Structure Mining: a Sequential Pattern Based Approach

C. Garboni and F. Masseglia

INRIA Sophia Antipolis
Project-Team AxIS
2004 route des lucioles - BP 93
06902 Sophia Antipolis, France

E-mail: {Calin.Garboni, Florent.Masseglia}@sophia.inria.fr

Abstract. The KDD[1] process implies several steps such as the preprocessing, the data mining step and finally the postprocessing of the results. The data mining step has been considered as the most challenging of this process and received a great deal of attention. Nevertheless, when applying a data mining technique to a raw set of data, in the context of a real case KDD process, one may have to solve preprocessing problems. Actually, in most cases, an expert knowing both the dataset and the data mining methods will be asked to prepare the data. In this paper, we propose a method for preprocessing the data by extracting the structure and parsing the data thanks to this result. Our experiments show that we extract the desired information in a reasonable time that classical methods would not reach.
Keywords: KDD, data mining, structure mining, sequential patterns, preprocessing, parsing.

1 Introduction

Since the definition of the KDD process, many work has been provided. Most of it deals with the data mining step, which was first considered as the most challenging part of this process (optimization of the extraction techniques, data mining algorithm efficiency and effectiveness, ...). This point of view has now evolved to better consider the steps coming before and after the data mining process. These steps are the preprocessing of the data and the post-processing of the results. The former mainly includes data cleaning, selection and transformation while the latter provides visualization tools, sorting algorithms for the provided result, etc.
In this paper we will focus on the preprocessing step. Actually, considering the actual representation of the KDD process, given in Figure 1, we are working on the automation of the preprocessing step (within the dash lines). This is motivated by the fact that today, in order to perform a knowledge discovery process on any kind of data, one has to transform the data from the original raw format to a specific format that will be understood by the data mining algorithm. This transformation is usually designed by an expert which has the required knowledge about the data and about the data

[1] Knowledge Discovery in Databases

mining algorithm. Our goal is to help and even replace the expert by providing an intelligent tool, able to "understand" the structure of the data to prepare on the one hand, and to propose a parsing over the data (based on the discovered structure) on the other hand. For this purpose we will first work on an access log file and try to discover the structure of this file. Even if this structure is already well known, this will be a bench for our proposal. The method will be based on the sequential pattern mining principle. Actually, mining sequential patterns aims at providing the frequent sequences hidden in the data. The structure in a file organizing the records line by line can be considered has a frequent sequence repeated in (almost) each record. We discuss this comparison in section 2.3. The paper is organized as follows: first we describe the required notions for this contribution (definitions of KDD, access logs files and sequential pattern mining) in section 2. we provide an overview of the contributions in the literature related to sequential pattern and structure extraction in section 3. In section 4 the solution we propose is presented and we give the experimental results in section 5. Finally we conclude the paper and give some future research issues.

2 Definitions

In this section we explain the notion of knowledge discovery in databases and give an overview of access log files. The sequential pattern mining problem in large databases is stated and an illustration is given.

2.1 Knowledge Discovery in Databases

Knowledge discovery is defined as "the non-trivial extraction of implicit, unknown, and potentially useful information from data" [5]. Figure 1 illustrates the KDD process. In [6] (p5-7) an overview of this process is given. We can give three main steps for the knowledge discovery process : data preparation, data mining and pattern evaluation. Data preparation includes (see [6] for more details):

- Data cleaning (to remove noise and inconsistent data)
- Data integration (combination of multiple data sources)
- Data selection
- Data transformation (where data are transformed or consolidated into forms appropriate for mining)

Our contribution focuses on the data preparation process since we first want to understand the structure of the data to analyze.

2.2 Access Logs

Raw data is collected in access log files by Web servers. Each input in the log file illustrates a request from a client machine to the server (*http daemon*). Access log files format can differ, depending on the system hosting the Web site. For the rest of this presentation we will focus on three fields: client address, the URL asked for by the user and the time and date for that request. We illustrate these concepts

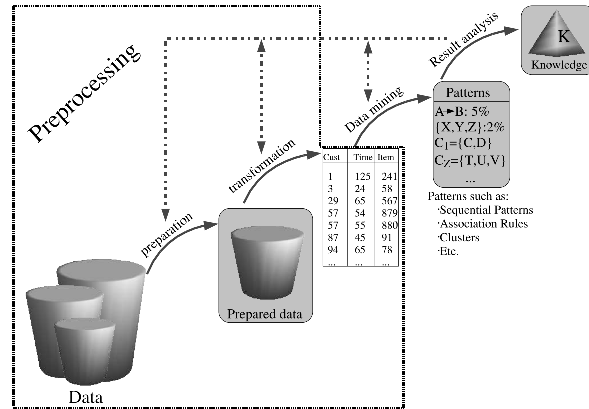

Fig. 1. The usual KDD process

with the access log file format given by the CERN and the NCSA [11], where a log input contains records made of 7 fields, separated by spaces [9]:**host user authuser [date:time] "request" status bytes**
Figure 2 is a sample, coming from one of the INRIA's access log files.

138.96.80.1 - - [06/Mar/2003:19:51:28 +0100] "HEAD /prisme/personnel/da/da.html HTTP/1.1" 200 0 "-" "libwww-perl/5.65"

138.96.80.21 - - [21/Mar/2003:10:19:24 +0100] "HEAD /odyssee/team/index.fr.html HTTP/1.1" 200 0 "-" "libwww-perl/5.65"

80.179.100.98 - - [28/Mar/2003:02:46:15 +0100] "GET /mimosa/mobility_list/threads.html HTTP/1.1" 200 15466 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows 98)"

Fig. 2. Example of an Access log file

The access log file is then processed in two steps. First of all, the access log file is sorted by address and by transaction. Then each "uninteresting" data is pruned out from the file. During the sorting process, in order to allow the knowledge discovery process to be more efficient, URLs and clients are mapped into integers. Each time and date is also translated into relative time, compared to the earliest time in the log file.

Definition 1. *Let Log be a set of server access log entries.*
An entry g, $g \in Log$, is a tuple $g =< ip_g, ([l_1^g.URL, l_1^g.time] \ ... \ [l_m^g.URL, l_m^g.time]) >$

such that for $1 \leq k \leq m$, $l_k^g.URL$ *is the item asked for by the user* g *at time* $l_k^g.time$ *and for all* $1 \leq j < k$, $l_k^g.time > l_j^g.time$.

2.3 Sequential Pattern Mining

In [1], the association rules mining problem is defined as follows:

Definition 2. *Let* $I = \{i_1, i_2, ..., i_m\}$, *be a set of* m *literals (items). Let* $D = \{t_1, t_2, ...t_n\}$, *be a set of* n *transactions ; Associated with each transaction is a unique identifier called its* TID *and an itemset* I. I *is a k-itemset where* k *is the number of items in* I. *We say that a transaction* T contains X, *a set of some items in* I, *if* $X \subseteq T$. *The support of an itemset* I *is the fraction of transactions in* D *containing* I: $supp(I) = \|\{t \in D \mid I \subseteq t\}\|/\|\{t \in D\}\|$. *An association rule is an implication of the form* $I_1 \Rightarrow I_2$, *where* I_1, $I_2 \subset I$ *and* $I_1 \cap I_2 = \emptyset$. *The rule* $I_1 \Rightarrow I_2$ *holds in the transaction set* D *with confidence* c *if* $c\%$ *of transactions in* D *that contain* I_1 *also contain* I_2. *The rule* $r : I_1 \Rightarrow I_2$ *has* support s *in the transaction set* D *if* $s\%$ *of transactions in* D *contain* $I_1 \cup I_2$ *(i.e.* $supp(r) = supp(I_1 \cup I_2)$*).*
Given two parameters specified by the user, minsupp and minconfidence, the problem of association rule mining in a database D aims at providing the set of frequent itemsets in D, i.e. all the itemsets having support greater or equal to minsupp. Association rules with confidence greater than minconfidence are thus generated.

As this definition does not take time into consideration, the sequential patterns are defined in [10]:

Definition 3. *A sequence is an ordered list of itemsets denoted by* $< s_1 s_2 \ldots s_n >$ *where* s_j *is an itemset. The data-sequence of a customer* c *is the sequence in* D *corresponding to customer* c. *A sequence* $< a_1 a_2 \ldots a_n >$ *is a* subsequence *of another sequence* $< b_1 b_2 \ldots b_m >$ *if there exist integers* $i_1 < i_2 < \ldots < i_n$ *such that* $a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, \ldots, a_n \subseteq b_{i_n}$.

Example 1. Let C be a client and S=$<$ (3) (4 5) (8) $>$, be that client's purchases. S means that "C bought item 3, then he or she bought 4 and 5 at the same moment (i.e. in the same transaction) and finally bought item 8".

Definition 4. *The* support *for a sequence* s, *also called* $supp(s)$, *is defined as the fraction of total data-sequences that contain* s. *If* $supp(s) \geq minsupp$, *with a minimum support value* $minsupp$ *given by the user,* s *is considered as a frequent sequential pattern.*

Our contribution is based on a comparison between sequential patterns and structures. The structure is common to all records in the datasets. Nevertheless, in numerous kind of records, such as log file entries, the data can be altered (due to errors while recording the entry, a system crash, etc.). The structure can thus be considered as a sequential pattern having a very high support over the dataset. Let us consider the lines given in figure 2. If each line is considered as a customer and each character is considered as an itemset then the sequential pattern having a support of 100% would be `... - - [/Mar/2003::: +000] "/o/d.html HTTP/1." 0 "" ""` which is

common to all the lines. the `/o/d.html` part of the structure is due to the fact that the 3 lines given in example 2 are not a large enough sample to find the real structure. Applying this comparison on a larger sample would give a correct structure.

3 Related Work

3.1 Sequential Pattern Mining

The traditional solutions to the sequential pattern mining problem are based generally on the Apriori principle (like GSP[2]). The algorithm generates candidate itemsets (patterns) of length k from k - 1 length itemsets. Then, the patterns which have an infrequent sub pattern are pruned. The other solutions are also based on a bottom-up method ([8],[3]).

Based on the Apriori property, proposed in the same paper: any super-pattern of a non frequent pattern cannot be frequent, GSP [2] adopts a multiple-pass, candidate-generation-and-test approach in sequential pattern mining. This is outlined as follows: The first scan finds all the frequent items. Each subsequent pass starts with a seed set of sequential patterns, which is the set of sequential patterns found in the previous scan. This seed is used to generate a new potential pattern set, called candidate sequences. All the candidates whose support in the database in no less than min_support form the set of a the newly found sequential patterns. The algorithm terminates when no new sequential pattern is found in the scan, or no candidate sequence can be generated. The Apriori-like sequential pattern mining methods have generally the same cost: potentially huge set of candidate sequences; multiple scans of databases; difficulties at mining long sequential patterns. The PSP algorithm [8] adopts the general principles defined by the GSP algorithm but proposes a different data structure for storing candidate and frequent sequences. This method is based on depth-first strategy, so it is not adapted for our problem (i.e. the long pattern extraction).

SPAM [3] is an algorithm for mining sequential patterns, based on search strategy of vertical bitmap representation of the database. SPAM utilizes also a depth-first traversal of the search space, in a generating-pruning manner. This is why this method is not adapted at our problem. As with PSP, we haven't obtained the desired structure in our experiments.

In [13], the CloSpan algorithm, specific to closed sequential pattern mining, is proposed. A sequential pattern is called closed when it is not containe in another sequential pattern having the same support. The optimizations for this method consist in avoiding the non closed patterns. In our case, the short patterns have a higher support, so the CloSpan optimizations are not very useful for our problem.

In [7] the authors propose an approximate sequential pattern mining roughly defined as identifying patterns approximately shared by many sequences. The method works in three steps: first, similar sequences are grouped together using kNN clustering; then

the method organizes and compresses sequences within each cluster into "weighted sequences" using multiple alignment. In the last step, the longest consensus pattern best fitting each cluster are generated from the "weighted sequences". We did not test this method for our data, but the clusters used by the ApproxMap seem to be smaller than the access log files and the alignment will certainly have the problems of sequences size and their large number.

3.2 Structure Discovery

In [4], the authors propose an approach for measuring the structural similarities between XML documents, represented as time series. They give the structural similarities between the two documents by exploiting the Discrete Fourier Transform of the corresponding signals. Our problem consists in processing shortest candidates but the set of entries is far more larger.
[12] studies the way in which the adaptive techniques used in text compression can be applied to text mining. This approach is applied also on the structure recognition. It uses the token discrimination, improved with the "soft parsing", based on the compression methodology. In our context, the structure is much more present, and the result must be more accurate.

4 SINTHES: an Algorithm for Structure Extraction

In order to propose a first method for structure discovery and to validate it, we used the access log files. The goal is not to find the structure of those files, because we already now it, but to develop a new method which could be applied to other data having the same characteristics: high density, semi structure, considerable length. Our test consist in finding a long sequential pattern which will be the structure of the log file. In this section we present a new method which avoids the disadvantages of the existent solutions for structure discovery, using a sequential pattern mining approach.

4.1 Naive Approach

A naive approach for discovering structures would consist in the following: (1) Select a random line and use it as a candidate; (2) Find the support of all the subsequences of size $k-1$ from a candidate of size k. The most frequent subsequence become the new candidate. This top-down generating-pruning method has some disadvantages: if all the subsequences have the same support, the algorithm will randomly select the new candidate. Another problem is the execution time for a very long candidate. This implies a late start of the method, i.e. it must have a certain number of random iterations before the increase of the support as it can be seen in the Figure 3. This is why we propose some filters, to avoid the disadvantages of the naive solution.

4.2 Applying Successive Filters

For the structure extraction from the data, we propose an heuristic method which, as the naive approach, initializes a random line as a candidate sequence (in the hypothesis

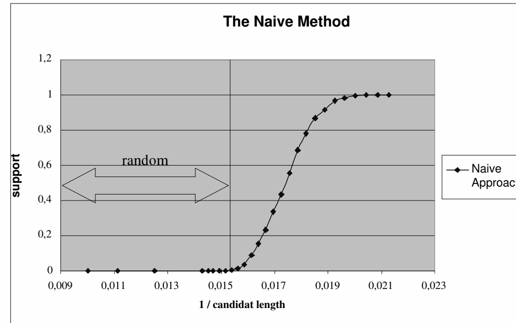

Fig. 3. The Naive Method

that it contains the structure) and prunes the elements that doesn't belong to the structure. We called our method SINTHES (Structure is IN THE Sequence). This method is based on successive filters for the candidate, followed by a top-down stage of generating-pruning approach.

The Frequent Items Filter In this stage of the algorithm, we select all the frequent items, with a support higher than minimum support. Based on this characters set, we filter the candidate which is, initially a randomly chosen line in the log file.

Example 2. In this example we have the frequent items from a log file, with a support higher than the minimum support, defined by the user. This is, in our case, the set of characters which appear in 95 % of the lines.

minimumSupport=0.95
Frequent Items:
< > <"> <+> <-> <.> </> <0> <1> <2> <3> <4> <:> <E> <G> <H>
<M> <P> <T> <[> <]> <a> <e> <i> <l> <m> <n> <o> <p> <r> <s> <t>

Example 3. We can observe here the reduction of a 102 characters long line to a 85 characters long line with the frequent items filter.

The candidate before the "frequent items filter":
195.154.174.164 - - [25/Mar/2003:12:03:24 +0100] "GET /epidaure/AISIM/ HTTP/1.0" 200 5679 "-" "NG/1.0"
- candidate size: 102

The candidate after the "frequent items filter":

1.14.14.14 - - [2/Mar/2003:12:03:24 +0100] "GET /epiare/M/ HTTP/1.0" 200 "-" "G/1.0"
- candidate size: 85

We keep in this document the candidate from this example for showing the effect of each applied filter.

The Precedence Matrix Filter The precedence matrix is a binary matrix containing the mutual relationship informations between each two items. An element A[i][j] is defined as follow:

$$A[i][j] = \begin{cases} 1 \text{ if (i,j) is frequent} \\ 0 \text{ if (i,j) is not frequent} \end{cases}$$

Example 4. This filter allows a new reducing of the seek space (range). The second occurrence of character "**M**" will be removed because of the character "+" (A[+][M]=0). this test will be repeated for each character untill stability of the candidate.

The Candidate before the "Precedence Matrix Filter":
1.14.14.14 - - [2/Mar/2003:12:03:24 +0100] "GET /epiare/**M**/ HTTP/1.0" 200 "-" "G/1.0"
The Candidate after the "Precedence Matrix Filter":
... - - [/Mar/2003:12:03: +0100] "GET /// HTTP/1.0" 00 "" "/1.0"
- candidate size: 65

The Longest Common Subsequence Filter This new stage of filtering consists in the computing of the longest common subsequence between the candidate and each line of the file. We try then to detect the items that make the difference between the candidate and the longest common subsequence.

Example 5. Let us consider a part of our candidate sequence (reduced at the date between the characters " [" et "] "). Let us assume that the first line in the file contains the date: "[/Mar/2003:06:55 +0100]". The longest common subsequence between our candidate and the first line is: "[/Mar/2003:: +0100]"(cf. Fig.4). The characters "1" "2" "0" et "3" will be penalized because they are not recognized in this line. After a certain number of penalties, these characters could be dynamically removed from our candidate. Line 10, for example, allows the removement of character "2" from the candidate sequence because it accumulated to many penalties. Character "3" will probably be removed after line 10 due to its high penalty level. At the end of this stage, the candidate is close to the structure.

The candidate before the "LCS Filter " :
... - - [/Mar/2003:12:03: +0100] "GET /// HTTP/1.0" 00 "" "/1.0"

Line 1	[	/	M	a	r	/	2	0	0	3	:	0	6	:	5	5		+	0	1	0	0	]
Candidate	[	/	M	a	r	/	2	0	0	3	:	1	2	:	0	3		+	0	1	0	0	]
LCS	[	/	M	a	r	/	2	0	0	3	:			:				+	0	1	0	0	]
Penalty	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	0	0	0	0	0	0
Line 2	[	/	M	a	r	/	2	0	0	3	:	1	3	:	1	0		+	0	1	0	0	]
Candidate	[	/	M	a	r	/	2	0	0	3	:	1	2	:	0	3		+	0	1	0	0	]
LCS	[	/	M	a	r	/	2	0	0	3	:	1		:				+	0	1	0	0	]
Penalty	0	0	0	0	0	0	0	0	0	0	0	1	2	0	2	2	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Line 9	[	/	M	a	r	/	2	0	0	3	:	2	7	:	1	1		+	0	1	0	0	]
Candidate	[	/	M	a	r	/	2	0	0	3	:	1	2	:	0	3		+	0	1	0	0	]
LCS	[	/	M	a	r	/	2	0	0	3	:			:				+	0	1	0	0	]
Penalty	0	0	0	0	0	0	0	0	0	0	0	6	9	0	7	8	0	0	0	0	0	0	0
Line 10	[	/	M	a	r	/	2	0	0	3	:	2	8	:	1	2		+	0	1	0	0	]
Candidate	[	/	M	a	r	/	2	0	0	3	:	1		:	0	3		+	0	1	0	0	]
LCS	[	/	M	a	r	/	2	0	0	3	:			:				+	0	1	0	0	]
Penalty	0	0	0	0	0	0	0	0	0	0	0	7		0	8	9	0	0	0	0	0	0	0

Fig. 4. The Longest Common Subsequences Filter

The candidate after the "LCS Filter" :
... - - [/Mar/2003::: +000] "GET // HTTP/1." 0 "" "."
- The new candidate size: 54

The Top-Down Generating-Pruning Filter The result of the last filter is a candidate of size k. this candidate will be the seed for all the subsequences of size $k-1$. For each one we determine the frequency and the most frequent subsequence will be chosen for the next iteration. The algorithm will continue until the support of one subsequence is higher than the minimum support and that will be the candidate representing the structure.

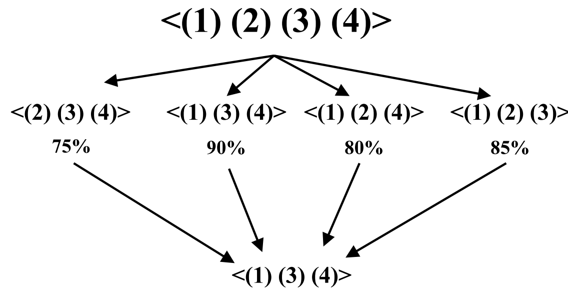

Fig. 5. The Top-Down Generating-Pruning Method

Example 6. Let us consider the sequence (1)(2)(3)(4) from Figure 5. According to the described method, this sequence of size 4 will generate all its subsequences of size 3: (2)(3)(4), (1)(3)(4), (1)(2)(4), (1)(2)(3), and for each one, we search the support. In our example (1)(3)(4) is the most frequent subsequence and, if its support is not higher than the minimum support, this sequence will initialize the next iteration (i.e. generating the subsequences set: {<(3)(4)>,<(1)(4)>,<(1)(3)>})

Example 7. Using this method, the final result is a sequence with a support higher than the minimum support and which represent the structure of a log file line:
Structure found:
... - - [/Mar/2003::: +000] "GET / HTTP/1." 0 "" ""
- structure size: 52

5 Experiments

The experiments were performed on an AMD PC workstation with a CPU rate of 1,1GHz, 256 Mb of main memory. The SINTHES algorithm was implemented in Java and compiled with java compiler from j2sdk1.4.0.

In order to evaluate our algorithm, we tested the capacity of providing the result on two coordinates: the file size and the support. We also used two versions of the algorithm. The difference consists in the non existence, in the initial version, of the longest common subsequence filter. This shows the utility in the performance and in the result quality of this filter. The results are reported in figure 6. We can observe that the execution time is reduced when using the dynamic counting filter (implementing LCS). The experiment has been performed on two datasets having size of 800Kb and 8Mb respectively. Using SINTHES on the whole dataset (8Mb) allows to discover the structure in 100 to 220 seconds (depending on the specified support). When a sampling is performed over the dataset (800Kb) the structure is extracted in 13 to 25 seconds.

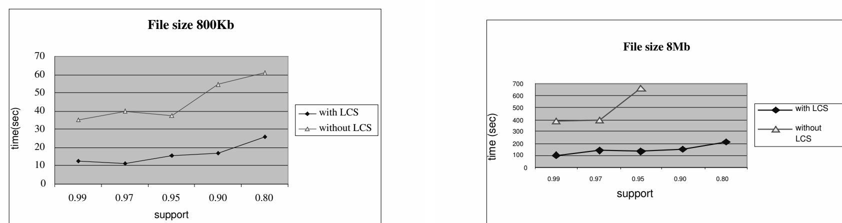

Fig. 6. Execution time for the SINTHES method with and without the LCS filter.

In order to show that the SINTHES method linearly depends on the dataset size, we performed a test involving the size of the log file. Figure 7 shows the linearity on the method in the terms of file size and execution time.

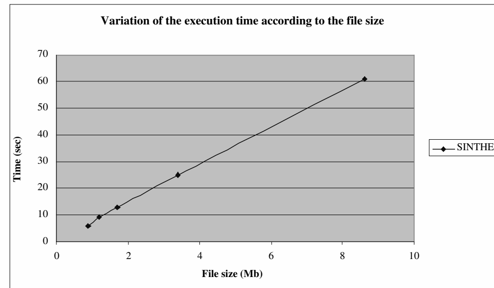

Fig. 7. Linearity of the SINTHES method.

Finally, on a log file regarding the format given in this paper, the discovered structure is that given in example 7. It is slightly different from the given format but gives interesting information about the recorded data. For instance, the "-' character is occuring twice at the beginning of the proposed structure, which shows that the corresponding values are never (or rarely) filled in.

6 Conclusion

In this paper we proposed a study on the preprocessing step of a KDD process. Without providing solutions to the selection or cleaning problems related to this step, our goal was to show that finding the structure of the data could be automated. For this purpose we made a link between the structure (common to most lines of the dataset) and a sequential pattern. Then we proposed a heuristic called SINTHES to extract the structure from the data. Starting from a sample dataset and assuming that the structure is included in this dataset, SINTHES is based on successive filters designed to rapidly remove the characters not belonging to the structure. Our experiments showed that SINTHES performs very well and the structure is obtained where other sequential pattern mining methods would fail. We plan to extend this work to other kind of data. The log files are organized such as each line provides a record, but we are interested into applying the proposed top-down method to more complex data, such as XML or HTML files for instance.

References

1. R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proceedings of the 1993 ACM SIGMOD Conference*, pages 207–216, Washington DC, USA, May 1993.
2. R. Agrawal and R. Srikant. Fast Algorithms for Mining Generalized Association Rules. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB'94)*, Santiago, Chile, September 1994.
3. J. Ayres and J. Flannick. Sequential pattern mining using a bitmap representation. In *SIGKDD*, pages 429–435, 2002.
4. Sergio Flesca and al. Detecting Structural Similarities between XML Documents. In *Fifth International Workshop on the Web and Databases (WebDB 2002)*, Madison, Wisconsin, 2002.
5. W.J. Frawley, G. Piatetsky-Shapiro, and C. Matheus. Knowledge Discovery In Databases: An Overview. *Knowledge Discovery In Databases*, pages 1–30, 1991.
6. J. Han and M. Kamber. *Data Mining, Concepts and Techniques.* Morgan Kaufmann, 2001.
7. H. C. Kum, J. Pei, W. Wang, and D. Duncan. Approxmap: Approximate mining of consensus sequential patterns. In *Proceedings of the SIAM International Conference on Data Mining*, San Francisco, CA, 2003.
8. F. Masseglia, F. Cathala, and P. Poncelet. The PSP Approach for Mining Sequential Patterns. In *Proceedings of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD'98), LNAI, Vol. 1510*, pages 176–184, Nantes, France, September 1998.
9. C. Neuss and J. Vromas. *Applications CGI en Perl pour les Webmasters.* Thomson Publishing, 1996.
10. R. Srikant and R. Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proceedings of the 5th International Conference on Extending Database Technology (EDBT'96)*, pages 3–17, Avignon, France, September 1996.
11. W3C. httpd-log files. In *http://www.w3.org/Daemon/User/Config/Logging.html*, 1995.
12. Ian H. Witten. Adaptive Text Mining: Inferring Structure from Sequences. In *2003 SIAM Int. Conf. Data Mining (SDM 03)*, San Fransisco, CA, 2003.
13. X. Yan, J. Hah, and R. Afshar. Clospan: Mining Closed Sequential Patterns in Large Datasets. In *J Discrete Algorithms*, 2004.