Mining Sequential Rules Based on Prefix-Tree

Thien-Trang Van[1], Bay Vo[1], and Bac Le[2]

[1] Faculty of Information Technology, Ho Chi Minh City University of Technology, Vietnam
[2] Faculty of Information Technology University of Science, Ho Chi Minh, Vietnam
{vtttrang,vdbay}@hcmhutech.edu.vn, lhbac@fit.hcmus.edu.vn

Abstract. We consider the problem of discovering sequential rules between frequent sequences in sequence databases. A sequential rule expresses a relationship of two event series happening one after another. As well as sequential pattern mining, sequential rule mining has broad applications such as the analyses of customer purchases, web log, DNA sequences, and so on. In this paper, for mining sequential rules, we propose two algorithms, *MSR_ImpFull* and *MSR_PreTree*. *MSR_ImpFull* is an improved algorithm of *Full* (David Lo et al., 2009), and *MSR_PreTree* is a new algorithm which generates rules from frequent sequences stored in a prefix-tree structure. Both of them mine the complete set of rules but greatly reduce the number of passes over the set of frequent sequences which lead to reduce the runtime. Experimental results show that the proposed algorithms outperform the previous method in all kinds of databases.

Keywords: frequent sequence, prefix-tree, sequential rule, sequence database.

1 Introduction

Mining sequential patterns from a sequence database was first introduced by Agrawal and Srikant in [1] and has been widely addressed [2-6, 10]. Sequential pattern mining is to find all frequent sequences that satisfy a user-specified threshold called the minimum support (minSup).

Sequential rule mining is trying to find the relationships between occurrences of sequential events. A sequential rule is an expression that has form X→Y, i.e., if X occurs in any sequence of the database then Y also occurs in that sequence following X with high confidence. In sequence databases, there are researches on many kinds of rules, such as recurrent rules [8], sequential classification rules [14], sequential rules [7, 9] and so on. In this paper, we focus on sequential rule mining. In sequential rule mining, there are researches on non-redundant sequential rules but not any real research on mining a full set of sequential rules. If we focus on interestingness measures, such as lift [12], conviction [11], then the approach of non-redundant rule [9] cannot be used. For this reason, we try to address the problem of generating a full set of sequential rules effectively.

Based on description in [7], the authors of paper [9] have generalized an algorithm for mining sequential rules, called *Full*. The key feature of this algorithm is that it requires multiple passes over the full set of frequent sequences. In this paper we

N.T. Nguyen et al. (Eds.): New Challenges for Intelligent Information, SCI 351, pp. 147–156.
springerlink.com

present two algorithms: *MSR_ImpFull* and *MSR_PreTree*. The former is an improved algorithm (called *MSR_ImpFull)*. The latter is a new algorithm (called *MSR_PreTree*) which effectively mines all sequential rules base on prefix-tree.

The rest of paper is organized as follows: Section 2 presents the related work and Section 3 introduces the basic concepts related to sequences and some definitions used throughout the paper. In section 4, we present the prefix-tree. Two proposed algorithms are presented in Section 5, and experimental results are conducted in Section 6. We summarize our study and discuss some future work in Section 7.

2 Related Work

The study in [7] has proposed generating a full set of sequential rules from a full set of frequent sequences and removing some redundant rules by adding post-mining phase.

Authors in [9] have investigated several rule sets based on composition of different types of sequence sets such as generators, projected-database generators, closed sequences and projected- database closed sequences. Then, they have proposed a compressed set of non-redundant rules which are generated from two types of sequence sets: LS-Closed and CS-Closed. The premise of the rule is a sequence in LS-Closed set and the consequence is a sequence in CS-Closed set. The authors have proved that the compressed set of non-redundant rules is complete and tight, so they have proposed an algorithm for mining this set. For comparison, based on description in [7] they also have generalized *Full* algorithm for mining a full set of sequential rules.

In this paper, we are interested in mining a full set of sequential rules because non-redundant sequential rule mining just minds the confidence, if we use alternative measures then this approach cannot be appropriate as mentioned in Section 1.

3 Preliminary Concepts

Let I be a set of distinct items. An itemset is a subset of items (without loss of generality, we assume that items of an itemset are sorted in lexicographic order). A sequence $s = \langle s_1\ s_2\ \dots\ s_n \rangle$ is an ordered list of itemsets. The size of a sequence is the number of itemsets in the sequence. The length of a sequence is the number of items in the sequence. A sequence with length k is called a k-sequence.

A sequence $\beta = \langle b_1\ b_2\ \dots b_m \rangle$ is called a subsequence of another sequence $\alpha = \langle a_1\ a_2 \dots\ a_n \rangle$ if there exist integers $1 \le i_1 < i_2 < \dots < i_m \le n$ such that $b_1 \subseteq a_{i1}$, $b_2 \subseteq a_{i2}$, …, $b_m \subseteq a_{im}$. Given a sequence database, the support of a sequence α is defined as the number of sequences in the sequence database that contains α. Given a minSup, we say that a sequence is frequent if its support is greater than or equal to minSup.

Definition 1. *(Prefix, incomplete prefix and postfix).* Given a sequence $\alpha = \langle a_1\ a_2\ \dots\ a_n \rangle$ and a sequence $\beta = \langle b_1\ b_2\ \dots\ b_m \rangle\ (m<n)$, (where each a_i, b_i corresponds to an itemset). β is called prefix of α if and only if $b_i = a_i$ for all $1 \le i \le m$. After removing the prefix β of sequence α, the remaining part of α is a postfix of α. Sequence β is called

an incomplete prefix of α if and only if $b_i=a_i$ for $1 \le i \le m-1$, $b_m \subset a_m$ and all the items in (a_m-b_m) are lexicographically after those in b_m.

Note that from the above definition, a sequence of size k has $(k-1)$ prefixes. For example, sequence ⟨(A)(BC)(D)⟩ have two prefixes: ⟨(A)⟩, ⟨(A)(BC)⟩. Consequently, ⟨(BC)(D)⟩ is the postfix with respect to prefix ⟨(A)⟩ and ⟨(D)⟩ is the postfix w.r.t prefix ⟨(A)(BC)⟩. Neither ⟨(A)(B)⟩ nor ⟨(BC)⟩ is considered as a prefix of given sequence, however, ⟨(A)(B)⟩ is a incomplete prefix of given sequence.

A sequential rule is built by splitting a frequent sequence in two parts: prefix *pre* and postfix *post* (concatenating *pre* with *post*, denoted as *pre++post*, we have the same pattern as before [9]). We denote a sequential rule as $r=pre \rightarrow post$ *(sup, conf)*.

- The support of r: $sup = sup(pre++post)$.
- The confidence of r: $conf = sup(pre++post)/sup(pre)$.

Note that *pre++post* is a frequent sequence, consequently *pre* is also a frequent sequential pattern (by the Apriori principle [1]). For each frequent sequence f of size k, we can possibly form $(k-1)$ rules. For example, if we have a frequent sequence ⟨(A)(BC)(D)⟩ which size is 3, then we can generate 2 rules such as ⟨(A)⟩→⟨(BC)(D)⟩, ⟨(A)(BC)⟩→⟨(D)⟩.

Sequential rule mining is to find out all significant rules that satisfy minSup and minimum confidence (minConf) from the given database. Usually thresholds of support and confidence are predefined by users.

Similar to association rule mining, this problem can be decomposed into two sub problems. The first problem is to find all frequent sequences (*FS*) that satisfy minSup, and the second is to generate sequential rules from those frequent sequences with satisfying minConf.

4 Prefix -Tree

In our approach *(MSR_PreTree)*, the set of rules is generated from frequent sequences, which is organized and stored in a prefix-tree structure as illustrated in Fig. 1. In this section, we briefly outline the prefix-tree (similar to lexicographic tree [5, 6, 10]).

Starting from the root of tree at level *0*, root is labeled with a null sequence ϕ. At level k, a node is labeled with a k-sequence. Recursively, we have nodes at the next level $(k+1)$ by extending k-sequence with a frequent item. There are two ways to extend a k-sequence: *sequence extension* and *itemset extension* [6].

In sequence extension, we add an item to the sequence as a new itemset. Consequently, the size of sequence-extended sequences always increases.

Remark 1: In sequence extension, a k-sequence α is a prefix of all sequence-extended sequences. Moreover, α is certainly the prefix of all sub nodes of the nodes which are sequence-extended of α.

In itemset extension, the item is added to the last itemset in the sequence so that the item is greater than all items in the last itemset. So, the size of a itemset-extended sequence does not increase.

Remark 2: In itemset extension, α is an incomplete prefix of all itemset-extended sequences. Moreover, α is an incomplete prefix of all sub nodes of the nodes which are itemset-extended of α.

Based on the above remarks, we develop the algorithm *MSR_PreTree* for generating rules. This algorithm is presented in Section 5.2.

For example, Fig. 1 shows the prefix-tree of frequent sequences. ⟨(A)(A)⟩ and ⟨(A)(B)⟩ are sequence-extended sequences of ⟨(A)⟩, and ⟨(AB)⟩ is an itemset-extended sequence of ⟨(A)⟩. Sequence ⟨(A)⟩ is a prefix of all sequences in T1 and it is an incomplete prefix of sequences in T2.

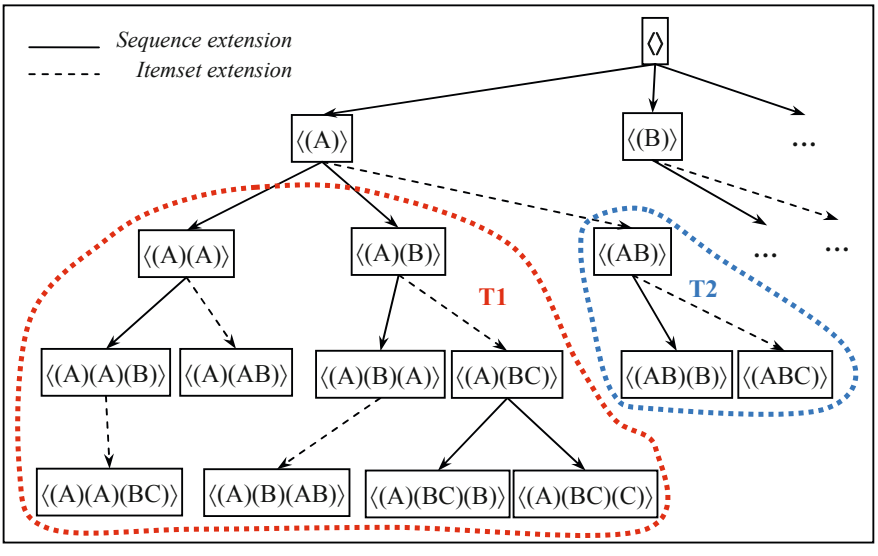

Fig. 1. The Prefix-Tree

5 Proposed Algorithms

The confidence of a rule depends on the support of the prefix. Firstly, for each frequent sequence *f*, *Full* has to generate all prefixes of *f*. Then, for each prefix, *Full* has to pass over *FS* to find that prefix's support. Let *n* be the number of sequences in *FS*, *k* be the size of the largest sequence in *FS*, the complexity of this algorithm is *O(n*k*n)* (not to mention the time of generate prefixes). In this section, we proposed two algorithms which effectively reduce the number of *FS* scans.

5.1 MSR_ImpFull

MSR_ImpFull algorithm is given in Fig. 2. It is an improved algorithm of *Full* algorithm. Firstly, the algorithm sorts all sequences in *FS* in ascending order according to their size. As a result, only sequences before X in the list are prefix of X. In detail, for

each sequence *f* in *FS*, the algorithm makes a pass over the set of sequences which consist of the sequences following *f* in ordering, we denote this set is *BS*. For each sequence *b* in *BS*, we check whether *b* contains *f* as a prefix or not. If *f* is a prefix of *b* then we try to form a rule $f \rightarrow post$ where *post* is a postfix of *b* with respect to prefix *f*, and compute the confident value of rule (lines 5-8). When the confidence satisfies minConf, we output that rule. If we just mind in the number of FS scans without the time of prefix checking then the complexity of *MSR_ImpFull* is $O(n*n)$.

```
MSR ImpFull algorithm:
Input: Sequence database, minSup, minConf
Output: All significant rules
Method:
   1. Let FS = All sequences with support ≥ minSup
   2. Sort all sequences in FS in ascending order by
      their size
   3. For each sequence f∈FS
   4.   Let BS = All sequences following f in order
   5.   For each sequence b in BS
   6.     If f is a prefix of b
   7.       Let post = postfix of b w.r.t prefix f
   8.       Let r = f→post,
                sup = sup(b), conf = sup(b)/sup(f)
   9.       If (rconf ≥ minConf)
   10.          Output the rule r(sup, conf)
```

Fig. 2. The *MSR_ImpFull* algorithm

5.2 MSR_PreTree

Although *MSR_ImpFull* algorithm has less numbers of *FS* scans than *Full*, but for each sequence *f*, it scans all sequences following *f* to identify which sequence contains *f* as a prefix. In order to overcome this, we use a prefix-tree structure as mentioned in Section 4. Based on prefix-tree, *MSR_PreTree* generates rules directly because any sequence α in the tree (besides the root of tree) is always a prefix of all sequences on the sub trees that their roots are sequence-extended nodes of α as Remark 1. We can see that the average number of sequences of those sets approximates *k*. Thus, the complexity of *MSR_PreTRee* is $O(n*k)$.

MSR_PreTree algorithm is shown in Fig. 3. Firstly, we find all frequent sequences using PRISM [6]. As a result, those frequent sequences are stored in the prefix-tree structure (line 1). For each node *r* at level 1, which is the root of a sub tree, we generate rules from each sub tree by calling procedure *Generate-Rule-From-Root(r)* (lines 2-4).

```
MSR PreTree Algorithm:
  Input: Sequence database, minSup, minConf
  Output: All significant rules
  Method:
   1. Let FS = All sequences with support ≥ minSup,
      stored in a prefix-tree (using PRISM [6])
   2. F1 = All nodes at level 1
   3. For each node r in F1
   4.   Generate-Rule-From-Root(r)
```

Fig. 3. The *MSR_PreTree* algorithm

```
Generate-Rule-From-Root(root):
 1. Let Seq-Set = Sequence extensions of root
 2. Let Items-Set = Itemset extensions of root
 3. For each node nseq in Seq-set
 4.   Let Sub-Tree = The tree which rooted at nseq
 5.   Let pre = sequence at root
 6.   Generate-Rules(pre, Sub-Tree)
 7. For each node nseq in Seq-set
 8.   Generate-Rule-From-Root(nseq)
 9. For each node nitem in Items-Set
10. Generate-Rule-From-Root(nitem)
```

Fig. 4. Generate all rules from a tree

Fig. 4. shows the pseudo-code for the procedure *Generate-Rule-From-Root*. For the root, we have two sets of nodes: a set of sequence-extended nodes and a set of itemset-extended nodes of the root (lines 1, 2). From Remarks 1 and 2, we just generate rules from the sequences on the sub trees that their roots are sequence-extended nodes of the root, because the sequence at the root (denoted as *pre*) is the prefix of these sequences. Hence, for each sub tree, we generate all rules from sequences on the sub tree with respect to prefix *pre*, using procedure *Generate-Rules* (lines 3-6). All

extended-nodes of current root will become the prefix for the sub trees at the next level, so we recursively call this procedure for every extended-node of the root (lines 7-10). This recursive process is repeated until the last level of the tree.

Finally, the detail of the procedure *Generate-Rules* is shown in Fig. 5. The input to the procedure is a tree and a sequence *pre* so that *pre* is a prefix of all sequences in that tree. The tree can be traversed in a DFS (Depth First Search) or BFS (Breadth First Search) manner. For each sequence *f* in the tree, we generate the rule *pre*→*post* which *post* is a postfix of *f* with respect to prefix *pre*.

```
Generate-Rules(pre, Sub-Tree):
   1. For each node n in Prefix-Tree
   2.   Let f = sequence at node n
   3.   Let post = postfix of f w.r.t prefix pre
   4.   Let r=pre→post,
        rsup = sup(f) and rconf = sup(f)/sup(pre)
   5.   if (rconf ≥ minConf)
   6.     Output the rule pre→post(rsup, rconf)
```

Fig. 5. Generate all rules from the sequences on the tree with given prefix

For example, in Fig. 1, in case of generating rules from the root node ⟨(A)⟩, we have results as in Table 1.

Table 1. The result of mining sequential rules from the root node ⟨(A)⟩

Prefix	Rules
⟨(A)⟩	⟨(A)⟩→⟨(A)⟩, ⟨(A)⟩→⟨(B)⟩, ⟨(A)⟩→⟨(A)(B)⟩, ⟨(A)⟩→⟨(AB)⟩, ⟨(A)⟩→ ⟨(B)(A)⟩, ⟨(A)⟩→⟨(BC)⟩, ⟨(A)⟩→⟨(A)(BC)⟩, ⟨(A)⟩→⟨(B)(AB)⟩, ⟨(A)⟩→⟨(BC)(B)⟩, ⟨(A)⟩→⟨(BC)(C)⟩
⟨(A)(A)⟩	⟨(A)(A)⟩→⟨(B)⟩, ⟨(A)(A)⟩→⟨(BC)⟩
⟨(A)(B)⟩	⟨(A)(B)⟩→⟨(A)⟩, ⟨(A)(B)⟩→⟨(AB)⟩
⟨(A)(BC)⟩	⟨(A)(BC)⟩→⟨(B)⟩, ⟨(A)(BC)⟩→⟨(C)⟩
⟨(AB)⟩	⟨(AB)⟩→⟨(B)⟩

In Fig. 1, consider root node ⟨(A)⟩, the itemset-extended sequence of ⟨(A)⟩ is ⟨(AB)⟩ and the sequence-extended sequences of ⟨(A)⟩ are ⟨(A)(A)⟩ and ⟨(A)(B)⟩. Because ⟨(A)⟩ is an incomplete prefix of ⟨(AB)⟩ and all sub nodes of ⟨(AB)⟩, so we don't generate rules from those nodes with prefix ⟨(A)⟩. On the contrary, ⟨(A)⟩ is a prefix of ⟨(A)(B)⟩ and ⟨(A)(C)⟩, so we have rules: ⟨(A)⟩→⟨(A)⟩, ⟨(A)⟩→⟨(B)⟩. Moreover, ⟨(A)⟩

is a prefix of all sub nodes of ⟨(A)(A)⟩ and ⟨(A)(B)⟩. We also generate rules from those sub nodes with prefix ⟨(A)⟩. We repeat the above process for all the sub nodes which are ⟨(A)(A)⟩, ⟨(A)(B)⟩ and ⟨(AB)⟩.

6 Experimental Results

We perform the experiments on two kinds of databases which are in potentially useful areas such as purchase histories, web logs, program execution traces. In the first kind of databases, the size of each itemset of a sequence is greater than or equal to 1. And in the second, all itemsets are of size 1. All experiments are performed on a PC with Intel Core 2 Duo CPU T8100 2x2.10GHz, and 2 GBs memory, running Windows XP Professional.

Synthetic and Real Database: For the first kind of databases, we use synthetic data generator provided by IBM which was used in [1, 6].The generator takes the parameters shown in Table 2. And for the second, we use tests on real database, Gazelle, in which each single item is considered as an itemset. Gazelle was obtained from Blue Martini company, which was also used in KDD-Cup 2000 [13]. This database contains 59602 sequences (i.e., customers), 149639 sessions, and 497 distinct page views. The average sequence length is 2.5 and the maximum sequence length is 267.

Table 2. Parameters for IBM Data Generator

C	Average itemsets per sequence
T	Average items per itemset
S	Average itemsets in maximal sequences
I	Average items in maximal sequences
N	Number of distinct items
D	Number of sequences

Table 3 shows the execution time of three algorithms in two synthetic databases C6T5S4I4N1kD1k and C6T5S4I4N1kD10k with the minSup is decreased from 0.8% to 0.4%. When the support threshold is high, there are only a limited number of frequent sequences, and the length of sequence is short. It is the reason why the number of rules is small, and three algorithms are close in terms of runtime. However, the gaps become clear when the support threshold decreases. Although three algorithms are generated the same rules, both *MSR_ImpFull* and *MSR_PreTree* are more efficient than *Full*. With *MSR_ImpFull* algorithm, runtime is improved up to 2 times and up to 3240 times with *MSR_PreTree* algorithm. So *MSR_PreTree* outperforms both *MSR_ImpFull* and *Full* algorithm. We also test three algorithms on real database (Gazelle) and obtain similar result. With all tests, we set minConf = 50%.

Table 3. The mining time comparison on different synthetic databases

Databases	minSup (%)	#FS	# Rules	Runtime (s)		
				Full	*MSR_ImpFull*	*MSR_PreTree*
C6T5S4I4N1kD1k	0.8	11,211	913	11.63	6.17	0.04
	0.7	14,802	1,441	25.86	13.88	0.05
	0.6	20,664	2,302	65.61	34.05	0.07
	0.5	31,311	4,045	157.90	84.33	0.09
	0.4	54,566	9,134	518.40	269.54	0.16
C6T5S4I4N1kD10k	0.8	8,430	248	6.23	4.25	0.02
	0.7	10,480	335	10.44	6.81	0.02
	0.6	13,628	441	19.81	11.33	0.03
	0.5	18,461	613	47.78	24.04	0.04
	0.4	27,168	925	115.77	51.62	0.06
Gazelle	0.11	4,781	4,721	5.59	2.21	0.13
	0.10	5,646	9,527	8.31	3.23	0.23
	0.09	6,883	16,224	14.03	4.88	0.21
	0.08	9,177	37,608	37.93	11.83	0.51
	0.07	15,197	48,672	272.61	49.71	0.77

7 Conclusion and Future Work

In this paper, we have proposed two algorithms named *MSR_ImpFull* and *MSR_PreTree* to mine sequential rules from sequence databases. The key idea is that if given a sequence, we can know which sequences contain a given sequence as a prefix based on sorting the set of frequent sequences; or we can immediately determine which sequences contain a given sequence as a prefix based on prefix-tree. *MSR_PreTree* avoids multiple passes over the full set of frequent sequences in prefix determining process, which can improve the efficiency.

Experimental results show that performance speed up can be obtained in both kinds of databases. Both *MSR_ImpFull* and *MSR_PreTree* are faster than *Full,* and *MSR_PreTree* is also faster than *MSR_ImpFull.* In future, we will apply this approach for generating rules in many kinds of interestingness measures. Besides, based on prefix-tree, we can also mine non-redundant sequential rules.

References

1. Agrawal, R., Srikant, R.: Mining Sequential Patterns. In: Proc. of 11th Int'l Conf. Data Engineering, pp. 3–14 (1995)
2. Srikant, R., Agrawal, R.: Mining Sequential Patterns: Generalizations and Performance Improvements. In: Proc. of 5th Int'l Conf. Extending Database Technology, pp. 3–17 (1996)
3. Zaki, M.J.: SPADE: An Efficient Algorithm for Mining Frequent Sequences. Machine Learning Journal 42(1/2), 31–60 (2000)
4. Pei, J., et al.: Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach. IEEE Trans. Knowledge and Data Engineering 16(10), 1424–1440 (2004)

5. Ayres, J., Gehrke, J.E., Yiu, T., Flannick, J.: Sequential Pattern Mining using a Bitmap Representaion. In: SIGKDD Conf., pp. 1–7 (2002)
6. Gouda, K., Hassaan, M., Zaki, M.J.: Prism: A Primal-Encoding Approach for Frequent Sequence Mining. Journal of Computer and System Sciences 76(1), 88–102 (2010)
7. Spiliopoulou, M.: Managing interesting rules in sequence mining. In: Żytkow, J.M., Rauch, J. (eds.) PKDD 1999. LNCS (LNAI), vol. 1704, pp. 554–560. Springer, Heidelberg (1999)
8. Lo, D., Khoo, S.-C., Liu, C.: Efficient Mining of Recurrent Rules from a Sequence Database. In: Haritsa, J.R., Kotagiri, R., Pudi, V. (eds.) DASFAA 2008. LNCS, vol. 4947, pp. 67–83. Springer, Heidelberg (2008)
9. Lo, D., Khoo, S.C., Wong, L.: Non-Redundant Sequential Rules-Theory and Algorithm. Information Systems 34(4-5), 438–453 (2009)
10. Yan, X., Han, J., Afshar, R.: CloSpan: Mining Closed Sequential Patterns in Large Databases. In: SDM 2003, San Francisco, CA, pp. 166–177 (2003)
11. Brin, S., Motwani, R., Ullman, J., Tsur, S.: Dynamic Itemset Counting and Implication Rules for Market Basket Data. In: Proc. of the 1997 ACM-SIGMOD Int'l Conf. on the Management of Data, pp. 255–264 (1997)
12. Berry, M.J., Linoff, G.S.: Data Mining Techniques for Marketing, Sales and Customer Support. John Wiley & Sons, Chichester (1997)
13. Kohavi, R., Brodley, C., Frasca, B., Mason, L., Zheng, Z.: KDD-Cup 2000 Organizers' Report: Peeling the Onion. SIGKDD Explorations 2(2), 86–98 (2000)
14. Baralis, E., Chiusano, S., Dutto, R.: Applying Sequential Rules to Protein Localization Prediction. Computer and Mathematics with Applications 55(5), 867–878 (2008)