

# Discovering Target Events Rules based on Time-Consecutive Pattern Mining\*

Ehud Gudes

ehud@cs.bgu.ac.il

Ben-Gurion University of the Negev  
Beer-Sheva, 84105, Israel

Litvak Marina

litvakm@cs.bgu.ac.il

Ben-Gurion University of the Negev  
Beer-Sheva, 84105, Israel

## Abstract

We are given temporal customer-oriented dataset, where each transaction consists of set of events that are associated with a customer id and a timestamp. For each customer there are several transactions with different timestamps. One of the events is defined as the target event. We introduce the problem of mining target events rules that are based on the discovery of continuous sequential patterns over such databases. We propose two algorithms to solve this problem and evaluate their performance using real-life data. The presented algorithms, CTSPD and CSPADE, have comparable evaluations, but the CTSPD, as expected, turns out to work faster.

## 1 Introduction

**Motivation** Many of the the modern databases are temporal and customer-oriented, like the famous basket database, that is, a dataset of purchases where each purchase is associated with an owner and a time [3, 8, 2]. Such databases collect and store the transactions in the order of receiving the data. A huge amount of data is collected every day in the form of event time sequences. Common examples are recordings of different values of stock shares during a day, accesses to a computer via an external network, bank transactions, or events related to malfunctions in an industrial plant. These sequences register events with corresponding values of certain processes, and are valuable sources of information not only to search for a particular value or event at a specific time, but also to analyze the frequency of certain events, or sets of events related by particular temporal relationships. This type of analysis can be very useful for predicting the future behavior of the monitored process.

Many commercial problems for data mining include prediction of the behavior of the customers. The miner can define the target predicted event, like the customer will/will

not buy something or will/will not change his/her status. The companies are usually interested in the sequence of actions or events that leads to such particular action (let's note them *basic* events and *target* event, respectively) [9]. For that purpose such companies may collect and store the information about customers in the form of transactions including customer id, time, set of basic events and the target event associated with that id and time. We introduce the problem of target events rules using continuous sequential pattern mining over such dataset.

A sequential pattern is defined as  $A_1 \rightarrow A_2 \rightarrow \dots \rightarrow A_n$  where  $A_i$ ,  $1 \leq i \leq n$  are events (or set of events) and  $\text{time}(A_i) \leq \text{time}(A_j)$  for  $i \leq j$  [6, 15, 5, 1].

In the previous algorithms for mining sequential patterns the resulting sequence need not be continuous (the sequence  $AB \rightarrow C$  is continuous if there is no event  $D$  that happens after  $AB$  and before  $C$ ). But such patterns do not contain information about how exactly the pattern's events affect its other events. For instance, the pattern  $A \rightarrow B$  may express “ $B$  happens year after  $A$ , and some other events happened during that period” and, at the same time, it can mean something different, like “ $B$  happened immediately after  $A$ ”. We are interested only in precise, continuous sequences that imply a target event. An example of such a rule may be: “After buying a ticket to the football match which was immediately after purchasing a merchandise, the customer buys a sport lottery”. We will give a precise definition of these rules later in this paper.

The application that we are interested in and have real data on is that of up-sale/cross-sale of telecommunication companies that are interested in selling additional or up-graded products to their customers [9]. Companies need to identify target-customers for product upgrade sales. One important potential source of information is usage data, e.g. how much bandwidth was used by each user at every hour during the last weeks or months. Customer usage data differs in that it is a time series. Although there is a considerable amount of work on time series, most of it deals with the prediction of a future value in the time series, as in stock market prediction [11]. In our case, we use the time se-

\*partially supported by the Lynn and Frankel center for computer science and by the Paul Ivanir Robotics center.

ries in order to predict a phenomenon external to the time series. Our target concept is a boolean feature that indicates whether the customer will buy an upgraded product in the near future. To reduce the problem to the attribute-based, we extract the subset of important threshold features of the form: the fraction of the time in the period  $X$  that a certain measure was above  $Y$  (for example, such feature may be the fraction of the time that the bandwidth utilization was above 90% in the last 5 weeks.) Since all the features were continuous, we used abstraction algorithms to map them to discrete values. The reduction of the problem into attribute based allows us to use conventional data mining algorithms (with our enhancements). In [9] were used rules received from such algorithms as Association Rules (including Quantitative and Clustered Association Rules) and BKB.

**Related work** The problem of sequential patterns was introduced in [5] by R. Agrawal and R. Srikant. They presented three algorithms for solving the problem of discovering sequential patterns over large databases of customer transactions. The proposed algorithms generate a data sequence for each customer from the database and search on this set of sequences for a frequent sequential pattern. For example, the algorithms can discover that customers typically rent “Star Wars”, then “Empire Strikes Back”, and then “Return of the Jedi”. Two of these algorithms, GSP and AprioriSome, were designed to find only maximal sequential patterns and the third algorithm, AprioriAll, finds all patterns. Briefly, AprioriAll is a three-phase algorithm. It first finds all itemsets with minimum support (frequent itemsets), transforms the database so that each transaction is replaced by the set of all frequent itemsets contained in the transaction, and then finds sequential patterns. All algorithms are based on the classic Apriori principle, introduced with the Association Rules Discovery problem [3], [4].

In a later paper, the same authors, [13], generalize the problem by introducing such terms as time constraints that specify a minimum and/or maximum time period between adjacent elements in a pattern, and a user-defined taxonomy (is-a hierarchy) on items allowing sequential patterns to include items across all levels of the taxonomy. A sequence in [13] consists of a list of sets of items, rather than being simply a list of items. In addition, authors of [13] are interested in finding all sequences with minimum support rather than only maximal frequent patterns.

H. Mannila et. al. [12] search event sequences for frequent patterns of events. These patterns have a simple structure (essentially a partial order) whose total span of time is constrained by a window given by the user. The technique of generating candidate patterns from sub-patterns, together with a sliding window method, results with effective algorithms. Similarly to [5], the strategy of [12] is starting with

simple sub-patterns (subsequences in this case) and incrementally building longer sequence candidates for the discovery process.

The work by Wang et. al. in [14] also deals with the discovery of sequential patterns, where the considered patterns are in the form of specific regular expressions with a distance metrics as a dissimilarity measure in comparing two sequences. The proposed approach is mainly tailored to the discovery of patterns in protein databases.

M. Zaki, [15], analyzes sequential dependencies among different events and introduces an algorithm that is called SPADE (Sequential Pattern Discovery using Equivalence classes). For efficient and fast search for all frequent patterns, SPADE utilizes combinatorial properties to decompose the original problem into smaller sub-problems, that can be independently solved in main-memory using efficient lattice search techniques, and using simple join operations. The ‘search space’ is decomposed into prefix-based parent equivalence classes and all frequent sequences are enumerated via BFS or DFS search within each class. All sequences are discovered in only three database scans.

In order to solve the event prediction problem, Gary M. Weiss in [10] developed Timeweaver, a genetic-based machine learning system that, given a pre-specified “target” event, learns to identify patterns in the data that successfully predict the future occurrence of that event.

In our work we use some of the above methods but adapt them to the special application domain where we are interested only with rules whose right side is a target event and left side is a consecutive sequential pattern which ends with the target event.

**Organization of the paper** We give a formal description of the problem of mining target events rules in Section 2. In Section 3, we describe CTSPD, an apriori-based algorithm for finding such rules from continuous patterns, and present an illustrative example. In Section 4 we describe the second algorithm (CSPADE) which is based on SPADE [15], and is modified according to our problem definition. In Section 5 we empirically compare the performance of CTSPD with the CSPADE, and study their scale-up properties. We conclude with a summary in Section 6. The appendix contains a more detailed description of SPADE [15].

## 2 Definitions and Problem Statement

We are given a dataset  $D$  of customer transactions. Each transaction consists of customer id, timestamps (start date and end date), several basic events and one target event. More formally, such transactions look like  $\langle cust\_id, s\_date, e\_date, ev_1, ev_2, \dots, ev_n, target \rangle$ , where  $ev_i$  is the basic event occurring at  $[s\_date, e\_date]$  and the  $target$  is a boolean feature that indicates whether

the customer did or didn't do some action in the end of this period (for example, did he buy an upgraded product or didn't). Formally, each basic event is associated with the period between  $s\_date$  and  $e\_date$ , and each target event is associated with  $e\_date$ . No customer has more than one transaction with the same timestamps.

An event  $ev_i$  is denoted by a pair  $\langle event\_type_i, event\_property_i \rangle$ , where  $event\_type_i$  is an attribute and  $event\_property_i$  is a value of that attribute in the record.

An *itemset* is a non-empty set of events occurred at the same period. Denote itemset  $l$  by  $(ev_1, ev_2, \dots, ev_n)$ , where  $ev_i$  is an event.

A *sequence* is an ordered list of itemsets and denoted by  $l_1 \rightarrow target_1 \rightarrow l_2 \rightarrow target_2 \rightarrow \dots \rightarrow l_n \rightarrow target_n$ , where  $l_i$  is an itemset of basic events and  $target_i$  is an event with the boolean property yes/no. We call  $l_i$  the  $i^{th}$  *basic level* (level of basic events) and  $target_i$  is the  $i^{th}$  *target level*. Note that  $l_1$  and/or  $target_n$  need not exist — the sequence can start from the target level and/or end on the basic level. The sequence has to be continuous. The sign  $\rightarrow$  means “happened immediately after” (or, more precisely, “without other events occurring in between”).

The sequence  $s$  is denoted by  $l_1 \rightarrow target_1 \rightarrow l_2 \rightarrow target_2 \rightarrow \dots \rightarrow l_n \rightarrow target_n$  is *continuous* if for each  $i$  there is no event  $ev : ev \notin l_i, ev \notin l_{i+1}$  and  $ev \neq target_i$  that happens after period of  $l_i$  and before the period of the  $l_{i+1}$ .

All customer transactions of the same id sorted by the timestamp can be viewed as a single sequence. We will call it a *customer-sequence*. Note, that a customer-sequence will always have the first basic level and the last target. A transaction database  $D$  converted to a collection of such customer-sequences is called *sequence database* and is denoted by  $T_D$ .

The *length* of a sequence is the number of levels in the sequence. Denote the sequence received from the sequence  $s$  by removing the last level by *R-prefix* and the sequence received after removing the first level by *R-suffix* of  $s$  (restricted prefix and suffix accordingly). The *size* of a sequence is the number of events in the sequence. Denote the sequence of size  $k$  by *k-sequence*. The *size* of an itemset is the number of events in the itemset. Denote the itemset of size  $k$  by *k-itemset*. For instance, the sequence  $AB \rightarrow U \rightarrow C$  has length equal to 3 and size equal to 4.  $AB$  is a 2-itemset and  $U$  and  $C$  are 1-itemsets (actually, the target levels are always 1-itemsets). And for sequence  $s$   $A \rightarrow U \rightarrow C$  both length and size are equal to 3.  $A \rightarrow U$  is the R-prefix and  $U \rightarrow C$  is the R-suffix of  $s$ .

Given any continuous and frequent  $k$ -sequence  $s$  denoted by  $l_1 \rightarrow ltarget_1 \rightarrow l_2 \rightarrow ltarget_2 \rightarrow \dots \rightarrow l_n \rightarrow ltarget_n$  and there is  $i$  (at least one) such that  $l_i$  is the  $n$ -itemset with  $n > 1$ . We call such sequence *narrowable* sequence. The

sequence formed by substitution of  $l_i$  for  $l'_i$ , where  $l'_i \subset l_i$  and  $|l'_i| = n - 1$  is called a *narrowed-sequence* of  $s$ .

**Lemma 1:**

- a) All narrowed-sequences of a frequent continuous sequence (if exist) must be frequent and continuous too (reverse is not true).
- b) R-suffix and R-prefix of the frequent continuous sequence must be continuous and frequent too.

The sequence  $s' l_1 \rightarrow ltarget_1 \rightarrow l_2 \rightarrow ltarget_2 \rightarrow \dots \rightarrow l_n \rightarrow ltarget_n$  is *contained* in another sequence  $s k_1 \rightarrow ktarget_1 \rightarrow k_2 \rightarrow ktarget_2 \rightarrow \dots \rightarrow k_m \rightarrow ktarget_m$  (denote  $s' \subset s$ ) if and only if  $\exists j : \forall i, 1 \leq i \leq n$   $l_i \subseteq k_{j+i}$  and  $ltarget_i = ktarget_{j+i}$ .

**Lemma 2:** If  $s' \subset s$  and  $size(s') = size(s) - 1$  then  $s'$  is either narrowed-sequence or R-prefix or R-suffix of  $s$ .

Also, some sequence  $s'$  may be contained more than once in another sequence  $s$ . For example,  $AB \rightarrow U \rightarrow C$  is contained in the  $ABC \rightarrow U \rightarrow CD \rightarrow \bar{U}$ . However, the sequences  $AB \rightarrow \bar{U} \rightarrow C$  or  $AD \rightarrow U \rightarrow C$  are not. The sequence  $C \rightarrow U$  is contained in the sequence  $AC \rightarrow U \rightarrow BC \rightarrow U \rightarrow C$  twice.

**Definition 1:**  $\chi_{s,s'}$  is the number of occurrences of sequence  $s'$  in a sequence  $s$ . A customer-sequence  $s$  *matches* the other sequence  $s'$  if  $s' \subset s$ .

**Definition 2:** The *support* of sequence  $s'$ , denoted by  $sup(s')$ , is a fraction  $\frac{\chi_{T_D,s'}}{|D|}$ , where  $\chi_{T_D,s'} = \sum_{s \in T_D} \chi_{s,s'}$  and  $|D|$  is the number of transactions in the transaction database. A sequence with minimum support is called a *frequent sequence*.

**Definition 3:** *Target rule* is a sequence tailed with any target event.

**Definition 4:** The *confidence* for target rule  $r$ , denoted by  $conf(r)$  is defined as the fraction:  $\frac{\chi_{T_D,r}}{\chi_{T_D,\hat{r}}}$ , where  $\hat{r}$  is the R-prefix of  $r$ . A rule with minimum support and minimum confidence is called a *frequent rule*.

Given a transaction dataset  $D$  as defined above, the problem of mining target events rules using continuous sequential mining is to find all frequent target rules.

### 3 The CTSPD algorithm

Our first algorithm CTSPD (Continuous Target Sequential Patterns Discovery) is based on Apriori and has two special properties:

- i. The structure of sequences and
- ii. The consecutiveness property.

These properties affect both candidate generation and pruning as is shown below. The CTSPD consists of several basic

phases, that transform the transaction database to the sequence database, mine all frequent sequences and then select from them appropriate rules. In this section we give a description for each phase and an illustrative example.

**1. Sorting the database** First of all, we sort the seed dataset by a customer id and start/end time as primary and secondary keys accordingly. The purpose of this action is straightforward transformation phase — transforming the dataset into a set of customer-sequences. An example of the dataset after such sorting can be seen in Figure 1.

**2. Frequent 1-sequences generation** To find the support for events we scan the dataset  $D$ , and count the number of occurrences for each event. At the end of the scan we know all frequent events. We store the frequent events in two separate structures: hash table for basic events (for quick access) and a 2-element array for the target events. We calculate the number of basic frequent events  $N$  and map them to a set of contiguous integers. These integers are used as the indexes for alternate bit representation of the sequences, described below. To represent the itemset we use a bit string of length  $N$ , where the bit  $i$  is set to 1 if the event with index  $i$  is present in this itemset. Example for both mappings are depicted in Figure 2.

**3. Transformation phase** The goal of this phase is the transformation of the given database to the collection of user-sequences (one sequence for a user, that includes all his frequent events in order of occurring). We don't include the non-frequent events to the customer-sequences, based on the Apriori principle, confirming that frequent itemsets do not include non-frequent elements. If all transactions for a customer do not contain frequent basic events at all, we do not create the respective customer-sequence. To store the temporal relations on events we need a special structure for transactions. We store each sequence in form of ordered list of  $plevels$ , where each  $plevel_i$  is the pair  $l_i \rightarrow target_i$ , where  $l_i$  is the  $i^{th}$  basic level and  $target_i$  is the  $i^{th}$  target level. We present the itemset of basic level by a bit-string as was described above. For example, the itemset  $ACD$  will look as 101100 given the mapping from the Figure 2. We use single bit for target level, that is set to 1 if the target = *true* and to 0 otherwise.

The set of customer-sequences is denoted by  $T_D$ .

**4. Frequent 2-sequences creation** Denote the term of basic event by  $bev$  and the target event by  $tev$ . We create 2 types of pairs:  $bev \rightarrow tev$  and  $tev \rightarrow bev$  from the events generated in the second phase. We need not create the pairs where both right and left parts are basic or target (like  $bev \rightarrow bev$  or  $tev \rightarrow tev$ ) because they cannot be continuous and

---

**Algorithm 1** Frequent sequence generation

---

```

Result  $\leftarrow \theta$ ;
 $L_k = L_2$ ;
while( $L_k \neq \theta$ ) {
    Result = Result  $\cup$   $L_k$ ;
     $C_k = \text{GenerateCandidates}(L_k, L_2)$ ;
    SupportCompute( $C_k$ );
     $L_{k++} = C_k$ ;
}

```

---

will not participate in the candidate generation phase. The pair of type  $bev \rightarrow tev$  is represented by one level and for the pair  $tev \rightarrow bev$  we create two levels, where the first one has an empty basic level, and the target of the second level is empty.

**5. The frequent sequences generation phase** In order to find all the frequent continuous sequences we perform multiple passes over the transformed database. In each pass  $k$  we start with the initial set of frequent  $k + 1$ -sequences and create the next potentially frequent  $k + 2$ -sequences, called *candidates*. We find the support for these candidates during the scan of  $T_D$ , and at the end of the scan we know which of them are frequent. The set of these new frequent sequences becomes the initial set for the next  $k + 1$  pass. We denote the initial set by  $L_k$  and the new candidates by  $C_k$ . This is shown in Algorithm 1. The procedure *GenerateCandidates* uses two different joins in candidates generation and pruning as described below.

**6. Rules generation** Rules generation is the last phase of the algorithm where we filter the sequences found in phase 5. The first filter passes sequences ended by the target level. Then, for each one we calculate the confidence, according to the formula. We remove all sequences having confidence less than *min\_conf*.

---

**Algorithm 2** GenerateCandidates( $L_k, L_2$ )

---

```

 $C_k \leftarrow \emptyset$ ;
for ( $i = 0$ ;  $i < L_k.size$ ;  $i++$ )
    for ( $j = i + 1$ ;  $j < L_k.size$ ;  $j++$ )
        ExpJoin ( $L_k[i]$ ,  $L_k[j]$ );
for ( $k = 0$ ;  $k < L_2.size$ ;  $k++$ )
    ConcatJoin ( $L_k[i]$ ,  $L_2[k]$ );

```

---

**3.1 Candidate Generation**

To generate the next potentially frequent continuous sequences we use two different joins (see Algorithm 2). The

first one joins the frequent  $k$ -sequences from the previous pass, and is called Expanding Join (ExpJoin). This procedure is responsible for *expansion* of the sequences. By *expansion* we denote the process from which two seed  $k$ -sequences of the same length, produces a new  $k + 1$ -sequence (of the same length). It may happen by extending one of the basic levels. This procedure is given two sequences,  $s_1$  and  $s_2$ , checks if their lengths are equal, and if there is a pair of plevels which is appropriate for the Expanding Join given all other plevels are identical. This expansion is similar to itemset expansion in Apriori (see Algorithm 3 and also 4). For example, two sequences  $AB \rightarrow U \rightarrow C$  and  $AD \rightarrow U \rightarrow C$  form the candidate  $ABD \rightarrow U \rightarrow C$ . This technique doesn't protect us from the repeating generations of the same candidates, and to avoid this before inserting the candidate to  $C_k$  we need to check if it's already contained.

The second join, called Concatenating Join (ConcatJoin), concatenates a  $k$ -sequences from  $L_k$  with 2-sequences from  $L_2$ . As a result from this procedure, we receive the new  $k+1$ -sequence of length  $m+1$ , where  $m$  is the length of the seed  $k$ -sequence. The procedure receives two sequences,  $s_1$  and  $s_2$ , and compares  $s_1$ 's last level to the first one of  $s_2$  and  $s_1$ 's first level to the last one of  $s_2$ . As a result, we can generate a maximum of two new candidates from a single pair. This is shown in Algorithm 5. For example, the sequences  $A \rightarrow U \rightarrow B \rightarrow U$  and  $U \rightarrow A$  are appropriate for double concatenation. The two new candidates are:  $A \rightarrow U \rightarrow B \rightarrow U \rightarrow A$  and  $U \rightarrow A \rightarrow U \rightarrow B \rightarrow U$ .

### 3.2 Pruning

In the Pruning stage we remove all sequences that are not continuous and/or not frequent. As we mentioned in the Lemmas of Section 2, all  $k - 1$ -subsequences of a frequent and continuous sequence must be frequent and continuous. To save time by not counting the support for the non-frequent and/or non-continuous sequences, we prune the candidates which do not support this check. For narrowable sequence we generate narrowed subsequences. For non-narrowable sequences we generate two subsequences: R-prefix and R-suffix. The procedure is described in Algorithm 6.

### 3.3 Support counting

In order to check frequency of candidates, we scan the sequence database  $T_D$  and count the number of sequence occurrences in the user-sequences (procedure SupportCount). The procedure used for support calculating and for checking the candidate continuity, is procedure Match described below. It receives two parameters: user-sequence and candidate, and returns number of occurrences of this

---

#### Algorithm 3 ExpJoin( $s_1, s_2$ )

---

```

lev ← -1;
if (s1.length == s2.length){
  for (i = 0; i < s1.length; i++){
    if (s1.pleveli ≠ s2.pleveli){
      if ExpJoinAppr(s1.pleveli, s2.pleveli){
        if (lev == -1)
          //first and uniquely matching
          lev = i;
        else return;
      }
      //there is more than one matching
    }
    else return;
  }
  //the plevels are not equal
  and not appropriate
}
Candidate = ExpandLevel(s1, s2,
plev);
//plev denotes where the expansion
occurs
if (!Prune(Candidate))
  Ck = Ck ∪ Candidate;

```

---



---

#### Algorithm 4 ExpJoinAppr(plev1, plev2)

---

```

if(first k-1 bits at the
basic levels are the same
&& kth bits are different
&& target1 == target2)
  return true;
else
  return false;

```

---



---

#### Algorithm 5 ConcatJoin( $s_1, s_2$ )

---

```

if(s1.lastLevel == s2.firstLevel){
  Candidate = concat(s1, s2);
  if (!Prune(Candidate))
    Ck = Ck ∪ Candidate;
}
if(s1.firstLevel = s2.lastLevel){
  Candidate = concat(s2, s1);
  if (!Prune(Candidate))
    Ck = Ck ∪ Candidate;
}

```

---

**Algorithm 6** Prune(s)

---

```

for all basic levels  $bl \in s$ ,
 $|bl| = m > 1$  {
  for all  $ist \subset bl$ ,  $|ist| = m - 1$  {
    seq = s with  $ist$  in-
stead of  $bl$ ;
    if seq  $\notin L_{k-1}$ 
      return true;
  }
}
if  $\nexists bl \in s, |bl| = m > 1$ 
  prefix = first  $k - 1$  levels;
  if prefix  $\notin L_{k-1}$ 
    return true;
  suffix = last  $k - 1$  levels;
  if suffix  $\notin L_{k-1}$ 
    return true;
return false;

```

---

candidate in the sequence. If the strings representing the itemsets are short enough we can use the XOR function to check containment fast. See Algorithm 8.

**Algorithm 7** SupportCount( $C_k$ )

---

```

for all sequences  $us \in T_D$ 
  for all candidates  $cand \in C_k$ 
     $cand.supp += Match(us, cand)$ ;
  if  $us$  the last sequence
    and  $cand.supp < min\_sup$ 
      remove  $cand$  from  $C_k$ ;

```

---

### 3.4 Example

Consider the dataset  $D$  from Figure 1. The transactions are sorted by  $userId$  and  $startDate$ . Given support of 30%, 6 basic events and all the target events are frequent. We enumerate the frequent basic events and map them to contiguous integers as illustrated in Figure 2. The same Figure presents an alternate representation of the target events. Figure 3 contains set of customer-sequences formed from  $D$ , which contain only the frequent events. Eight frequent pairs are presented in Figure 4 with their alternate representation. Note, that there are no frequent patterns of type  $U \rightarrow ev$ . Figure 5 illustrates all candidates received during the discovery process. The frequent patterns are in bold. During 4-sequences generation we prune such sequences as  $F \rightarrow \bar{U} \rightarrow AC$  (generated from  $F \rightarrow \bar{U}$  and  $\bar{U} \rightarrow AC$  by ConcatJoin),  $F \rightarrow \bar{U} \rightarrow A \rightarrow U$  (ConcatJoin on  $F \rightarrow \bar{U}$  and  $\bar{U} \rightarrow A \rightarrow U$ ),  $F \rightarrow \bar{U} \rightarrow C \rightarrow U$  (concat  $F \rightarrow \bar{U}$  with  $\bar{U} \rightarrow C \rightarrow U$ ),  $B \rightarrow \bar{U} \rightarrow E \rightarrow U$  ( $B \rightarrow \bar{U}$  and

**Algorithm 8** Match(us, cand)

---

```

count  $\leftarrow 0$ ;
for ( $i = 0; i < us.plevels.size -$ 
cand.plevels.size;
 $i++$ )
  for ( $j = 0; j <$ 
cand.plevels.size;  $j++$ ) {
     $cl = cand.level_j$ ;
     $sl = us.level_{i+j}$ ;
    if ( $(cl.basicLevel \subseteq sl.basicLevel$ 
or  $cl.basicLevel == \theta$ )
and  $cl.target == sl.target$ )
      count++;
    else break;
  }
return count;

```

---

**Frequent Basic Events**

event	den. by	sup	int	bit-string
(trendDay, increase)	$A$	50%	0	100000
(trendDay, flat)	$B$	40%	1	010000
(trendNight, increase)	$C$	50%	2	001000
(trendNight, flat)	$D$	30%	3	000100
(trendDayNight, flat)	$E$	40%	4	000010
(trendDayNight, decrease)	$F$	40%	5	000001

**Target events**

event	denoted by	support	bit
(DidUpgrade, Yes)	$U$	40%	1
(DidUpgrade, No)	$\bar{U}$	60%	0

Figure 2. Enumeration and mapping the frequent events

$\bar{U} \rightarrow E \rightarrow U$  accordingly) and  $F \rightarrow \bar{U} \rightarrow E \rightarrow U$ . The first two patterns are pruned since  $F \rightarrow \bar{U} \rightarrow A \notin L_3$ , the third is pruned because  $F \rightarrow \bar{U} \rightarrow C$  is not frequent, the fourth from the reason that  $B \rightarrow \bar{U} \rightarrow E \notin L_3$  and for the last  $F \rightarrow \bar{U} \rightarrow E \notin L_3$ . Some patterns were repeatedly generated, like  $\bar{U} \rightarrow A \rightarrow U$  and  $B \rightarrow \bar{U} \rightarrow A$  during the first step, and  $B \rightarrow \bar{U} \rightarrow AC$  and  $B \rightarrow \bar{U} \rightarrow A \rightarrow U$  during the second iteration of algorithm 5. In Figure 6 all target rules, given  $min\_conf$  70%, are presented. We also have four rules :  $A \rightarrow U$ ,  $C \rightarrow U$ ,  $\bar{U} \rightarrow A \rightarrow U$  and  $\bar{U} \rightarrow C \rightarrow U$  with confidence less than  $min\_conf$  (60%).

## 4 The CSPADE algorithm

In this section we describe the basic phases of slightly modified SPADE algorithm that we call CSPADE (Continuous Sequential Patterns Discovery using Equivalent

userId	startDate	endDate	trendDay	trendNight	trendDayNight	DidUpgrade
1	20/05/01	12/06/01	flat	flat	increase	No
1	13/06/01	09/09/01	increase	increase	increase	No
1	12/09/01	05/10/01	increase	increase	flat	Yes
2	25/01/01	03/02/01	decrease	decrease	decrease	No
2	10/02/01	30/01/01	increase	flat	flat	Yes
2	05/02/01	12/02/01	flat	decrease	decrease	No
2	13/02/01	20/02/01	increase	increase	decrease	Yes
3	30/01/01	15/02/01	flat	flat	flat	No
3	17/02/01	27/02/01	increase	increase	decrease	No
3	01/03/01	12/03/01	flat	increase	flat	Yes

Figure 1. Dataset D after sorting by userId and startDate

userId	sequence	alternate representation
1	$BD \rightarrow \bar{U} \rightarrow AC \rightarrow \bar{U} \rightarrow ACE \rightarrow U$	010100;0 → 101000;0 → 101010;1
2	$F \rightarrow \bar{U} \rightarrow ADE \rightarrow U \rightarrow BF \rightarrow \bar{U} \rightarrow ACF \rightarrow U$	000001;0 → 100110;1 → 010001;0 → 101001;1
3	$BDE \rightarrow \bar{U} \rightarrow ACF \rightarrow \bar{U} \rightarrow BCE \rightarrow U$	010110;0 → 101000;0 → 011010;1

Figure 3. Transformed dataset - set of customer sequences

pattern	support	alternate representation
$A \rightarrow U$	30%	100000;1
$C \rightarrow U$	30%	001000;1
$E \rightarrow U$	30%	000010;1
$B \rightarrow \bar{U}$	30%	010000;0
$F \rightarrow \bar{U}$	30%	000001;0
$\bar{U} \rightarrow A$	50%	NULL;0 → 100000;NULL
$\bar{U} \rightarrow C$	50%	NULL;0 → 001000;NULL
$\bar{U} \rightarrow E$	30%	NULL;0 → 000010;NULL

Figure 4. Frequent pairs

Classes). We made three main modifications to this algorithm:

- i. Using our definition of support,
- ii. We added an additional criterion for pruning during the processing of a class,
- iii. Filtering the non-continuous patterns after discovering all frequent sequences.

Recall that SPADE ([15]) works independently on equivalent classes which are specified by a common prefix. The main disadvantage of this algorithm is that partitioning into Equivalent Classes does not allow us to use our technique for generating only continuous patterns inside each class (the ConcatJoin and, in some cases, the ExpJoin do not work, because the seed sequences belong to different classes). Also, we cannot prune all non-appropriate patterns

candidate	support
$AC \rightarrow U$	20%
$AE \rightarrow U$	20%
$CE \rightarrow U$	20%
$BF \rightarrow \bar{U}$	10%
$\bar{U} \rightarrow AC$	<b>40%</b>
$\bar{U} \rightarrow AE$	20%
$\bar{U} \rightarrow CE$	20%
$\bar{U} \rightarrow A \rightarrow U$	<b>30%</b>
$\bar{U} \rightarrow C \rightarrow U$	<b>30%</b>
$\bar{U} \rightarrow E \rightarrow U$	<b>30%</b>
$B \rightarrow \bar{U} \rightarrow A$	<b>30%</b>
$B \rightarrow \bar{U} \rightarrow C$	<b>30%</b>
$B \rightarrow \bar{U} \rightarrow E$	0%
$F \rightarrow \bar{U} \rightarrow A$	20%
$F \rightarrow \bar{U} \rightarrow C$	20%
$F \rightarrow \bar{U} \rightarrow E$	20%

3-sequences

candidate	support
$\bar{U} \rightarrow AC \rightarrow U$	20%
$\bar{U} \rightarrow AE \rightarrow U$	20%
$\bar{U} \rightarrow CE \rightarrow U$	20%
$B \rightarrow \bar{U} \rightarrow AC$	<b>30%</b>
$B \rightarrow \bar{U} \rightarrow A \rightarrow U$	10%
$B \rightarrow \bar{U} \rightarrow C \rightarrow U$	10%

4-sequences

Figure 5. Candidates generation

rule	support	confidence
$E \rightarrow \overline{U}$	30%	75%
$B \rightarrow \overline{U}$	30%	75%
$F \rightarrow \overline{U}$	30%	75%
$\overline{U} \rightarrow E \rightarrow U$	30%	100%

Figure 6. Target Rules

during the class processing because they can produce appropriate candidates in the future (non-continuous patterns can form continuous). However, we can use the independence of classes and the fact, that each of them give patterns with the same prefix to limit the number of generated candidates. Recall that sequential patterns in our domain cannot have a “non-continuous” (or violated) structure, that is  $l_1 \rightarrow l_2 \rightarrow \dots \rightarrow l_k$ , where  $\exists i : l_i$  and  $l_{i+1}$  are both either target or basic levels, or  $l_i$  contains target event as well as basic ones. We avoid creation of classes specified by prefix that has such structure. This affects the way of creating new candidates, namely temporal join. We describe this later.

Note that the structure based method of pruning still does not guarantee that we’ll get only frequent continuous patterns. So, after processing of all classes we filter all patterns which are inappropriate to be target rules. We describe all the new phases below. Since the first, third and last phases, called Sorting the database, Transformaton phase and Rules generation respectively, are equivalent to the respective phases in Section 3, we omit these phases in the description. The phases describing partition into the Equivalent Classes and their processing do not differ from the respective phases of the original algorithm [15]. We briefly summarize them in the Appendix. The other phases are:

**2. The computation of the frequent 1-sequences** We use a vertical database format (see figure 8), where we maintain an id-list for each item. Each entry of the id-list is a  $\langle cid, tid \rangle$  pair where the item occurs (cid is the customer id and tid is the time id). Note, that for the basic event we associate the *start\_time* and for the target the *end\_time*. Using our definition of support, we do not calculate the fraction of customers supporting the pattern, but the number of occurrences of this pattern in the database. Given the vertical id-list database, all frequent 1-sequences can be computed in a single database scan. For each database item, we scan its id-list, incrementing the support for each new entry encountered.

**4. The computation of the frequent pairs** We use horizontal format of the database as described in the Appendix and [15]. We create all possible pairs, except for the Event Atoms (see Appendix) where at least one of the events is target. The reason is that such patterns will give non-

appropriate candidates in the future due to their illegal structure and temporal join properties. We give more explanations in the paragraph 4.

## 5. The decomposition into prefix-based parent equivalence classes

See Appendix and [15].

**6. Processing the classes** In addition to what is described in the Appendix and [15], we do not create new equivalent classes with a prefix which is a sequence with non-continuous structure, during the recursive application of  $\theta_k$ . We can prove that such class will give us only non-continuous patterns that will not affect the whole process of sequence generation due to the independence of classes. In order to avoid the creation of such classes we need to exclude the patterns forming them, namely, the patterns with violated prefix (denote them violated patterns). It affects the frequent sequence enumeration inside a class, namely the temporal join (see Appendix, 6.2). We add several checkings to the join 3. The candidates of joining  $P \rightarrow A$  and  $P \rightarrow B$  will be:  $P \rightarrow AB$  if it is valid,  $P \rightarrow A \rightarrow B$  if  $P \rightarrow A$  is valid and  $P \rightarrow B \rightarrow A$  if  $P \rightarrow B$  is valid.

It can be proved (we omit this proof here due to space limits), that these modifications in addition to constraints introduced in paragraph 4 guarantee, that we’ll never receive violated sequences during processing the classes.

We also use the support-based (see Appendix) pruning during processing of classes and the temporal join, but in calculation of the support we do not check the continuity of a pattern, and therefore we may receive a value which is equal or bigger than the actual support value. This is discussed next.

---

### Algorithm 9 Filtering of non-appropriate patterns

---

```

 $L_2 = \{p\}, p \in L_2$  and  $p$  is frequent
and continuous;
 $L \leftarrow L_2$ ;
 $k \leftarrow 3$ ;
while( $L_k \neq \emptyset$ ) {
  for all  $p \in L_k$  {
    for all  $subs \subset p, |subs| = |p| - 1$ ;
      if ( $subs \notin L$ ) {
        remove  $p$  from  $L_k$ ;
        break;
      }
    }
  }
SupportCount( $L_k$ );
 $L = L_{k++}$ ;
}

```

---

cid	st_time	end_time	b_items	target
1	10	14	$C D$	$\bar{U}$
1	15	20	$A B C$	$U$
1	21	24	$A B F$	$\bar{U}$
1	25	31	$A C D F$	$U$
2	15	19	$A B F$	$\bar{U}$
2	20	25	$E$	$U$
3	10	15	$A B F$	$U$
4	10	15	$D G H$	$\bar{U}$
4	20	24	$B F$	$\bar{U}$
4	25	30	$A G H$	$U$

Figure 7. Original Input-Sequence Database

**7. Filtering the non-relevant sequences** As we already mentioned, after using the structure-based and support-based (see the Appendix) pruning techniques during processing of classes, we still may receive non-frequent sequences. Before checking support and continuity for the discovered patterns via scanning the customer-sequences, we save time by pruning potentially non-appropriate patterns. We use the pruning technique described in Section 3. It is clear that we cannot filter patterns of a class until we finish with all the classes (the subsequences of a pattern may belong to different classes). To start the checking, we filter  $F_2$  by scanning  $T_D$ , to form the seed set for the initial iteration of the Pruning procedure. Then we go from the next upper set  $L_3$  to the last one and, at each iteration  $i$ , filter the patterns of set  $L_{i-2}$  by generating appropriate subsequences for each pattern (see the Pruning subsection of Section 3), and check if they belong to the previous set  $L_{i-1}$ . If the pattern was not pruned, we scan  $T_D$  and check its frequency and continuity, using procedure Match, described earlier. The remaining patterns form the seed set for the next iteration. This is shown in Algorithm 9.

#### 4.1 Example

Consider the input database shown in Figure 7. The database has 8 basic events, 4 customers (specified by a cid), and 10 events in all. Figure 8 shows all the frequent events with a minimum support of 40% with their id-lists. The support of  $A$  is 60% and of  $B, F, U$  and  $\bar{U}$  is 50%. In Figure 9 the horizontal format for the events is depicted. All frequent pairs and 3-sequences, and lattice of equivalence classes, induced by them, are seen in Figure 14.

## 5 Performance evaluation and experiments

To compare the performance of our algorithms and represent their scale-up properties, we performed several experiments on a Dual Xeon workstation with a CPU clock

	CID	TID
A:	1	15
	1	21
	1	25
	2	15
	3	10
B:	4	25
	1	15
	1	21
	2	15
	3	10
F:	4	20
	1	21
	1	25
	2	15
	3	10
U:	4	20
	1	20
	1	31
	2	25
	3	15
$\bar{U}$ :	4	30
	1	14
	1	24
	2	19
	4	15
	4	24

Figure 8. Id-lists for the Atoms

dataset	$D$	$C$	$T$	$E$
DB1	102100	11540	9	20
DB2	102100	11540	9	11
DB3	81700	11530	7	20

Figure 10. Tested Datasets Parameters

cid	(item,tid) pairs
1	(A 15) (A 21) (A 25) (B 15) (B 21) (F 21) (F 25) (U 20) (U 31) ( $\bar{U}$ 14) ( $\bar{U}$ 24)
2	(A 15) (B 15) (F 15) (U 25) ( $\bar{U}$ 19)
3	(A 10) (B 10) (F 10) (U 15)
4	(A 25) (B 20) (F 20) (U 30) ( $\bar{U}$ 15) ( $\bar{U}$ 24)

Figure 9. Vertical-to-Horizontal Database Recovery

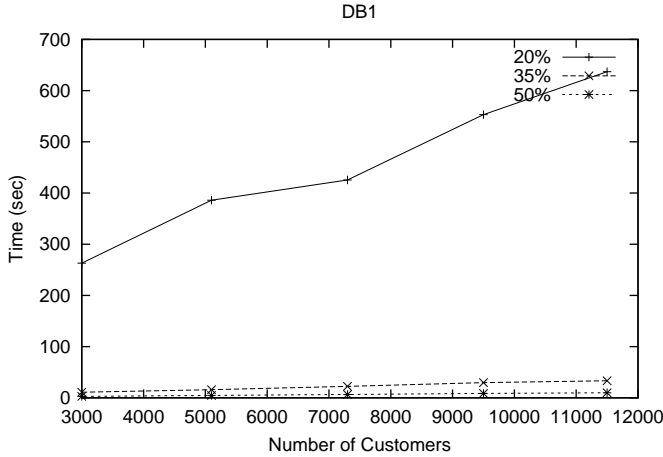


Figure 11. Scale-up of CTSPD

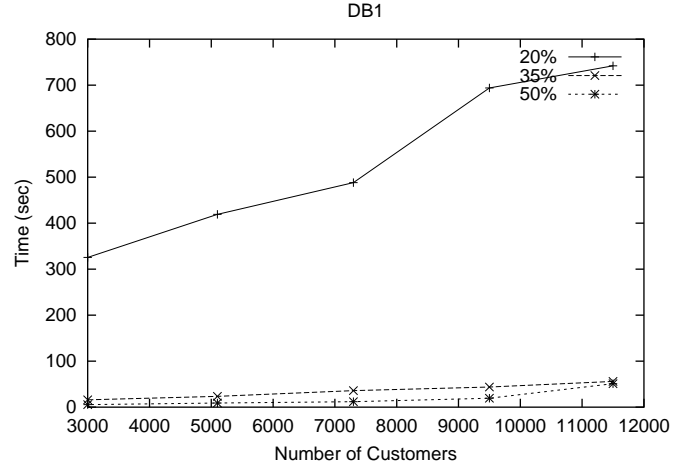


Figure 12. Scale-up of CSPADE

rate of 2.4 GHz, 2 GB RAM, running Linux. The data is stored on Oracle 9i. The algorithms were implemented in Java and intergrated with the FlexMine system[7].

### 5.1 The Tested Data

We evaluated the performance of the algorithms over relative big real-life data. We varies such parameters as number of customers in the dataset ( $C$ ), number of events per transaction ( $E$ ), average number of transactions per customer ( $T$ ) or number of transactions at all ( $|D|$ ). These parameters for the chosen 3 datasets (DB1, DB2 and DB3 respectively) are shown in Figure 10. DB3 is a 80% sample of DB1. In this paper we show the results of relative performance on DB1 and DB3.

### 5.2 Scale-up Properties

In this section we present the results of scale-up experiments for both our algorithms. We tested the scale-up both as depending on the number of customers and on the number of events in a single transaction. Figures 11 and 12 represent the scale-up experiments for both algorithms with the number of customers increased from 3000 to 11500. The results are shown for the DB1. We keep all parameters of dataset constant and fixed the minimum support, that

is 20%, 35% and 50% respectively. As shown, the execution time increased with the customers number increased. The CTSPD is faster and its execution time scale more uniformly. An execution time of CSPADE strongly increased on customer number equals 9500. We also made experiments with the number of events per transaction, that varies from 11 to 20, and with the number of generated rules, that varies from 1 to 482 in DB1, from 1 to 106 in DB2 and from 1 to 309 in DB3. Both algorithms show similar scale-up properties for both cases — an execution time increased with the number of events/generated rules increased. Also, we got experimental results, that demonstrate increasing the number of generated rules with the support decreased.

### 5.3 Relative Performance

Figure 13 presents the relative execution times for both algorithms. CSPADE is slower than CTSPD, because it generates a lot of candidates which are filtered at the end of process. The results are shown for DB1 with fixed minimum support, from 20% to 60%. As expected, the execution time increased as the minimum support decreased due to the increased number of generated rules. Also, we present the relative execution times as a function of the actual number of generated rules. These results received on DB3 and show that the execution time scales quite linearly.

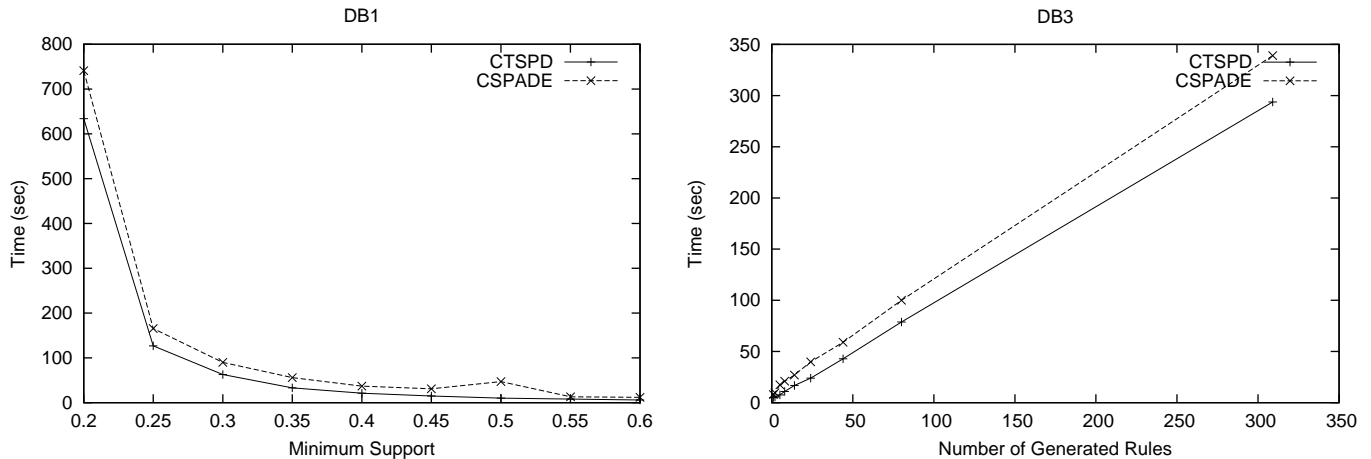


Figure 13. Relative Performance

## 6 Conclusions

In this paper a new problem definition was introduced for target rules mining, and two algorithms for solving it were proposed. Both algorithms, CTSPD and CSPADE, mine target sequential patterns, that have to be continuous, over the temporal database of customer transactions.

The first algorithm, CTSPD, is based on the Apriori principle, but adapted to our special application domain and considering only continuous candidates. The second algorithm, CSPADE, is based on SPADE and was adapted similarly to our continuous target based patterns.

We presented the relative performance and scale-up experiments for both algorithms, using real-life data. As expected, CTSPD was faster than CSPADE, which generated many more candidates. Both algorithms have comparable scale-up properties. The derived rules are used for improving the prediction of customers product upgrade that was introduced in [9]. After presenting the prediction experiments on our rules, we got results much better than any of those reported in [9].

## References

- [1] J. Adamo. *Data Mining for Association Rules and Sequential Patterns*. Springer-Verlag new York, 2001.
- [2] C. C. Aggarwal, C. M. Procopiuc, and P. S. Yu. Finding localized associations in market basket data. *Knowledge and Data Engineering*, 14(1):51–62, 2002.
- [3] R. Agrawal, T. Imieliski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216. ACM Press, 1993.
- [4] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th Int'l Conference on Very Large Databases*, Santiago, Chile, September 1994.
- [5] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of the Int'l Conference on Data Engineering (ICDE)*, Taipei, Taiwan, March 1995.
- [6] Antunes and Oliveira. Temporal data mining: an overview. In *KDD 2001 Workshop on Temporal Data Mining, 7th ACM SIGKDD*, San Francisco, CA, USA, August 2001.
- [7] R. Ben-Eliyahu-Zohary, C. Domshlak, E. Gudes, N. Liusternik, A. Meisels, T. Rosen, and S. E. Shimony. Fleximine - a flexible platform for kdd research and application development. *Annals of Mathematics and Artificial Intelligence*, 39(1-2):175–204, 2003.
- [8] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *Proceedings ACM SIGMOD*, pages 265–276, Tucson, Arizona, USA, May 1997.
- [9] A. Budker, E. Gudes, T. Hildeshaim, M. Litvak, E. Shimony, L. Amit, S. Meltzin, and G. Sotolorevsky. Targeting customers by mining usage time-series. In *CS/Stat'03 Second Haifa Winter Workshop on Computer Science and Statistics*, Haifa, Israel, December 2003.
- [10] Gary M. Weiss. Mining predictive patterns in sequences of events. In *Proc. of AAAI/GECCO Workshop on Data Mining with Evolutionary Algorithms: Research Directions*, 1999.
- [11] M. Gavrilov, D. Anguelov, P. Indyk, and R. Motwani. Mining the stock market: Which measure is best? (extended abstract).
- [12] H. Mannila, H. Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.
- [13] R. Srikant and R. Agrawal. Mining sequential patterns. generalizations and performance improvements. In *Proc. of the Fifth Int'l Conference on Extending Database Technology (EDBT)*, Avignon, France, March 1996.
- [14] J. T.-L. Wang, G. Chirn, T. Marr, B. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proceedings of the 1994 ACM SIGMOD*, pages 115–125, Minneapolis, Minnesota, US, May 1994.

