occur, some other event(s) are likely to occur with a given confidence afterward. Algorithms have been developed for mining sequential rules in a single sequence (e.g. [5, 7]), across sequences (e.g. [4]) or common to sequences (e.g. [2, 3, 6, 8]). In this paper, we are interested by mining rules common to multiple sequences because it has many potential applications (e.g. mining sequential rules common to several customer transactions to make product recommendation). Although several algorithms have been proposed for this task, none considers the duration of sequential rules in terms of time. But for real applications, users often only need to discover patterns occurring within a maximum time span. In this paper, we address this issue by proposing an algorithm named TRuleGrowth. It is an extension of the RuleGrowth algorithm [13]. The rest of this paper is organized as follows. Section 2 presents the problem of mining sequential rules and introduces RuleGrowth. Section 3 presents TRuleGrowth. Section 4 presents the conclusion.

2 Mining Sequential Rules with RuleGrowth

There are two definitions of the problem of mining sequential rules common to multiple sequences [3, 6]. In this paper, we use the one of [3] because it produces rules that are more general. According to this definition, the problem of mining sequential rules is defined as follows. A *sequence database* SD is a set of sequences

$S=\{s_1, s_2 \ldots s_s\}$ and a set of items $I=\{i_1, i_2, \ldots i_t\}$ occurring in these sequences. A *sequence* is an ordered list of *itemsets* (sets of items) $s_x=I_1, I_2, \ldots I_n$ such that $I_1, I_2, \ldots I_n \subseteq I$, and where each sequence is assigned a unique *sid* (sequence id). As an example, figure 1 depicts a sequence database containing four sequences. For instance, sequence *seq1* in figure 1 means that items *a* and *b* occurred at the same time, and were followed successively by *c*, *f*, *g* and *e*. A *sequential rule* X⇒Y is a relationship between two unordered itemsets X, Y ⊆ I such that X∩Y = Ø and X, Y are not empty. The interpretation of a rule X⇒Y is that if the items of X occur in a sequence, the items in Y will occur afterward in the same sequence. Formally, a rule X⇒Y is said to *occur* in a sequence $s_x=I_1, I_2 \ldots I_n$ if there exists an integer *u* such that $1 \leq u < n$, $X \subseteq \bigcup_{i=1}^{u} I_i$ and $Y \subseteq \bigcup_{i=u+1}^{n} I_i$. For example, the rule {*a*, *b*, *c*}⇒{*e*, *f*, *g*} occurs in the sequence {*a*, *b*}, {*c*}, {*f*}, {*g*}, {*e*}. But the rule {*a*, *b*, *f*}⇒{*c*} does not. A rule X⇒Y is said to be of size *v***w* if |X| = *v* and |Y| = *w*. For example, the rules {*a*}⇒{*e*, *f*} is of size 1*2. Furthermore, a rule of size *f***g* is *larger than* another rule of size *h***i* if $f > h$ and $g \geq i$, or if $f \geq h$ and $g > i$.

ID	Sequences
seq1	{*a, b*},{*c*},{*f*},{*g*},{*e*}
seq2	{*a, d*},{*c*},{*b*},{*a, b, e, f*}
seq3	{*a*},{*b*},{*f*},{*e*}
seq4	{*b*},{*f, g*}

→

ID	Rule	Support	Conf.
r1	{*a, b, c*}⇒{*e*}	0.5	1.0
r2	{*a*}⇒{*c, e, f*}	0.5	0.66
r3	{*a, b*}⇒{*e, f*}	0.5	1.0
r4	{*b*}⇒{*e, f*}	0.75	0.75
r5	{*a*}⇒{*e, f*}	0.75	1.0
r6	{*a*}⇒{*b*}	0.5	0.66

Fig 1. A sequence database (left) and some sequential rule found (right)

For a sequence database and a rule X⇒Y, the notation *sids*(X⇒Y) represents the *sids set* (the set of sequence ids) of the sequences where the rule occurs. For an itemset X and a sequence database, the notation *sids*(X) denotes the sids set corresponding to sequences where all the items of X appears. Two interestingness measures are defined for sequential rules. The first one is the *sequential support*. For a rule X⇒Y, it is defined as sup(X⇒Y) = |*sids*(X⇒Y)| / |S|. The second one is the *sequential confidence* and it is defined as conf(X⇒Y) = |*sids*(X⇒Y)| / |*sids*(X)|. The *problem of mining sequential rules common to multiple sequences* is to find all valid rules, i.e. rules such that their support and confidence are respectively no less than user-defined thresholds *minsup* and *minconf*. For instance, figure 1 shows some rules found in a database for *minsup* = 0.5 and *minconf* = 0.5. Several algorithms were proposed for this problem. In this paper, we adapt the current best algorithm named RuleGrowth. To discover rules, RuleGrowth proceeds as follows. It first finds rules of size 1*1 and then recursively grows them by scanning the sequences containing them to find single items that can expand their left or right parts. This strategy ensures that only rules occurring in the database are considered as potential valid rules by the algorithm. The two processes for expanding rules in RuleGrowth are named left expansion and right expansion. Formally, a *left expansion* is the process of adding an item *i* to the left side of a rule X⇒Y to obtain a larger rule X∪{*i*}⇒Y. Similarly, a *right expansion* is defined as the process of adding an item *i* to the right side of a rule X⇒Y to obtain a larger rule X⇒Y∪{*i*}. An important property of expansions is that any rule obtained by an expansion has a support that is lower or equal to that of the original rule [8] (the support is anti-monotonic with respect to left/right expansions). Therefore, all the rules having a support of at least *minsup* can be found by recursively performing expansions on frequent rules of size 1*1 (rules with a support higher or equal to *minsup*). Moreover, this property guarantees that expanding a rule having a support less than *minsup* will not result in a rule having a support no less than *minsup*.

RuleGrowth takes as input a sequence database S, minsup and *minconf*. Its main procedure is shown in Figure 2. It first scans the database once to calculate *sids*(*c*) for each item *c*. Then, the algorithm scans the database a second time and removes each item *c* such that $|sids(c)| / |S| \leq minsup$, because all such item cannot be part of a valid rule. After that, the algorithm generates all valid rules of size 1*1 with the remaining items. This is done by considering each pair of items *i, j* one by one. The algorithm scans sequences in $sids(i) \cap sids(j)$ to calculate $sids(i{\Rightarrow}j)$ and $sids(j{\Rightarrow}i)$, the sids of sequences where the rule $\{i\}\Rightarrow\{j\}$ and $\{j\}\Rightarrow\{i\}$ occur, respectively. After this, the support of the rule $\{i\}\Rightarrow\{j\}$ is obtained by dividing $|sids(i{\Rightarrow}j)|$ by $|S|$. If the support is no less than *minsup*, the procedure EXPANDLEFT and EXPANDRIGHT are called to try to expand the rule's left and right parts recursively, and the confidence of the rule is calculated by dividing $|sids(i{\Rightarrow}j)|$ by $|sids(i)|$. If the confidence is higher than or equal to *minconf*, the rule is valid and the algorithm outputs the rule. After this, the same process is repeated for the rule $\{j\}\Rightarrow\{i\}$. Then, the algorithm considers all other pairs of items in the same way. It can be easily seen that the main procedure of RuleGrowth outputs all and only the valid rules of size 1*1. The next paragraphs explain how it can also find rules of larger size by recursively expanding rules.

```
RuleGrowth(S, minsup, minconf)
1.  Scan the database one time. For each item $c$ found, record the sids of the sequences that contains $c$ in a
    variable $sids(c)$.
2.  Scan the database S a second time and remove each item $c$ such that $|sids(c)| / |S| \leq minsup$.
3.  FOR each pair of items $i, j$ :
4.      $sids(i{\Rightarrow}j) := \emptyset$.   $sids(j{\Rightarrow}i) := \emptyset$.
5.      FOR each sid $s \in (sids(i) \cap sids(j))$
6.          IF $i$ occurs before $j$ in $s$, $sids(i{\Rightarrow}j) := sids(i{\Rightarrow}j) \cup \{s\}$.
7.          IF $j$ occurs before $i$ in $s$, $sids(j{\Rightarrow}i) := sids(j{\Rightarrow}i) \cup \{s\}$.
8.      IF $(|sids(i{\Rightarrow}j)| / |S|) \geq minsup$ THEN
9.          EXPANDLEFT($\{i\}\Rightarrow\{j\}$, $sids(i)$, $sids(i{\Rightarrow}j)$).
10.         EXPANDRIGHT($\{i\}\Rightarrow\{j\}$,$sids(i)$, $sids(j)$, $sids(i{\Rightarrow}j)$).
11.         IF $(|sids(i{\Rightarrow}j)| / |sids(i)|) \geq minconf$ THEN OUTPUT rule $\{i\}\Rightarrow\{j\}$ with its conf. and support.
12.     IF $(|sids(j{\Rightarrow}i)| / |S|) \geq minsup$ THEN
13.         EXPANDLEFT($\{j\}\Rightarrow\{i\}$, $sids(j)$, $sids(j{\Rightarrow}i)$).
14.         EXPANDRIGHT($\{j\}\Rightarrow\{i\}$,$sids(j)$, $sids(i)$, $sids(j{\Rightarrow}i)$).
15.         IF $(|sids(j{\Rightarrow}i)| / |sids(j)|) \geq minconf$ THEN OUTPUT rule $\{j\}\Rightarrow\{i\}$ with its conf. and support.
```

Fig. 2. The RuleGrowth algorithm

The main problem that had to be solved is how to identify items that can expand a rule left part or right part to produce a valid rule. By exploiting the fact that any valid rule is a rule with a support higher or equal to *minsup*, this problem is decomposed into two sub-problems, which are (1) determining items that can expand a rule I⇒J to produce a frequent rule and (2) assessing if a frequent rule obtained by an expansion is valid. The first sub-problem is solved as follows. To identify items that can expand a rule *r*:I⇒J and produce a frequent rule, RuleGrowth scan the sequences from *sids*(I⇒J). During this scan, each item *c* such that $c \notin$ I, $c \notin$ J and *c* occurs before the last occurrence of J in at least $minsup \times |S|$ sequences from *sids*(I⇒J) is noted. Those items are the one that will produce a frequent rule by a left expansion of *r*. For right expansions, we note each item *c* such that $c \notin$ I and $c \notin$ J and *c* occurs after the first occurrence of I in at least *minsup*×|S| sequences from *sids*(I⇒J). The second sub-problem is to determine if a rule obtained by an expansion of a frequent rule I⇒J with an item *c* is a valid rule. To do this, the confidence of the rule has to be calculated. For a left expansion, the confidence is obtained by dividing |*sids*(I∪{*c*}⇒J)| by |*sids*(I∪{*c*})|. The set *sids*(I∪{*c*}⇒J) is determined by noting each sequence where *c* expand the rule I⇒J when searching items for the left expansion of I⇒J, as explained in the previous paragraph. The set *sids*(I∪{*c*}) is calculated by scanning each sequences from *sids*(I) to see if *c* appears in it. For a rule of size 1*1, *sids*(I) is determined during the initial database scan of the algorithm, and for larger rules, it can be updated after each left expansion. For a right expansion, the confidence is calculated by dividing |*sids*(I⇒J∪{*c*})| by |*sids*(I)|. The set *sid*(I⇒J∪{*c*}) is determined by noting each sequence where *c* expand the rule I⇒J when searching items for the right expansion of I⇒J as explained in the previous paragraph. Pseudo-code of the EXPANDLEFT and EXPANDRIGHT procedures, which incorporate the ideas discussed above, can be found in the RuleGrowth paper [8]. RuleGrowth is a correct and complete algorithm (it generates all and only valid rules) (see [8] for details). RuleGrowth is also guaranteed to not find the same rule twice, thanks to two strategies (see [8] for justifications). The first one is to not allow performing a right expansion after a left expansion. The second one is to only add an item to rule itemset if the item is greater than each item in the itemset according to the lexicographic ordering [8].

3 The TRuleGrowth Algorithm

We now present TRuleGrowth. We define the problem of mining sequential rules with a sliding-window as being the same as the problem of mining sequential rules except that a rule X⇒Y to **occur** in a sequence s=I_1, I_2 … I_n if there exist integers *j, k, m* such that $1 \leq j \leq k < m \leq n$, $X \subseteq \bigcup_{i=j}^{k} I_i$ and $Y \subseteq \bigcup_{i=k+1}^{m} I_i$ and that $m - j + 1 \leq$ *window_size*, where *window_size* is defined by the user. We have made the following modifications to RuleGrowth. First, in the main procedure, all occurrences of each item are kept for each sequence. Then, when considering each pair of items *i* and *j* to generate rules of size 1*1, this information is used to quickly check if *i* and *j* appear within the time window for each sequence from *sids(i)* ∩ *sids(j).* This enforces the window size constraint for rules of size 1*1. For larger rules, we have modified EXPANDLEFT and EXPANDRIGHT. Due to space limitation, we only explain here the modifications to EXPANDLEFT. First, the check for identifying each potential item *c* that can expand a rule I⇒J is modified to only consider items *c* such that the resulting rule would respect the time window. This is done efficiently as follows. To identify potential items for a left expansion, each sequence containing the rule is scanned one time. Each time that an itemset is read from a sequence, each item of that itemset that is included in I is added to an HashMap *hashI* with the position of the itemset in the sequence and each item that is included in J is added to an HashMap *hashJ* with the position of the itemset in the sequence. When considering the next itemset, all items that were found more than *window_size* itemsets before are removed from *hashI* and *hashJ* (because we consider them to be falling outside the window defined by the current itemset and the last *window_size* -1 itemsets read). When the sum of |*hashI*| and |*hashJ*| is the same as |I∪J|, it means that all items are in the current window. However, to ensure that I occurs before J in the window, items should be only added to *hashI* if |*hashJ*| = |J| and *hashI* should be emptied if |*hashJ*| becomes smaller than |J|. When | *hashI*| + |*hashJ*| becomes equal to |I∪J|, each item *c* occurring after the first item of J such that the window size is respected can be added to the set of potential items for this sequence, given that *c* ∉ I and *c* ∉ J. An important note is that for the implementation, the HashMaps should only keep the most recent position for each item. In Java, this behavior is the default behavior if each sequence is scanned from right to left. The second modification is that when *sids(*I∪{*c*}⇒J) is calculated, the calculation has to take into account the window constraint. This can be achieved in a similar way as what is described in the previous paragraph. The full pseudocode of EXPAND-LEFT is given in figures 2. The proof of corectness and completedness is not shown due to space limitation. But it can be easily proved given that RuleGrowth is correct and complete [8].

```
EXPANDLEFT($I \Rightarrow J$, sids(I), sids(I⇒J))
1.  FOR each sid ∈ sids(I⇒J)
2.      hashI := Ø.  hashJ := Ø.
3.      FOR each itemset X in sequence sid, from the last one to the first one.
4.          REMOVE all items from hashI and hashJ seen more than window_size – 1 itemsets
               before.
5.          IF |hashJ| was equal to |J| and became smaller after removing items THEN |hashI| := Ø.
6.          IF |hashJ| = |J| THEN add each item c ∈ I∩X to hashI with the position of X in
               sequence sid.
7.          IF |hashJ| < |J| THEN add each item d ∈ J∩X to hashJ with the position of X in
               sequence sid.
8.          IF |hashI| = |I| and |hashJ| = |J| THEN add sid to a variable sids(I∪{c}⇒J) for each item
               c ∉ I∪J  occurring before the first item of J in the window.
9.  FOR each item c where |sids(I∪{c}⇒J)| / |S| ≥ minsup :
10.     sids(I∪{c}) := Ø.
11.     FOR each sid ∈ sids(I) such that sid ∈ sids(c):
12.         IF c and I occur within the maximum window THEN sids(I∪{c}):= sids(I∪{c}) ∪{sid}.
13.     EXPANDLEFT(I∪{c}⇒J, sids(I∪{c}), sids(I∪{c}⇒J))
14.     IF |sids(I∪{c}⇒J)| / | sids(I∪{c})| ≥ minconf THEN OUTPUT rule I∪{c}⇒J.
```

Fig. 2. The EXPANDLEFT procedure of TRuleGrowth

Performance evaluation. To evaluate TRuleGrowth, we compared its performance with RuleGrowth because it is the current best algorithm. RuleGrowth and TRuleGrowth were implemented in Java. We used several datasets. Due to space limitation, we only show results for **Kosarak** (http://fimi.cs.helsinki.fi/data/), a dataset of 990,000 sequences of click-stream data. Algorithms were run with *minconf*=0.2, while varying *minsup* from 0.004 to 0.001. TRuleGrowth was run with *window_size* = 10, 12 and 14. Results are shown in figure 3. In this figure, the notation W*x* represents TRuleGrowth with *window_size* = *x*. We observed that TRuleGrowth can be several orders of magnitudes faster than RuleGrowth and generate several orders of magnitudes less rules. We also found that when *window_size* is increased, there is a point where TRuleGrowth becomes slower than

RuleGrowth. This is because TRuleGrowth has to perform extra calculations for verifying the window size constraint, and when *window_size* is set above a certain value, this calculation takes more time than what is saved by pruning the search space. Lastly, we found that TRuleGrowth has better memory scalability than RuleGrowth.

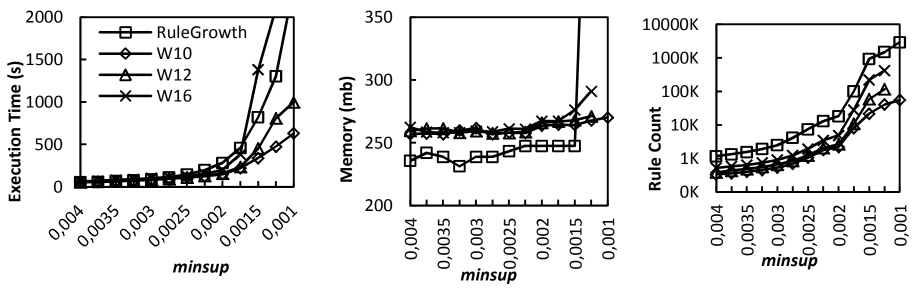

Figure 3. Influence of *window_size* for Kosarak

4 Conclusion

This paper presented TRuleGrowth, an algorithm for mining sequential rules common to several sequences with a sliding-window. Experiments have shown that TRuleGrowth can be up to several of magnitudes faster and uses up to an order of magnitude less memory than RuleGrowth. Source code of RuleGrowth and TRuleGrowth can be downloaded at http://www.philippe-fournier-viger.com/spmf/ .

References

1. Laxman, S. & Sastry, P.: A survey of temporal data mining. Sadhana 3, 173-198 (2006)
2. Zaki, M. J.: SPADE: An Efficient Algorithm for Mining Frequent Sequences. Machine Learning, 42(1/2), 31-60 (2001)
3. Fournier-Viger, P., Faghihi, U., Nkambou, R. & Mephu Nguifo, E.: CMRules: An Efficient Algorithm for Mining Sequential Rules Common to Several Sequences. Knowledge Based Systems, 25 (1), 63-76 (2012)
4. Das., G., Lin, K.-I., Mannila, H., Renganathan, G., & Smyth, P.: Rule Discovery from Time Series. In: Proc. ACM SIGKDD'98, pp.16-22. ACM Press (1998)
5. Deogun, J.S. & Jiang, L.: Prediction Mining – An Approach to Mining Association Rules for Prediction. In: Proc. RSFDGrC 2005, LNCS, vol. 3641, pp. 98-108. Springer (2005)
6. Lo, D., Khoo, S.-C. & Wong, L.: Non-redundant sequential rules – Theory and algorithm. Inform. Systems, 34 (4-5), 438-453 (2009)
7. Mannila, H., Toivonen & H., Verkano, A.I.: Discovery of frequent episodes in event sequences. Data Mining and Knowledge Discovery, 1(1): 259-289 (1997)
8. Fournier-Viger, P., Nkambou, R. & Tseng, V. S.: RuleGrowth: Mining Sequential Rules Common to Several Sequences by Pattern-Growth. In: Proc. SAC 2011, pp. 954-959 (2011)

Acknowledgments. Our thanks go to the FQRNT and NSERC for their financial support.