Applied Mathematics and Nonlinear Sciences

<https://www.sciendo.com>

Temporal association rules discovery algorithm based on improved index tree

Chen Yuanyuan¹, Wang Rui^{2†}, Zeng Bin¹, Griffith W. S.³

¹Department of Management and Economics, Naval University of Engineering, Wuhan, Hubei, China

²Teaching and Research Support Center, Naval University of Engineering, Wuhan, Hubei, China

Submission Info

Communicated by Juan Luis García Guirao

Received December 1st 2020

Accepted January 31st 2021

Available online March 19th 2021

Abstract

With the rapid increase of information generated from all kinds of sources, temporal big data mining in business area has been paid more and more attention recently. A novel data mining algorithm for mining temporal association is proposed. Mining temporal association can not only provide better predictability for customer behaviour but also help organisations with better strategies and marketing decisions. To compare the proposed algorithm, two methods to mine temporal association are presented. One is improved based on a traditional mining algorithm, Apriori. The other is based on an Index-Tree. Moreover, the proposed method is extended to mine temporal association in multi-dimensional space. The experimental results show that the Index-Tree method outperforms the Apriori-modified method in all cases.

Keywords: data mining, temporal data mining, association rule, apriori algorithm

1 Introduction

Data mining is motivated by the decision support problem faced by many organisations. Among all techniques, mining association rules are able to find an interesting association or correlation relationships from a large collection of data items. It is useful to discover all such relationships for many organisations. Most of the proposed methods developed in mining frequent patterns focus on the problems with discrete data items such as customers' buying data in a supermarket. However, there exist some problems where the data items are associated with intervals (continuous or discrete) in database transactions [1, 2]. All the previously proposed methods are not completely suitable for these problems.

First, let us take a look at some applications in the real world where the data items are associated with intervals. For example, there is a large database recording the time and number of people making phone calls with cellular phones for a telephone company. The phone usage data are usually associated with an interval with

[†]Corresponding author.

Email address: 18438080@qq.com

the starting and ending time. Hence, if we can mine the frequent calling intervals and thus figure out the ‘hotter’ time intervals in which customers are more likely to use their cellular phones, the cellular phone companies can be more comprehensive of the usage patterns of customers and can therefore make better marketing decisions or provide better services.

Therefore, in this article, we propose two methods to mine frequent intervals. The first method is revised from the Apriori algorithm [3]. However, because of the inefficiency with the Apriori-modified algorithm [4], we proposed the second algorithm. The algorithm uses an index tree structure, called Index-Tree, which is used to efficiently store all the intervals in a database. The new mining frequent intervals can not only provide better predictability for customer behaviour but also help organisations with better strategies and marketing decisions.

2 Problem description

If we want to mine large itemsets in a database, we just have to generate all possible candidate itemsets and scan the whole database to verify which ones are frequent for the exact solution. But things will be different if the database is composed of many data intervals. If the intervals are within the discrete domain as in the railroad example, the situation is simpler and we can view each railroad station as an individual item. A transaction will become a set of several successive stations, then, and we may be able to use traditional methods to find frequent intervals. The major difference is that the stations in the frequent interval must be in sequential order.

However, when we are mining frequent intervals within a continuous domain such as the temporal data in the telephone company example, the main difficulty is that the number of data points in a continuous interval is infinite [5, 6]. In a huge database containing plenty of data intervals, it is impossible and unrealistic to find all possible frequent data point, just like what we do for those discrete transaction data. Therefore, what we do here is just to discretise the whole continuous domain into several equal-sized unit intervals, and then transform the original data according to these unit intervals, as shown in Figure 1. After transforming the data, we can then treat them the same way as data intervals with discrete domain as in the railroad company example.

Fig. 1 Discretising the continuous domain.

After surveying the related data mining techniques, we can see that all the previous methods focus on the problems of finding the large itemsets or finding the frequent path/segments [7, 8]. These methods are not suitable for the problems of finding the temporal association. When the data are associated with intervals, some special properties are anticipated to arise, which may facilitate us to mine frequent intervals. Therefore, we shall propose methods to mine frequent intervals in the following section.

Here an example is given as below that there is a database D has 10 transactions as illustrated in Figure 2.

Although we know that the performance of the FP growth algorithm is better than the Apriori algorithm [9], our proposed methods still adopt the Apriori-modified approaches. This is because the FP growth algorithm has some problem in dealing with multi-dimensional intervals. Since the FP growth algorithm cannot intuitively and efficiently handle the multi-dimensional problems [10], whereas the concept of the Apriori algorithm does very well, we choose the Apriori algorithm for the basic approach in this article.

Fig. 2 Graph representation of transactions in example 1.

3 Apriori modified method

The first algorithm we propose to solve the problem of mining frequent intervals is revised from the Apriori algorithm. We can then directly apply the Apriori algorithm to find any frequent k intervals. The first step is to find all frequent 1 intervals and their corresponding supports by scanning the database once. Assume that F_k means the set of all frequent k intervals. Then we can generate all candidate $(k + 1)$ intervals, C_{k+1} , by joining F_k with itself, denoted by $F_k * F_k$. However, due to the characteristic of our problem, we modify the joining approach as follows. For any two distinct frequent k intervals in F_k , say (U_s, U_e) and $(U_{s'}, U_{e'})$, we join them into a candidate $(k + 1)$ interval only if $s - s' = 1$. The resulting candidate interval is $(U_{s'}, U_e)$. With the set of all candidate $(k + 1)$ intervals, C_{k+1} , we can scan the database and get the set of all frequent $(k + 1)$ intervals, F_{k+1} , and their supports. The process is repeated until no new frequent interval generated. Then we can get all frequent intervals and their corresponding supports. This Apriori-modified algorithm is illustrated in Figures 3 and 4.

```

Algorithm Apriori-modified
Input : Database,  $D$ ; minimal support,  $minsup$ .
Output : A set of frequent intervals.
begin
     $k \leftarrow 1$ ;
    Scan  $D$  to get  $F_1$ ;
    while  $F_k \neq \emptyset$  do
        begin
             $C_{k+1} \leftarrow \text{Join}(F_k)$ ;
            Scan  $D$  to get  $F_{k+1}$ ;
             $k \leftarrow k+1$ ;
        end
    return  $F_k$ ;
end

```

Fig. 3 The Apriori-modified algorithm.

The main advantage of the Apriori algorithm is simple. However, there are two drawbacks for the Apriori algorithm in finding large itemsets for association rules [11]. One is that a large number of candidate itemsets may be generated (at most 2^n , n is the number of items). The other is that the times of scanning database may be the number of items in the longest large itemset.

However, when we are mining frequent intervals, the maximum number of possible candidate intervals is bound by $O(m^2)$, where m is the number of unit intervals in the domain, because of the characteristic of such a

```

Function Join
Input : Frequent interval set F.
Output : Candidate interval set C.
begin
  C ← ∅;
  for each pair of distinct intervals  $I_1=(U_{s1}, U_{e1})$   $I_2=(U_{s2}, U_{e2})$  in F
    if ( $s1 - s2 = 1$ ) then
       $I_3=(U_{s2}, U_{e1})$ ;
       $C \leftarrow C \cup I_3$ ;
    end if
  end for
  return C;
end

```

Fig. 4 Function join.

Table 1 All frequent intervals and their supports

ID	Starting	Ending	Support	ID	Starting	Ending	Support
1	2	2	2/5	12	5	6	3/5
2	3	3	1/2	13	6	7	7/10
3	4	4	1/2	14	7	8	3/5
4	5	5	3/5	15	8	9	2/5
5	6	6	4/5	16	4	6	2/5
6	7	7	7/10	17	5	7	1/2
7	8	8	3/5	18	6	8	3/5
8	9	9	2/5	19	7	9	2/5
9	2	3	2/5	20	5	8	2/5
10	3	4	2/5	21	6	9	2/5
11	4	5	2/5				

kind of mining problem [12]. This number is much smaller than $O(2^m)$. The reason is that the total number of intervals in a domain with m unit intervals is:

$$m + (m - 1) + \dots + 2 + 1 = \sum_{k=1}^m k = \frac{m(m+1)}{2} = O(m^2).$$

Therefore, the problem of generating tremendous candidates in finding frequent intervals is not as serious as in the association rules mining.

But the Apriori-modified method is still not efficient enough since it may scan the database as many times as the length of the longest frequent interval. We know that in the real-world application, database D is usually very large. It is unsatisfactory to scan the whole database multiple times. Therefore, we propose the second method to improve efficiency in the next section.

Example: Consider the data shown below to ascertain the steps with which to find all the frequent intervals using the Apriori-modified algorithm, as shown in Figure 5.

4 Index-tree algorithm

4.1 Index-tree definition

An Index-Tree is used to keep all the information of intervals in the database. Since a database contains a large number of transactions and each transaction contains an interval, an Index-Tree must preserve all these

Fig. 5 Steps to find frequent intervals by using the Apriori-modified algorithm.

elements of information for mining frequent intervals later.

In an Index-Tree, there may be several nodes with count zero. Such nodes not only waste disk space but also degrade the performance of the algorithm of mining frequent intervals [13, 14]. Therefore we do not create such nodes in an Index-Tree. In other words, we have to do tree node compression. There are two possible situations: (1) the empty node is a rightmost node in a certain level and (2) the empty node is not a rightmost node. In the latter situation, we can just ignore such empty nodes. However, in the former situation, we cannot eliminate such node unless all the counters in the left subtree of this node are equal to zero. After the compression, the number of the empty nodes in an Index-Tree will be less than the maximum level of the tree. Figure 6 illustrates the example of a compressed Index-Tree where the nodes with dotted rectangles are eliminated, the interval in the root node '1~4 (20)' means that the number of occurrences of interval (1, 4) is 20.

Fig. 6 An example of an index-tree.

To construct an Index-Tree, we first create a root node. The root node represents the maximum interval for the domain of the database. The counter of the root node is initially set to zero. Then we start scanning the database. For each interval in the database, we insert it into the Index-Tree. The insert algorithm is performed

Algorithm Insert**Input** : An Index-Tree I , and an interval i .**Output** : An updated Index-Tree I .**begin** find a node of i 's lowest "rightmost" ancestor in I ; **if** (lowest "rightmost" ancestor does not exist) **then** create the lowest "rightmost" ancestor of i with counter = 0; **end if** find a node of i in I such that the interval of the node is equal to i ; **if** (such a node exists) **then**

increment the counter by 1;

else create a node with interval i and set its counter to 1; **end if** **return** I ;**end****Fig. 7** Algorithm insert.

as follows. First, we traverse the Index-Tree from the root node to find the lowest 'rightmost' ancestor which is the rightmost node in the lowest level such that its interval contains the inserting interval. If the lowest rightmost ancestor does not exist, create it and set its counter to zero. Second, find the target node from the rightmost ancestor through its left child link such that the interval of the target node is equal to the inserting interval. If the target node does not exist, create it and set its counter to 1; otherwise, increment the counter of the node by one. The insert algorithm is performed repeatedly until all transactions in the database are inserted. Finally, we shall get the Index-Tree for further mining frequent intervals. The insert algorithm is shown in Figure 7.

If the Index-Tree is too large, we do not construct the entire Index-Tree. Instead, we directly construct the conditional Index-Tree from the database and mine frequent intervals from the conditional Index-Tree. If the conditional Index-Tree is still too large, we can recursively construct a smaller conditional Index-Tree from the database. After mining the frequent intervals for all conditional Index-Tree, the union of all frequent intervals will be the complete set of the frequent intervals.

Theorem 1: *The time complexity of constructing an Index-Tree with respect to a database is bound by $O(\text{number of transactions} \times \text{number of unit intervals})$.*

Proof: The time complexity of inserting one interval into an Index-Tree will be proportional to the number of nodes in the traversal path starting from the root node to the target node. Therefore, in the worst case, the time complexity of inserting one interval is bound by $O(\text{maximum number of levels in the Index-Tree})$, and the maximum number of levels in the Index-Tree is bound by $O(\text{number of unit intervals})$. However, since we have performed the scheme of nodes compression in our inserting algorithm, the average cost will be much less than that of the worst case. The time complexity of constructing an Index-Tree with respect to a database is bound by the product of the number of transactions and the insertion cost of each transaction. That is, the time complexity of constructing an Index-Tree is bound by $O(\text{number of transactions} \times \text{number of unit intervals})$.

4.2 Intervals mining with index-tree

Once the Index-Tree is constructed from the database, we can use the Index-Tree to mine frequent intervals. The main idea of our algorithm is illustrated as follows. First, we scan the database once to get frequent 1-intervals and construct the Index-Tree accordingly. Second, we perform the same iterative process to join the frequent intervals to generate candidate intervals as in the Apriori-modified algorithm. The only difference is that we scan the Index-Tree to get the supports of the intervals instead of scanning the whole database.

To calculate the support of an interval, we start from the root of the Index-Tree. If a node contains the interval, the counter of the node is added to its support. Then we recursively traverse the left and right (if exist)

```

Algorithm Index-Tree-Method
Input : Database,  $D$ ; minimal support,  $minsup$ ; number of transactions  $n$ .
Output : A set of frequent intervals.
begin
   $k \leftarrow 1$ ;
  Scan  $D$  to get  $F_1$  and build an Index-Tree  $I$ ;
  while  $F_k \neq \emptyset$  do
     $C_{k+1} \leftarrow \text{Join}(F_k)$ ;
    //the join operation is the same as in the Apriori-modified
    algorithm
     $F_{k+1} = \text{Count}(I, C_{k+1}, n \times minsup)$ ;
     $k \leftarrow k+1$ ;
  end
  return  $\bigcup_k F_k$ ;

```

Fig. 8 Algorithm index-tree method.

```

Function Count
Input : Index-Tree  $I$ , candidate interval set  $C$ , and minimal frequency
 $minfre$ .
Output : Frequent interval set  $F$ .
begin
   $F \leftarrow \emptyset$ ;
  foreach candidate  $c_i$  in  $C$ , set  $c_i$ 's counter to zero;
  foreach node  $n$  in  $I$ 
     $C_n = \{\text{intervals in } C \text{ which are contained in } n\}$ 
    foreach candidate  $c$  in  $C_n$  do
       $c.\text{counter} \leftarrow c.\text{counter} + n.\text{counter}$ ;
  foreach candidate  $c_i$  in  $C$  do
    if  $\{c_i.\text{counter} \geq minfre\}$  then
       $F \leftarrow F \cup \{c_i\}$ ;
  return  $F$ ;

```

Fig. 9 Function count.

child nodes of the node. If the interval of any child node contains the interval, the counter of this node is added to its support. If a node in the Index-Tree does not contain the interval, all the sub-nodes of the node will not contain the interval, either. This property means that when we calculate the support of an interval, we do not have to perform a complete breadth-first search of the Index-Tree. By using this property, we can speed up the process of counting supports. The algorithm is shown in Figures 8 and 9.

Lemma 1: The number of nodes in the Index-Tree is bound by $O((\text{number of unit intervals})^2)$.

Proof: In the worst case, the number of nodes in an Index-Tree is $\frac{n(n+1)}{2}$, where n is the number of unit intervals. Therefore, the number of nodes in the Index-Tree is bound by $O((\text{number of unit intervals})^2)$. However, since the Index-Tree is constructed with the node compression scheme, the average cost will be much less than that of the worst case.

Lemma 2: The total time complexity of all join operation in the Index-Tree-Method algorithm is bound by $O((\text{number of unit intervals})^2)$.

Proof: The total time complexity of all join operation in the Index-Tree-Method algorithm is in proportion to the number of the generated candidate intervals. In the worst case, the number of the generated candidate intervals is bound by $O((\text{number of unit intervals})^2)$. Therefore, the total time complexity of all join operation in the Index-Tree-Method algorithm is bound by $O((\text{number of unit intervals})^2)$.

Lemma 3: The total time complexity of all count operation in the Index-Tree-Method algorithm is bound by $O((\text{number of unit intervals})^4)$.

Proof: The total time complexity of all count operation in the Index-Tree-Method algorithm is bound by the production of the number of scanning the Index-Tree and the average cost of scanning the Index-Tree once. The number of scanning the Index-Tree is bound by $O(\text{length of the longest frequent interval})$. The worst case of scanning an Index-Tree once is when we have to perform a complete breadth-first search against the Index-Tree. Hence, the time complexity of scanning an Index-Tree once is bound by $O(\text{number of candidate intervals in an iteration} \times (\text{number of unit intervals})^2)$. Therefore, the total time complexity of all count operation in the Index-Tree-Method algorithm is bound by $O(\text{length of the longest frequent interval} \times \text{number of candidate intervals in an iteration} \times (\text{number of unit intervals})^2) = O(\text{total number of candidate intervals} \times (\text{number of unit intervals})^2) = O((\text{number of unit intervals})^4)$.

Theorem 3: The time complexity of the Index-Tree-Method algorithm is bound by $O(\max\{(\text{number of transactions} \times \text{number of unit intervals}), (\text{number of unit intervals})^4\})$.

Proof: The time complexity of the Index-Tree-Method algorithm is the sum of the cost of constructing the Index-Tree, that of the join operation, and that of count operation. The first one is presented in Theorem 2 and is bound by $O(\text{number of transactions} \times \text{number of unit intervals})$. As shown in Lemma 2 and Lemma 3, the second one is bound by $O((\text{number of unit intervals})^2)$ and the third one is bound by $O((\text{number of unit intervals})^4)$. Therefore, the time complexity of the Index-Tree-Method algorithm is bound by $O(\max\{(\text{number of transactions} \times \text{number of unit intervals}), (\text{number of unit intervals})^4\})$.

The performance of the Index-Tree-Method algorithm is better than that of the Apriori-modified algorithm. This is because an Index-Tree is much more condensed than the database. The Index-Tree-Method algorithm is required to scan the database only once. However, the Apriori-modified algorithm needs to scan the database several times.

4.3 Multi-dimensional data mining

In the previous sections, we have discussed the problem of mining frequent intervals in the one-dimensional (1-d) space. The idea can be extended to multi-dimensional space [15]. For a multi-dimensional interval, it has one range in each dimension. In the two-dimensional (2-d) space, an interval represents a rectangle, as shown in Figure 10. In the three-dimensional (3-d) space, an interval represents a cube. For the dimensions higher than 3, an interval represents a hyper-cube.

Fig. 10 An example of 2-d intervals.

The corresponding interval of a direct child node of a certain dimension has the following property that the range of that dimension is one unit less than its parent node's but with the same ranges in all other dimensions. The corresponding interval of a direct child node is a subinterval of the parent node's interval and the sub-interval is one unit less in size in only one dimension. This notion (in 2-d) is shown in Figure 11, where R11 and R12 are the x-dimensional sub-intervals of two direct child nodes of their parent node whose interval is R1, and R21

Fig. 11 An example of sub-intervals.

Fig. 12 Runtime vs. number of transactions ($n = 100$, $m = 5\%$, $s = 5\%$).

and R22 are the y-dimensional sub-intervals of two direct child nodes of their parent node whose interval is R1.

According to the previously described insert algorithm, the node compression scheme is enhanced automatically. We do not create a node if the counter is zero except that node is an 'anchor' ancestor of a non-zero node. By doing node compression, the number of nodes in a k -d Index-Tree is smaller. It is worthy of mention that, when constructing an Index-Tree in the 1-d case, we first find the 'rightmost' ancestor and then find the target node. In fact, the 'rightmost' ancestor is an 'anchor' ancestor we have defined in constructing a k -d Index-Tree.

5 Experiment results

In this section, we compare the performance of our proposed methods for mining frequent intervals. We first synthesise 1-d interval data to evaluate these two methods. The synthetic data are generated as follows. The lengths of intervals are in Poisson distribution with mean m . Then the starting point of an interval is distributed uniformly. Four factors are dominating the performance of both methods in the 1-d case: the number of transactions (D), mean length of an interval (m), minimum support threshold (s) and the number of unit intervals (n). We freely set the values of the first three factors in the synthetic data.

In Figure 12, we compare the running time against the number of transactions for both methods. The number of transactions ranges from 10K to 100K, the number of unit intervals is 100, the mean length of each interval is 5%, and the minimum support threshold is 5%. As shown in Figure 12, the performance of the Index-Tree-Method algorithm outperforms that of the Apriori-modified algorithm. The running time of both algorithms increases as the number of transactions increases since the cost of scanning the database and that of the Index-

Fig. 13 Runtime vs. mean interval length ($d = 10k$, $n = 100$, $s = 5\%$).

Fig. 14 Runtime vs. support threshold ($d = 100k$, $n = 100$, $m = 5\%$).

Tree also increases.

In Figure 13, we compare the running time against the mean length of each interval for both methods. The mean length of each interval ranges from 5% to 50%, the number of transactions is 10K, the number of unit intervals is 100 and the minimum support threshold is 5%. As shown in Figure 13, the running time of both algorithms increases linearly as the mean length of each interval increases. When the mean length of intervals increases, so does the number of scanning the database in the Apriori-modified algorithm and that of scanning the Index-Tree. Therefore, the running time of both methods increases when we increase the mean length of intervals. The running time of the Index-Tree-Method algorithm scales better than that of the Apriori-modified algorithm with respect to the mean length of intervals.

Fig. 15 Runtime vs. number of unit intervals ($d = 10K$, $m = 5\%$, $s = 5\%$).

In Figure 14, we compare the running time against the minimum support threshold for both methods. The support threshold ranges from 0.01% to 30%, the number of transactions is 100K, the number of unit intervals is 100 and the mean length of each interval is 5%. When the support threshold decreases, the number of candidate intervals increases dramatically. It also increases the number of scanning the database in the Apriori-modified algorithm as well as that of scanning the Index-Tree in the Index-Tree-Method algorithm. Therefore, the running time of both methods increases enormously when we decrease the minimum support threshold. Because the

Fig. 16 Runtime vs. number of dimensions ($d = 10K$, $n = 10$, $m = 40\%$, $s = 30\%$).

Index-Tree is more condensed than the database, the running time of the Index-Tree-Method algorithm scales better than that of the Apriori-modified algorithm with respect to the minimum support threshold.

We take a different approach to generate data when evaluating the effect of the number of unit intervals. To ensure fairness, we enforce all the data are in the same distribution. That is, we first generate data with $D = 10K$, $n = 500$ and $m = 5\%$ using the procedure described previously. Then we expand the number of unit intervals by dividing each unit interval into several equal-sized unit intervals. The original interval in each transaction is also transformed according to the new unit intervals. The factor of division is varied from 1 to 10 and the result is shown in Figure 15.

When the number of unit intervals increases, the number of candidate intervals increases significantly. The number of scanning the database in the Apriori-modified algorithm and that of scanning the Index-Tree in the Index-Tree-Method algorithm also increase. Therefore, the running time of both methods increases exponentially with respect to the number of unit intervals. However, the performance of the Index-Tree-Method algorithm still scales better than that of the Apriori-modified algorithm.

Fig. 17 Runtime vs. memory size ($d = 10K$, $n = 100$, $m = 5\%$, $s = 0.5\%$).

The next parameter is the number of dimensions. The method of generating data remains the same except that we only generate data for one dimension. The other dimensions are the same as the first dimension. This means the edges of the generated rectangles, cubes and hyper-cubes are all of equal length. The number of dimensions ranges from 1 to 5, the number of transaction is 10K, the number of unit intervals is 10, the mean length of interval is 40% and the minimal support threshold is 30%. The effect of a different number of dimensions is shown in Figure 16.

When the number of dimensions increases, the number of candidate intervals increases enormously. The number of scanning the database in the Apriori-modified algorithm and that of scanning the Index-Tree in the Index-Tree-Method algorithm also increase. Although the size of the Index-Tree is large when the number of dimensions is large, the Index-Tree is more condensed than the database, and the performance of the Index-Tree-Method algorithm is better than that of the Apriori-modified algorithm.

The last experiment about mining frequent intervals is to evaluate the performance of both methods when there exists a memory constraint. We only compare the running time of scanning and joining operations in both methods, because we assume that the Index-Tree has already been constructed on the disk. We want to show that even if an Index-Tree is constructed on the disk, the cost of scanning an Index-Tree is less than that of

scanning the database. The memory size ranges from 1K bytes to 5K bytes. The number of transactions is 10K, the number of unit intervals is 100, the mean lengths of intervals are 5% and the support threshold is 0.5%. The experiment result is shown in Figure 17.

When the available memory decreases, we have to fetch one block of data from the database or to fetch one segment of the disk-resident Index-Tree at one time. Besides, the memory consumed by generating candidate intervals may exceed the memory available. Therefore, we have to store them on the disk and swap them to count their frequencies and supports. As a result, the less memory we have, the more input–output operations will be required. However, the Index-Tree is more condensed than the database, so the performance of the Index-Tree-Method algorithm is better than that of the Apriori-modified algorithm as shown in Figure 17.

From the results of those experiments, we can see that the Index-Tree method outperforms the Apriori-modified method for all the cases.

6 Conclusions

A compact data structure Index-Tree is proposed and used the Index-Tree to store all the needed information. The Index-Tree is an extended prefix-tree structure which is capable of storing crucial, quantitative information about the intervals. The Index-Tree is compact in the following three aspects. First, since only frequent items will play a role in the frequent mining, the Index-Tree stores the set of frequent items for each transaction. Second, if multiple transactions contain equivalent itemsets, they are stored in the Index-Tree once and a counter is used to record the number of occurrences. Third, if any two transactions share a common prefix, the prefix items are stored in the Index-Tree once.

Some future work is shown as follows. Although an Index-Tree is compact, we have to construct it each time, as and when we mine frequent intervals. If we are going to regularly mine the same database, it could be more efficient to materialise the Index-Tree for later mining. Once an Index-Tree has been constructed, we can thereafter use the same Index-Tree to reduce the cost of reconstructing it unnecessarily. The cost of constructing an Index-Tree may sometimes be a non-trivial overhead. This is an interesting research-issue which concerns the question of solving problems using the Index-Tree structure.

References

- [1] Wen, F., Zhang, G., Sun, L., Wang, X., & Xu, X. (2019). A hybrid temporal association rules mining method for traffic congestion prediction. *Computers & Industrial Engineering*, vol. 130, pp.779-787.
- [2] Chen, C. H., Lan, G. C., Hong, T. P., & Lin, S. B. (2016). Mining fuzzy temporal association rules by item lifespans. *Applied Soft Computing*, vol. 41, pp. 265-274.
- [3] Wang, C., & Zheng, X. (2019). Application of improved time series Apriori algorithm by frequent itemsets in association rule data mining based on temporal constraint. *Evolutionary Intelligence*, vol. 132, pp.1-11.
- [4] Sarma, H. K. D., & Mishra, S. (2016, December). Mining time series data with Apriori tid algorithm. In 2016 International Conference on Information Technology (ICIT) (pp. 160-164). IEEE.
- [5] Wang, L., Meng, J., Xu, P., & Peng, K. (2018). Mining temporal association rules with frequent itemsets tree. *Applied Soft Computing*, vol. 62, pp. 817-829.
- [6] Agarwal, R. (2018). Ordering policy and inventory classification using temporal association rule mining. *International Journal of Productivity Management and Assessment Technologies (IJPMAT)*, vol.6, pp. 37-49.
- [7] Nguyen, D. T., & Khuat, B. D. L. (2019). Discovery of Temporal Association Rules in Multivariate Time Series. *International Journal of Applied Engineering Research*, vol.14, pp.79-84.
- [8] Li, Z., Bu, F., & Yu, F. (2017, July). Temporal fuzzy association rules mining based on fuzzy information granulation. In 2017 13th IEEE. International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD), pp.1168-1174.
- [9] Zhou, H., & Hirasawa, K. (2019). Evolving temporal association rules in recommender system. *Neural Computing and Applications*, vol. 31, 2605-2619.
- [10] Zhan, L., Yu, F., & Zhang, H. (2017, July). A fast algorithm for mining temporal association rules based on a new definition. In 2017 13th IEEE. International Conference on Natural Computation, Fuzzy Systems and Knowledge

Discovery (ICNC-FSKD) pp.1548-1553.

- [11] Yonggang, W. (2016). Sequential association rules based on apriori algorithm applied in personal recommendation. *International Journal of Database Theory and Application*, vol.9, pp. 257-264.
- [12] Radhakrishna, V., Kumar, P. V., & Janaki, V. (2016). An approach for mining similar temporal association patterns in single database scan. In *Proceedings of First International Conference on Information and Communication Technology for Intelligent Systems: Vo. 2*, pp. 607-617. Springer, Cham.
- [13] Guo, Y., Wang, M. and Li, X. (2017), Application of an improved Apriori algorithm in a mobile e-commerce recommendation system, *Industrial Management & Data Systems*, Vol. 117, pp. 287-303.
- [14] Krishna B., Amarawat G. (2019) Data Mining in Frequent Pattern Matching Using Improved Apriori Algorithm. In: Abraham A., Dutta P., Mandal J., Bhattacharya A., Dutta S. (eds) *Emerging Technologies in Data Mining and Information Security. Advances in Intelligent Systems and Computing*, vol 813, pp. 699-709. Springer, Singapore.
- [15] Bhandari, A., Gupta, A., & Das, D. (2015). Improvised Apriori Algorithm Using Frequent Pattern Tree for Real Time Applications in Data Mining, *Procedia Computer Science*, vol. 45, pp. 644-651.

This page is intentionally left blank