Efficient Mining of Frequent Sequence Generators

Chuancong Gao[†], Jianyong Wang[‡], Yukai He[§], Lizhu Zhou[¶]
Tsinghua University, Beijing, 100084, P.R.China
{[†]gaocc07, [§]heyk05}@mails.tsinghua.edu.cn,{[‡]jianyong,[¶]dcszlz}@tsinghua.edu.cn

ABSTRACT

Sequential pattern mining has raised great interest in data mining research field in recent years. However, to our best knowledge, no existing work studies the problem of frequent sequence generator mining. In this paper we present a novel algorithm, FEAT (abbr. Frequent sEquence generATor miner), to perform this task. Experimental results show that FEAT is more efficient than traditional sequential pattern mining algorithms but generates more concise result set, and is very effective for classifying Web product reviews.

Categories and Subject Descriptors: H.2.8 [Database Management]: Database applications - Data Mining

General Terms: Algorithms, Performance

Keywords: Sequence Generators, Sequence, Web Mining

1. INTRODUCTION

Sequential pattern mining has raised great interest in data mining research field in recent years. Various mining methods have been proposed, including sequential pattern mining[1][5], and closed sequentialpattern mining[7][6]. Sequential pattern mining has also shown its utility for Web data analysis, such as mining Web log data[2] and identifying comparative sentences from Web forum posting and product reviews[3]. However, there exists no existing work on mining frequent sequence generators, where a sequence generator is informally defined as one of the minimal subsequences in an equivalence class. Thus, generators have the same ability to describe an equivalence class as their corresponding subsequences of the same equivalence class, and according to the MDL principle[4], generators are preferable to all sequential patterns in terms of Web page and product review classification.

In the rest of this paper, we first give a formal problem formulation and focus on our solution in Section 2, then present the performance study in Section 3. We conclude the study in Section 4.

2. MINING SEQUENTIAL GENERATORS

2.1 Problem Formulation

An **input sequence database** SDB contains a set of input sequences, where an **input sequence** is an ordered list of items (each item can appear multiple times in a sequence) and can be denoted by S=$e_1e_2\ldots e_n$. Given a **prefix** of sequence S, S_{pre}=$e_1e_2\ldots e_i$, we define the **projected sequence** of S_{pre} w.r.t. S as $e_{i+1}e_2\ldots e_n$. The complete set of projected sequences of S_{pre} w.r.t. each sequence in SDB is called the **projected database** of S_{pre} w.r.t. SDB, denoted by $SDB_{S_{pre}}$. Given a subsequence $S_p = e_{p_1}e_{p_2}\ldots e_{p_m}$, its **support** $sup(S_p)$ is defined as the number of sequences in SDB_{S_p} each of which contains S_p, denoted by $|SDB_{S_p}|$. Given a user specified minimum support threshold, min_sup, S_p is said to be **frequent** if $sup(S_p) \geq min\_sup$ holds. S_p is called a **sequence generator** iff $\not\exists S'_p$ such that $S'_p \sqsubset S_p$ (i.e., S_p contains S'_p) and $sup(S_p) = sup(S'_p)$. In addition, given a sequence S=$e_1e_2\ldots e_n$ and an item e', we denote $e_1e_2\ldots e_{i-1}e_{i+1}\ldots e_n$ by $S^{(i)}$, $e_ie_{i+1}\ldots e_j$ by $S_{(i,j)}$, and $e_1e_2\ldots e_ne'$ by $<S,e'>$.

Given a minimum support threshold min_sup and an input sequence database SDB, the task of *frequent sequence generator mining* is to mine the complete set of sequence generators which are frequent in database SDB.

2.2 Pruning Strategy

A naïve approach to mining the set of frequent sequence generators is to first apply a sequential pattern mining algorithm to find the set of frequent subsequences and check if each frequent subsequence is a generator. However, it is inefficient as it cannot prune the unpromising parts of search space. In this subsection we propose two novel pruning methods, *Forward Prune* and *Backward Prune*, which can be integrated with the pattern-growth enumeration framework [5] to speed up the mining process. We first introduce Theorems 1 and 2 which form the basis of the pruning methods, but due to limited space we eliminate their proofs here.

THEOREM 1. *Given two sequences S_{p1} and S_{p2}, if $S_{p1} \sqsubset S_{p2}$ (i.e., S_{p1} is a proper subsequence of S_{p2}) and $SDB_{S_{p1}}$=$SDB_{S_{p2}}$, then any extension to S_{p2} cannot be a generator.* [1]

THEOREM 2. *Given subsequence S_p=$e_1e_2\ldots e_n$ and an item e', if SDB_{S_p} =$SDB_{S_p^{(i)}}(i = 1, 2, \cdots, n)$, then we have that $SDB_{<S_p,e'>}$= $SDB_{<S_p^{(i)},e'>}$.*

LEMMA 1. *(Forward Prune). Given subsequence S_p,and let S_p^*=$<S_p, e'>$, if $sup(S_p)$=$sup(S_p^*)$ and for any local frequent item u of S_p^* we always have $SDB_{<S_p,u>}$=$SDB_{<S_p^*,u>}$, then S_p^* can be safely pruned.*

PROOF. Easily derived from Theorem 1. □

LEMMA 2. *(Backward Prune). Given S_p=$e_1e_2\ldots e_n$, if there exists an index $i(i = 1, 2, \cdots, n-1)$ and a corresponding index $j(j$=i+$1, i$+$2, \cdots, n)$ such that $SDB_{(S_p)_{(1,j)}}$=$SDB_{((S_p)_{(1,j)})^{(i)}}$, then S_p can be safely pruned.*

PROOF. Easily derived from Theorem 2 and Theorem 1. □

[1]Note that a similar checking has been adopted in a closed sequential pattern mining algorithm, CloSpan [7]. Here we adapted the technique to the setting of sequence generator mining.

WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

2.3 Generator Checking Scheme

The preceding pruning techniques can be used to prune the unpromising parts of search space, but they cannot assure each mined frequent subsequence S=$e_1e_2 \dots e_n$ is a generator. We devise a generator checking scheme as shown in Theorem 3 in order to perform this task, and it can be done efficiently during pruning process by checking whether there exists such an index $i(i$=$1, 2, \cdots, n)$ that $|SDB_S|$=$|SDB_{S^{(i)}}|$, as $sup(S)$=$|SDB_S|$ holds.

THEOREM 3. *A sequence* S=$e_1e_2 \dots e_n$ *is a generator if and only if* $\nexists i$ *that* $1 \leq i \leq n$ *and* $sup(S)$=$sup(S^{(i)})$.

PROOF. Easily derived from the definition of generator and the well-known downward closure property of a sequence. □

2.4 Algorithm

By integrating the preceding pruning methods and generator checking scheme with a traditional pattern growth framework [5], we can easily derive the FEAT algorithm as shown in Algorithm 1. Given a prefix sequence S_P, FEAT first finds all its locally frequent items, uses each locally frequent item to grow S_P, and builds the projected database for the new prefix (lines 2,3,4). It adopts both the *forward* and *backward* pruning techniques to prune the unpromising parts of search space (lines 8,11), and uses the generator checking scheme to judge whether the new prefix is a generator (lines 7,9,11,12). Finally, if the new prefix cannot be pruned , FEAT recursively calls itself with the new prefix as its input (lines 14,15).

Algorithm 1: $FEAT(S_p, SDB_{S_p}, min\_sup, FGS)$

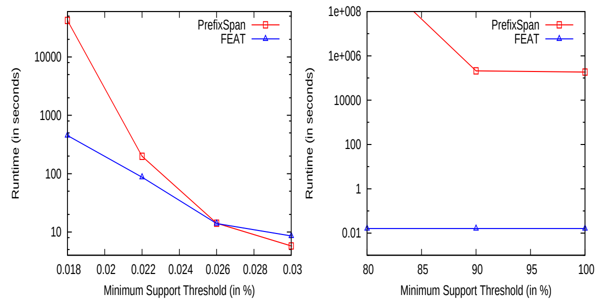
Input : Prefix sequence S_P, S_P's projected database SDB_{S_p}, minmum support min_sup, and result set FGS

1 **begin**
2 **foreach** i in $localFrequentItems(SDB_{S_p}, min\_sup)$ **do**
3 $S_p^i \leftarrow < S_p, i >$;
4 $SDB_{S_p^i} \leftarrow projectedDatabase(SDB_{S_p}, S_p^i)$;
5 $canPrune \leftarrow false$;
6 $isGenerator \leftarrow true$;
7 **if** $sup(SDB_{S_p}) = sup(SDB_{S_p^i})$ **then**
8 $canPrune \leftarrow ForwardPrune(S_p, SDB_{S_p}, S_p^i, SDB_{S_p^i})$;
9 $isGenerator \leftarrow false$;
10 **if** not $canPrune$ **then**
11 $BackwardPrune(S_p^i, SDB_{S_p^i}, canPrune, isGenerator)$;
12 **if** $isGenerator$ **then**
13 $FGS \leftarrow FGS \cup \{S_P^i\}$;
14 **if** not $canPrune$ **then**
15 $FEAT(S_P^i, SDB_{S_p^i}, min\_sup, FGS)$;
16 **end**

3. PERFORMANCE EVALUATION

We conducted extensive performance study to evaluate FEAT algorithm on a computer with Intel Core Duo 2 E6550 CPU and 2GB memory installed. Due to limited space, we only report the results for some real datasets. The first dataset, *Gazelle*, is a Web clickstream data containing 29,369 sequences of Web page views. The second dataset, *ProgramTrace*, is a program trace dataset. The third dataset, *Office07Review*, contains 320 consumer reviews for Office 2007 collected from Amazon.com, in which 240 and 80 reviews are labeled as positive and negative, respectively.

Figure 1 shows the runtime efficiency comparison between FEAT and PrefixSpan, a state-of-the-art algorithm for mining all sequential patterns. Figure 1a) demonstrates that FEAT is slightly slower than $PrefixSpan$ when the minimum support threshold is high for sparse dataset *Gazelle*, however, with a minimum support threshold less than 0.026%, FEAT is significantly faster than $PrefixSpan$.



a) $Gazelle$ b) $ProgramTrace$

Figure 1: Runtime Efficiency Comparison

This also validates that our pruning techniques are very effective, since without pruning FEAT needs to generate the same set of sequential patterns as PrefixSpan and perform generator checking to remove those non-generators, thus it should be no faster than PrefixSpan if the pruning techniques are not applied. Figure 1 b) shows that for dense dataset *ProgramTrace*, FEAT is significantly faster than PrefixSpan with any minimum support. For example, PrefixSpan used nearly 200,000 seconds to finish even at a minimum support of 100% , while FEAT costs less then 0.02 seconds.

We used generators and sequential patterns as features to build SVM and Naïve Bayesian classifiers respectively. The results for *Office07Review* dataset show that both generator-based and sequential pattern-based models achieve almost the same accuracy. For example, with a minimum support of 2% and a minimum confidence of 75%, both generator-based and sequential pattern-based Naïve Bayesian classifiers can achieve the same best accuracy of 80.6%. As generator-based approach is more efficient, it has an edge over sequential pattern-based approach in terms of efficiency.

4. CONCLUSIONS

In this paper we study the problem of mining sequence generators, which has not been explored previously to our best knowledge. We proposed two novel pruning methods and an efficient generator checking scheme, and devised a frequent generator mining algorithm, FEAT. An extensive performance study shows that FEAT is more efficient than the state-of-the-art sequential pattern mining algorithm, PrefixSpan, and is very effective for classifying Web product reviews. In future we will further explore its applications in Web page classification and click stream data analysis.

5. ACKNOWLEDGEMENTS

This work was partly supported by 973 Program under Grant No. 2006CB303103, and Program for New Century Excellent Talents in University under Grant No. NCET-07-0491, State Education Ministry of China.

6. REFERENCES

[1] R. Agrawal, R. Srikant. Mining Sequential Patterns. ICDE'95.

[2] J. Chen, T. Cook. Mining Contiguous Sequential Patterns from Web Logs. WWW'07 (Posters track).

[3] N. Jindal, B. Liu. Identifying Comparative Sentences in Text Documents. SIGIR'06.

[4] J. Li, et al. Minimum description length principle: Generators are preferable to closed patterns. AAAI'06.

[5] J. Pei, J. Han, et al. Prefixspan: mining sequential patterns effficiently by prefix-projected pattern growth. ICDE'01.

[6] J. Wang,J. Han. BIDE:efficient mining of frequent closed sequences. ICDE'04.

[7] X. Yan, J. Han, R. Afshar. CloSpan: Mining closed sequential patterns in large datasets. SDM'03.