A Pattern Based Predictor for Event Streams

Cheng Zhou[a,b,*], Boris Cule[a], Bart Goethals[a]

[a] *Department of Mathematics and Computer Science, University of Antwerp, Belgium*
[b] *Science and Technology on Information Systems Engineering Laboratory, National University of Defense Technology, China*

Abstract

Recently, new emerging applications, such as web click-stream mining, failure forecast and traffic analysis, introduced a new challenging data model referred to as data streams. Mining such data can reveal up-to-date patterns, which are useful for predicting future events. Consequently, pattern mining in data streams is a popular field in data mining that presents unique challenges. The data is large and endlessly keeps on coming, making it impossible to store it, or to re-analyse historical data once it has been discarded. To solve this, we first present a novel method for mining sequential patterns from a data stream, in which we maximise memory usage in order to achieve higher accuracy in terms of results. In a second step, we use the discovered patterns in order to try to predict future events. We propose a number of ways to assign a score to each pattern in order to generate predictions. The prediction performance of these scoring strategies is then extensively experimentally evaluated. The predictor offers an opportunity for a faster detection and response to an important, though perhaps unexpected, event, which will occur in the future.

Keywords: pattern mining, event stream, prediction, lossy counting

1. Introduction

Recently, new emerging applications, such as web click-stream mining, failure forecast and traffic analysis, introduced a new challenging data model referred to as data streams. In this setting, datasets are large, and are considered to be infinite, as new data keeps on coming. As a result, historical data cannot be stored and cannot be reassessed once it has been processed. The field of sequential pattern mining in data streams has mostly been limited to a setting where the stream consists of incoming sequences. A sequential pattern is then typically defined as a sequence that often occurs within these incoming sequences (Ezeife and Monwar, 2007; Mendes, Ding, and Han, 2008; Koper and Nguyen, 2011). Some work has also gone into mining patterns from multiple streams, where a pattern is considered frequent if it occurs in many streams (Raıssi, Poncelet, and Teisseire, 2005; Chen, Wu, and Zhu, 2005; Tanbeer, Ahmed, Jeong, and Lee, 2008). We propose to mine sequential patterns in a setting where the stream consists of incoming sets of events, and a sequential pattern consists of events occurring in separate transactions that took place sequentially in time.

Since an event stream is continuous and unbounded, with events often coming at a high rate, we need to process the data efficiently and keep only the necessary information before discarding data from the past. In our framework, we first split the stream into batches of fixed size. This allows us to process each batch at a time, meaning we only need to update the discovered patterns at the end of each batch. Within a batch, we use the sliding window model (Mannila, Toivonen, and Verkamo, 1997), to transform each batch into a sequence database, in which we find frequent sequential patterns. Throughout the process, we maintain a list of sequential patterns that are frequent in the whole stream, and we update this list at the end of each batch, based on the newly discovered information.

Due to the nature of stream mining, we will never be able to keep exact information about the discovered patterns. In our work, we adopt the main lines of the Lossy Counting method (Manku and Motwani, 2002) to keep track of the frequent patterns. In Lossy Counting, a user-defined error parameter is used to limit the effect of approximations when computing the frequency of a pattern, which then allows us to give certain guarantees about the quality of the output. More concretely, Lossy Counting always finds all truly frequent patterns, but the output could also contain a number of patterns that are not truly frequent, or false positives. An error parameter is used to guarantee that the frequency of these false positives will never be far below the chosen frequency threshold. However, if the error bound is set too high, we will generate too many false positives, while the algorithm will run out of memory if it is set too low. In our work, we propose to dynamically determine the optimal error parameter by maximising the memory usage. We show that by doing so, we will either end up with a smaller error parameter than the user-chosen one for Lossy Counting, in which case we will generate fewer false positives, or with a higher error parameter, in which case we will still be able to produce output, while the Lossy Counting algorithm will run out of memory.

On top of mining patterns, another important application of stream mining is to try to predict what events will occur in the near future. Stated simply, if event a is often closely followed by event b, then we could use this pattern to predict that event

*Corresponding author. Tel.: +32483387164.
Email addresses: cheng.zhou@uantwerpen.be (Cheng Zhou), boris.cule@uantwerpen.be (Boris Cule), bart.goethals@uantwerpen.be (Bart Goethals)

b will occur soon if event a has just occurred. Naturally, rather than simply looking at how often events follow each other, there are numerous other ways to evaluate how useful a pattern is for making a prediction, and how reliable a generated prediction could be. In this paper, we propose a number of novel prediction strategies, and experimentally evaluate their performance in comparison to existing standard measures.

The rest of the paper is structured as follows. We describe the related work in Section 2, before presenting the preliminaries and the basic concepts in Section 3. In Section 4 we lay out the framework of our method and describe our algorithms in detail. The experimental evaluation is provided in Section 5, while our conclusions are presented in Section 6.

2. Related Work

Stream mining is an important field in data mining, with a number of interesting applications (Garofalakis, Gehrke, and Rastogi, 2002). Data streams pose a number of challenges which are not present when analysing static databases (Cheng, Ke, and Ng, 2008). Due to not being able to store all historical data from the incoming stream, a lot of work has gone into solving the problem of how to mine patterns in a setting where approximations must be made, while still being able to give reasonable guarantees on the quality of the output.

Lossy Counting (Manku and Motwani, 2002) was one of the first algorithms for finding frequent items and itemsets from a data stream. Lossy Counting provides an accuracy guarantee on the set of frequent items or itemsets and their reported frequencies by setting a user-specified support threshold σ, and an error bound ϵ. The method attempts to approximate the true frequency of each pattern, and is guaranteed to find all patterns that have a true frequency higher than σN, but could also output some patterns that have a true frequency lower than σN, but higher than $(\sigma - \epsilon)N$, where N is the current length of the data stream. A number of other studies were based on this idea (Arasu and Manku, 2004; Metwally, Agrawal, and El Abbadi, 2005; Dimitropoulos, Hurley, and Kind, 2008), all of them requiring a user-chosen error bound. Along similar lines, in a problem setting most similar to ours, Mendes, Ding, and Han (2008) proposed to mine sequential patterns in data streams. The memory usage of the Lossy Counting-like algorithms is, in theory, unbounded, and they will run out of memory if the error bound is set too low. However, if ϵ is set too high, the output will contain too many false positives. In this paper, we propose a method to mine frequent sequential patterns in a stream without setting a fixed error bound, thus increasing the quality of the output, and making the algorithm easier to use.

The prediction of future events has great importance in many applications, like prediction of users' requests on the web (Gündüz and Özsu, 2003; Jalali, Mustapha, Sulaiman, and Mamat, 2010), forecast of failures (Gu, Papadimitriou, Yu, and Chang, 2008; Martin, Méger, Galichet, and Becourt, 2012; Wang, Ma, Chow, and Tsui, 2014; Bala and Chana, 2015), traffic analysis (Chrobok, Kaumann, Wahle, and Schreckenberg, 2004; Merah, Samarah, Boukerche, and Mammeri, 2013), intensive care (Casanova, Campos, Juarez, Fernandez-Fernandez-Arroyo, and Lorente, 2015), etc.

Laxman, Tankasali, and White (2008) used Hidden Markov Models based on frequent episodes mined from historical data for a prediction task in a stream. However, this method can only predict whether a predefined target event type will occur, and is not suitable for predicting arbitrary events. Based on the definition of the latest minimal occurrence of the antecedent of an episode rule, Cho, Wu, Yen, Zheng, and Chen (2011) proposed two algorithms, *DeMO* and *CBS-Tree*, to match an episode rule over event streams for the prediction of the consequent event. Zhu, Wang, Wang, and Shi (2011) introduced an approach to match multiple episode rules for stream prediction after proposing an algorithm to generate all representative episode rules based on frequent closed episodes. However, using episode rules to predict an event in a sequence is computationally expensive because of the complex structure of an episode rule. Given the sequential nature of data streams, we propose a novel method for the prediction task in streams, using sequential patterns, which seem naturally closer to the stream structure.

All of these approaches get the patterns from historical data, while, in a streaming environment, we need to get the latest information from the current data since up-to-date patterns are useful for predicting. Therefore, we present a novel algorithm for finding patterns in data streams by maximising memory usage in order to achieve higher accuracy. Then, we make reliable on-the-fly predictions for future events in the stream based on generated patterns. Different from using traditional measures (support or confidence), we have designed new scoring functions by combining new measures of a pattern to improve the predictor.

3. Problem Statement

An event stream is a list of events represented as $ES = \langle (e_1, t_1), (e_2, t_2), \ldots, (e_n, t_n), \ldots \rangle$, where e_i is an event ($e_i \in E$), t_i is a time stamp ($t_i \in \mathbb{N}$ *and* $t_1 \leq t_2 \leq \ldots \leq t_n \leq \ldots$) and e_n is the latest event that has already occurred. E is the set of all possible event types and $\mathbb{N}$ is the set of natural numbers. For simplicity, in our examples, we omit the time stamps and implicitly assume they are consecutive natural numbers. Note that multiple events can occur at the same time stamp, which we take into account when we define sequential patterns below.

An event *sequence* is an ordered list of event sets denoted by $s = \langle a_1, a_2, \ldots, a_m \rangle$, with $a_i \subseteq E$. We denote $|s|$ as the length of sequence s. The *prefix* of a sequence s is defined as $\bar{s} = \langle a_1, a_2, \ldots, a_{m-1} \rangle$. A sequence $s' = \langle b_1, b_2, \ldots, b_k \rangle$ is said to be a *subsequence* of s if there exist integers $1 \leq i_1 < i_2 < \ldots < i_k \leq m$ such that $b_1 \subseteq a_{i_1}, b_2 \subseteq a_{i_2}, \ldots, b_k \subseteq a_{i_k}$, denoted as $s' \sqsubseteq s$ (if $s' \neq s$, written as $s' \sqsubset s$). Given a set of sequences S and a sequence p, we define the *frequency* of p in S as $fre(p) = |\{s \in S \,|\, p \sqsubseteq s\}|$.Given a minimum support threshold *minSup*, if $fre(p) \geq \lceil minSup \times |S| \rceil$, p is considered a frequent sequential pattern in S.

Once a frequent pattern $p = \langle c_1, c_2, \ldots, c_k \rangle$ is discovered, its prefix $\bar{p}$ must be frequent too. Therefore, each pattern

can also be used to generate a *rule*: $\langle c_1, c_2, \ldots, c_{k-1}\rangle \Rightarrow c_k$. The *confidence* of this rule is defined as $conf(\bar{p} \Rightarrow c_k) = fre(p)/fre(\bar{p})$. Given a sequence $s = \langle a_1, a_2, \ldots, a_m\rangle$, if there exists an i, $0 < i \leq m$, such that $\bar{p} \sqsubseteq s'$ while $p \not\sqsubseteq s'$ where $s' = \langle a_i, a_{i+1}, \ldots, a_m\rangle$, we say that p *matches* s. We define the minimum match length of p in s as $minMatch(p, s) = \min_{0<i\leq m}\{ts(m) - ts(i) + 1\}$, where $ts(m)$ and $ts(i)$ denote the time stamps at which event sets a_m and a_i occurred, respectively. For example, given a frequent pattern $p = \langle a, b, c\rangle$ and a sequence $s = \langle a, b, c, a, b, d\rangle$, pattern p matches s and $minMatch(p, s) = 3$.

In order to efficiently process the input stream, we will divide it into batches. Given a batch size of l time units, we define the i_{th} batch of event stream ES as

$$B_i = \{(e, t) | (e, t) \in ES \text{ and } t \in [(i-1)l + 1, il]\}.$$

We traverse the stream using a sliding window of length w. For each time stamp ts, with $ts \in \mathbb{N}$, we define the corresponding window as

$$W_{ts} = \{(e, t) | (e, t) \in ES \text{ and } t \in [ts - w + 1, ts]\}.$$

Note that each window is itself a sequence. We denote the set of windows generated from a batch B_i as S_i. Note that the first $w-1$ windows of B_{i+1} contain some events that actually occurred in B_i, which enables us to capture patterns whose occurrences may have spanned over two batches.

Example 1. *Given an input stream*

$$\langle a, c, b, d, e, a, b, d, a, c, b, f, d, a, c, b, d, e, f, a, c, b, d \ldots\rangle.$$

The first two batches of size 10 are

$$B_1 = \langle a, c, b, d, e, a, b, d, a, c\rangle$$

and

$$B_2 = \langle b, f, d, a, c, b, d, e, f, a\rangle.$$

Assume that $w = 4$. *The windows obtained from* B_1 *and* B_2 *are shown in Table 1.*

Table 1: Example of Sliding Windows

Windows of B_1 (S_1)		Windows of B_2 (S_2)	
ID	Sequence	ID	Sequence
1	$\langle a\rangle$	11	$\langle d, a, c, b\rangle$
2	$\langle a, c\rangle$	12	$\langle a, c, b, f\rangle$
3	$\langle a, c, b\rangle$	13	$\langle c, b, f, d\rangle$
4	$\langle a, c, b, d\rangle$	14	$\langle b, f, d, a\rangle$
5	$\langle c, b, d, e\rangle$	15	$\langle f, d, a, c\rangle$
6	$\langle b, d, e, a\rangle$	16	$\langle d, a, c, b\rangle$
7	$\langle d, e, a, b\rangle$	17	$\langle a, c, b, d\rangle$
8	$\langle e, a, b, d\rangle$	18	$\langle c, b, d, e\rangle$
9	$\langle a, b, d, a\rangle$	19	$\langle b, d, e, f\rangle$
10	$\langle b, d, a, c\rangle$	20	$\langle d, e, f, a\rangle$

Given an input stream, the goal of our method is two-fold. First of all, we wish to reliably mine frequent sequential patterns, while taking into account that in a streaming context we cannot keep an exact frequency count for each pattern. We will show that we are able to guarantee that we find all truly frequent patterns, while keeping the number of false positives as low as possible. Our second goal is to then use the discovered patterns in order to predict future events. In order to efficiently mine patterns, we will update the set of discovered patterns only after processing an entire batch of data. However, based on the already discovered patterns, we are able to make new predictions at any moment in time.

4. Algorithms

Since the goal of our method is two-fold, our algorithm, *SPEP* (Sequential Pattern based Event Prediction), also consists of two main stages, a frequent sequential pattern miner (*SPEP_PM*) and an event predictor (*SPEP_EP*). Algorithm 1 shows the high level structure of the algorithm. Line 1 initialises a batch id i, line 2 initialises the content of the current batch *batchData* and line 3 initialises an error bound ϵ. Lines 4-17 contain the main part of *SPEP*. When an event comes in (line 4), we update the content of the current batch (line 5). Lines 6-8 generate a prediction based on the already discovered sequential patterns. Note that this can only be done once i is larger than 1, since no patterns are found until the first batch has been processed. If we have reached the end of a batch (line 9), we mine sequential patterns in the batch (line 11). After processing the batch, we update the batch id i and reset the *batchData* variable (line 12-13). Line 14 stores the last window of the last batch since this historical data is needed to make predictions at the beginning of the next batch (see line 7). Finally, we update the error bound ϵ (line 15).

In the following subsections, we discuss the individual elements of the main algorithm in detail.

4.1. Sequential Pattern Miner

Algorithm 2 shows the outline of mining frequent sequential patterns from a batch B_i. There are three phases in *SPEP_PM*. In the first phase, we run a modified SPADE algorithm (Zaki, 2001) to get the set of all frequent sequential patterns P, which we sort by ascending pattern size (line 2). In the second phase, *updatePatternTable* is called to update the pattern table T, which contains the sequential patterns mined from the complete event stream seen so far. After the first two phases, some of the patterns in T may not be frequent enough any more. Thus, in the last phase, we prune the patterns that are not frequent (line 4). We discuss these three phases in detail in the following sections.

4.1.1. Modified SPADE algorithm

Some algorithms have been developed to mine sequential patterns in data streams. Some important algorithms (Chen, Wu, and Zhu, 2005; Mendes, Ding, and Han, 2008; Koper and Nguyen, 2011) are derived from PrefixSpan (Pei, Han,

Algorithm 1 *SPEP*

Input: event stream ES, batch size $|B|$, sliding window length w, prediction time span *span* and maximum number of patterns kept in memory *maxNum*.

```
1:  i = 1;
2:  batchData = ∅;
3:  ε_i = 0;
4:  while ES.nextEvent ≠ null do
5:     update batchData;
6:     if i > 1 then
7:        SPEP_EP(batchData, H_i, span);
8:     end if
9:     if end of a batch then
10:       B_i = batchData;
11:       SPEP_PM(B_i, i, ε_i);
12:       i++;
13:       batchData = ∅;
14:       H_i = W_{(i−1)×|B|};
15:       ε_i = updateErrorBound(maxNum, i);
16:    end if
17: end while
```

Algorithm 2 $SPEP_PM(B_i, i, \epsilon_i)$

```
1: S_i = the set of sliding windows from batch B_i;
2: P = modifiedSPADE(S_i, ε_i);
3: updatePatternTable(P, i);
4: pruning(ε_i, i);
```

Mortazavi-Asl, Wang, Pinto, Chen, Dayal, and Hsu, 2004). PrefixSpan shows good performance and scales well in memory, but, when dealing with dense databases, the performance of PrefixSpan may be worse than that of SPADE (Gomariz, Campos, Marin, and Goethals, 2013). As a result, we choose SPADE as our sequential pattern miner. Note that any other existing sequential pattern miner would have done the job, too.

In order to make good predictions, we not only want to get the frequency of each pattern, but we also wish to determine the lengths of its minimal occurrences. In a given window W_i within the input stream ES, a subsequence s of W_i is called a *minimal window* of pattern p if $p \sqsubseteq s$ and $\forall s' \sqsubset s$, $p \not\sqsubseteq s'$.

We therefore made two modifications to the original SPADE algorithm. First, we added a function to get the minimal windows of a frequent pattern by tracking the occurrences of all the items composing the pattern. Second, as the densities of different batches in the data stream can vary, we noted that it can be difficult to set an appropriate support threshold to get enough patterns for prediction and, at the same time, ensure the algorithm will not run out of memory in one of the batches. Therefore, we add a memory limit to SPADE (*spadeMemory*). We monitor the memory usage of SPADE in the process of enumerating frequent sequences (Zaki, 2001). First we run SPADE with an absolute support threshold $\lfloor \epsilon_i \times |B| \rfloor + 1$ to mine the patterns in a set of sequences S_i. If this proves infeasible, we use an optimisation method to try to find the smallest absolute support threshold sup_i at which the memory usage of SPADE is smaller than a user defined memory limit. Then, the new error bound is set to $\epsilon_i = \frac{sup_i - 1}{|B|}$.

4.1.2. Updating pattern table

We use a hashmap T to store the sequential patterns mined from the event stream. The key of T is the corresponding sequential pattern p and the value of the key contains five attributes of the pattern:

(1) *sumF*: the sum of the frequencies of p found by SPADE in all batches processed so far;

(2) *lastF*: the frequency of p found by SPADE in the last processed batch;

(3) Δ: the maximum possible error for the frequency of p (i.e., the maximum possible frequency of p in batches in which p was infrequent);

(4) *wSum*: the sum of the lengths of known minimal windows which cover p in all batches processed so far;

(5) *wCount*: the count of the known minimal windows which cover p in all batches processed so far.

Example 2. *Consider Example 1, and suppose that we want to get the minimal windows of pattern $\langle d, a \rangle$ after B_2. Denote the lengths of the minimal windows of pattern $\langle d, a \rangle$ in batches 1 and 2 as $minWins1 = \{3, 2\}$ and $minWins2 = \{2, 2, 4\}$, respectively. After mining the frequent patterns from batch 2, we could have one of two possible cases. If $\langle d, a \rangle$ already exists in the pattern table T, the set of lengths of the known minimal windows of pattern $\langle d, a \rangle$ after B_2 is $minWins = \{3, 2, 2, 4\}$, which is not the bag union of $minWins1$ and $minWins2$ since the window spanning over both batches, $\langle d, a, c \rangle$, would be duplicated. If $\langle d, a \rangle$ was not frequent in batch 1, the set of lengths of the known minimal windows of pattern $\langle d, a \rangle$ after B_2 is $\{2, 2, 4\}$ since the minimal windows of the pattern in batch 1 are not available.*

In order to solve the problem described in Example 2, we need to mark the minimal windows spanning over two batches. In our implementation we do this by adding a minus before them, i.e., $minWins2 = \{-2, 2, 4\}$ in Example 2.

The algorithm for updating the pattern table is given in Algorithm 3. The algorithm consists of two stages. In the first stage, lines 1-23 update pattern table T for each sequential pattern $p \in P$. If pattern p is already in T (line 2), we first compute the sum and count of new minimal windows of p (lines 3-4). Then, we update the pattern's *sumF*, *lastF*, *wSum* and *wCount* attributes (lines 5-8). If p is not yet in T, we store p and its new attributes into T (lines 9-21). Lines 10 and 11 compute the sum and count of *minWins* of p. Note that there is no error for the frequency of p in batch 1 (line 12), while we need to compute the error afterwards (lines 13-19). The method to get the maximum possible positive error (Δ) for the frequency of p is based on three observations:

1. An upper bound for the Δ of a pattern which is not yet in T is $UPB1 = \lfloor \epsilon_i \times (i-1) \times |B| \rfloor$, where i is the *id* of current batch.

2. One occurrence of a pattern p contributes at most $w - |p| + 1$ to the frequency of the pattern.

3. If pattern X is the prefix of pattern Y, the frequency of Y cannot be larger than the frequency of X. The maximum possible frequency of X in the past batches should be $maxFrePast(X) = X.maxF - X.lastF$, where $X.maxF = X.sumF + X.\Delta$. Assume that X occurs k times in the past batches, it holds that Y occurs at most k times in the past batches and the maximum possible frequency of Y should be $maxFrePast(X) - k$ based on observation 2. In other words, another upper bound for the Δ of a pattern Y that is frequent in the current batch, and is not yet in T, but its prefix X already is in T, is $UPB2 = maxFrePast(X) - min\{k\}$, where $min\{k\}$ is the minimum possible value of k. We know that $min\{k\} = \lceil \frac{maxFrePast(X)}{w-|X|+1} \rceil$ based on observation 2.

As a result, we set Δ of a pattern p whose prefix $\bar{p}$ is not in T to $\lfloor \epsilon_i \times (i-1) \times |B| \rfloor$ (line 14). However, Δ of a pattern whose prefix is in T is set to $min\{UPB1, UPB2\}$ based on the observations above (lines 15-18). Note that this is why we sort the found patterns by ascending pattern size, ensuring that a prefix of a pattern is always inserted into T before the pattern itself. Finally, we update the attributes of pattern p (line 20) and store it and its attributes into T (line 21).

In the second stage, we update the Δ attribute for the patterns in T that were not found in the current batch. Line 25 finds patterns P' that exist in T but not in P. The Δ of each such pattern is updated by adding the maximal possible frequency of an infrequent pattern in the current batch, i.e., $\lfloor \epsilon_i \times |B| \rfloor$ (lines 26-28).

Our algorithm can be asked to output the discovered frequent patterns at any point. We output a pattern X if $X.sumF + X.\Delta \geq \lceil i \times |B| \times minSup \rceil$, where i is the number of batches processed so far. By doing so, we are guaranteed to output all truly frequent patterns (since Δ is an upper bound for the true frequency of a pattern in batches where the pattern was infrequent), and, as discussed already, we output fewer false positives than existing methods.

4.1.3. Pruning

After updating and adjusting the pattern table, some patterns in T may have become infrequent. We say a pattern in T is not frequent enough if the upper bound of its frequency is lower than the current error threshold (obtained by multiplying the number of processed windows with the current value of the dynamic error parameter). The complete method for removing the infrequent patterns from the pattern table is given in Algorithm 4.

4.2. Updating Error Bound

Algorithm 5 describes the procedure for updating the error bound. If the size of the pattern table has become larger than *maxNum*, we first sort all the patterns stored in T by descending *maxF* and store them in a pattern list *list* (line 2). We then get the *maxF* value of the highest ranked pattern that no longer fits into the table and find the *index* of the last pattern in *list* whose *maxF* is larger than this value (lines 3-4). Then, we remove the patterns ranked below the *index* (lines 5-7). Finally, we compute a new error bound (line 8), and return it if it is larger

Algorithm 3 *updatePatternTable*(P, i)

```
1:  for each pattern p ∈ P do
2:    if p ∈ T then
3:      sum = ∑_{m∈p.minWins and m>0} m;
4:      count = |{m|m ∈ p.minWins and m > 0}|;
5:      p.sumF + = p.fre;
6:      p.lastF = p.fre;
7:      p.wSum + = sum;
8:      p.wCount + = count;
9:    else
10:     sum = ∑_{m∈p.minWins} |m|;
11:     count = |p.minWins|;
12:     Δ = 0;
13:     if i > 1 then
14:       Δ = ⌊ε_i × (i − 1) × |B|⌋;
15:       if p̄ ∈ T then
16:         Δ_1 = maxFrePast(p̄) − ⌈maxFrePast(p̄) / (w−|p̄|+1)⌉;
17:         Δ = min{Δ, Δ_1};
18:       end if
19:     end if
20:     value = (p.fre, p.fre, Δ, sum, count);
21:     T.put(p, value);
22:   end if
23: end for
24: if i > 1 then
25:   P′ = T − P;
26:   for each pattern p′ ∈ P′ do
27:     p′.Δ+ = ⌊ε_i × |B|⌋;
28:   end for
29: end if
```

Algorithm 4 *pruning*(ϵ_i, i)

```
1: for each pattern p ∈ T do
2:   p.maxF = p.sumF + p.Δ
3:   if p.maxF ≤ ε_i × i × |B| then
4:     remove p from T;
5:   end if
6: end for
```

than the previous error bound (line 10), otherwise, the error bound remains the same (line 12).

Consider Example 1 from Section 3, and suppose the maximum number of patterns to be kept in memory is set to 10. The pattern table after processing batch 1 is shown in Table 2. Since we can only keep a maximum of 10 patterns in memory, all patterns with a frequency of 3 or lower had to be removed. As a result, the error bound for the next batch has been set to $\epsilon_2 = 0.3$. Therefore, we try to use an absolute support threshold $0.3 \times 10 + 1 = 4$ to mine frequent patterns in S_2 (as defined in Table 1).

The pattern table after processing batch 2 is shown in Table 3. The last two patterns in this table were infrequent in batch 1, so their Δ attribute has been set to 3 (the highest possible frequency of a pattern that was infrequent in batch 1). However, note that

Algorithm 5 *updateErrorBound*($maxNum, i$)

Output: ϵ_i

1: **if** $T.size() > maxNum$ **then**
2: $list = getSortedPatterns()$;
3: $b = maxF$ of the $(maxNum + 1)$th pattern in $list$;
4: i = index of the last $p \in list$ whose $maxF > b$;
5: **for** jth pattern $p \in list$ where $j > i$ **do**
6: remove p from T;
7: **end for**
8: $b = \frac{b}{(i-1)\times|B|}$;
9: **if** $b > \epsilon_{i-1}$ **then**
10: **return** b;
11: **else**
12: **return** ϵ_{i-1};
13: **end if**
14: **else**
15: **return** ϵ_{i-1};
16: **end if**

Table 2: Pattern Table after B_1

Pattern	*maxF*	*sumF*	*lastF*	Δ	*wSum*	*wCount*
$\langle a\rangle$	9	9	9	0	3	3
$\langle b\rangle$	8	8	8	0	2	2
$\langle d\rangle$	7	7	7	0	2	2
$\langle b, d\rangle$	6	6	6	0	4	2
$\langle c\rangle$	5	5	5	0	2	2
$\langle a, b\rangle$	5	5	5	0	5	2
$\langle e\rangle$	4	4	4	0	1	1
$\langle a, c\rangle$	4	4	4	0	4	2
$\langle d, a\rangle$	4	4	4	0	5	2

the *wSum* and *wCount* values of these two patterns are based on batch 2 alone. Meanwhile, pattern $\langle e\rangle$, that was frequent in batch 1, has now dropped out of the table. The *maxF* of $\langle e\rangle$ after batch 2 is 7, which is not high enough to be considered frequent. Indeed, note that all patterns shown in Table 3 have a *maxF* value of at least 9.

Table 3: Pattern Table after B_2

Pattern	*maxF*	*sumF*	*lastF*	Δ	*wSum*	*wCount*
$\langle a\rangle$	16	16	7	0	5	5
$\langle b\rangle$	16	16	8	0	4	4
$\langle d\rangle$	16	16	9	0	4	4
$\langle c\rangle$	12	12	7	0	3	3
$\langle b, d\rangle$	11	11	5	0	9	4
$\langle a, b\rangle$	9	9	4	0	11	4
$\langle a, c\rangle$	9	9	5	0	6	3
$\langle d, a\rangle$	9	9	5	0	11	4
$\langle f\rangle$	9	6	6	3	2	2
$\langle c, b\rangle$	9	6	6	3	4	2

4.3. Event Predictor

The algorithm for predicting future events is shown in Algorithm 6. Note that we use a window of length w to discover patterns, while *span* defines the time span we want to predict. Line 1 gets the training data *td* by getting the latest events in a window of length $w - span$. Then we find the patterns *ps* matching the training data from the pattern table T (line 2). Two optional parameters, *minSup* and *minConf*, can be used to limit the patterns only to those with $sumF + \Delta \geq \lceil minSup \times (i-1) \times |B| \rceil$ and $conf \geq minConf$, where $conf = p.sumF/\bar{p}.sumF$. We use a hashmap Z to store the predicted events obtained from *getPredictions* (line 3). The key is the predicted event and the value of this key is the score of the predicted event which measures the likelihood of the event appearing in the future. We present a discussion of a number of possible scoring policies in Section 5.5. Given a user-defined parameter k, we return the top k predicted events after sorting on their scores in descending order.

Algorithm 6 *SPEP_EP*($batchData, H_i, span$)

1: $td = getTrainingData(batchData, H_i, span)$;
2: $ps = findMatchedPatterns(td, T, minSup, minConf)$;
3: $Z = getPredictions(ps, k)$

Example 3. *Let us go back to Example 1, and assume we want to predict which events will occur after batch 2. Suppose the sliding window length is* 4*, and the time span we want to predict is* 2*, so we can use the latest events in a window of length 2, i.e.,* $\langle f, a\rangle$ *after batch 2 is processed, to predict the future events. Assume that the pattern table after batch 2 contains, among others, patterns* $\langle a, b\rangle$ *and* $\langle a, c\rangle$*, and that these two patterns are the only patterns matching the training data* $\langle f, a\rangle$*. Events b and c will, therefore, be predicted if* $k \geq 2$ *(assuming both patterns satisfy the frequency and confidence thresholds).*

5. Experimental Evaluation

We implemented our method in java and ran it on a PC with Intel Xeon CPU at 2.90GHz, setting the maximum heap size to 2GB. The operating system was Ubuntu 12.04.4. All experiments were performed on four real-life datasets, with very differing characteristics, allowing us to cover a variety of settings.

5.1. Datasets

FIFA is a long sequence obtained by merging 20 450 sequences of click stream data from the website of the 1998 FIFA World Cup[1]. *KOSARAK* is a long sequence obtained by merging web sessions from a Hungarian news portal[2]. Here the sequences shorter than 30 items have been removed to keep only 46815 sequences. *BIBLE* contains the full text of the Bible,

[1] http://www.philippe-fournier-viger.com/spmf/datasets/FIFA.txt
[2] http://fimi.ua.ac.be/data/kosarak.dat

where each word is considered to be an event[3]. *ALARM* contains a sequence of alarms triggered in a factory, stretching over 18 months. An entry in the dataset consists of a time stamp and an event type. Note that the first three datasets are dense, and have no time stamps, but, implicitly, the events are assumed to have "taken place" on consecutive time stamps. The first two datasets are merged to form two long streams, respectively, by adding a gap of 50 time stamps between the original sequences, since we never use a sliding window greater than 50 on these two datasets. In this way, we wish to avoid erroneously implying that events at the end of one stream have an influence on the start of the next one. The time stamps in *ALARM* are expressed in seconds, and most time stamps are not associated with any event. Furthermore, this dataset also contains events that occur at the same time, which is never the case in other datasets. *BIBLE*, unlike the other three datasets, does not originate from a stream, but we wanted to test our method on text data, which, in other contexts, could indeed form an incoming data stream. Table 4 summarises the characteristics of the four real-life datasets. The second column contains the number of events in the stream, the third the support of the most frequent item, the fourth the average length of a sequence in the first two datasets and the fifth the number of unique events in the stream.

Table 4: Characteristics of the used datasets

Dataset	Size	MaxSup	AvgLen	#Items
FIFA	741 092	0.018	36.24	2 990
KOSARAK	3 510 442	0.013	74.99	25 926
BIBLE	787 066	0.076	N/A	13 905
ALARM	514 502	0.023	N/A	5 001

5.2. False Positive Rate

In our first set of experiments, we demonstrate that our pattern mining method improves the accuracy of existing methods. We compare our method to the *SS_BE* algorithm proposed by Mendes, Ding, and Han (2008), which, like the original Lossy Counting idea, mines patterns from a data stream with a fixed user-chosen error bound. In order to evaluate the false positive rate for the two methods, we first needed to obtain an exact number of truly frequent patterns. For this, we treated *FIFA*, *KOSARAK*, *BIBLE* and *ALARM* as static sequence databases, and used the SPADE method to find all frequent patterns. We set the minimum support threshold *minSup* to be 0.01, 0.01, 0.02 and 0.0003 for *FIFA*, *KOSARAK*, *BIBLE* and *ALARM*, respectively. Using a sliding window length of 10 for all four datasets, we set the batch size to be 20 000, 20 000, 10 000 and 200 000 for *FIFA*, *KOSARAK*, *BIBLE* and *ALARM*, respectively. The error bounds of *SPEP_PM* at the end of the four datasets were 0.00015, 0.00155, 0.0004 and 0.00009, respectively. We then needed to set fixed error bounds for the *SS_BE* algorithm. We know that *SS_BE* would run out of memory if the error bound was set lower than the final values we obtained for our dynamic error bound. We therefore chose slightly higher values for the *SS_BE* error bounds, namely, 0.0002, 0.0016, 0.0005 and 0.0001 for the four datasets, respectively. This allowed us to make a fair comparison, but it should be noted that, with these values, *SS_BE* could still be expected to run out of memory soon, as the streams continue to grow. Table 5 shows the number of patterns generated by SPADE (column Baseline) and the two streaming algorithms, output after every 10 batches. Note that the N/As in the columns of *FIFA* mean that this data stream is not long enough to run 100 batches. We can see that *SPEP_PM* always has a lower false positive rate than *SS_BE*. There are two reasons for this. Firstly, as a result of maximising memory usage, *SPEP_PM* uses a lower error bound than *SS_BE*, especially in the early batches. Secondly, we minimise the maximum possible positive error for the frequency of a pattern based on the information of its prefix, as shown in Algorithm 3.

5.3. Efficiency Analysis of Pattern Mining

The runtime of our algorithm is composed of the runtime of the pattern miner (*SPEP_PM*) and the event predictor. The *minSup* and *minConf* only affect the number of patterns used for making a prediction and we find that the parameters do not make a difference for the runtime of the event predictor. Therefore, we now compare the efficiency of our pattern miner against another state-of-the-art sequential pattern mining algorithm (*SS_BE*) for streaming data at different batch sizes and sliding window lengths, respectively. That is because other parameters don't influence the runtime of pattern mining for streaming data except the error bound parameter. However, we do not analyse the impact of error bound since our pattern miner dynamically determine the optimal error bound by maximising the memory usage.

Figure 1 shows the runtimes of the algorithms with various batch sizes where the length of the processed stream is 720 000, the sliding window length is fixed at 10 and *maxNum* for *SPEP_PM* is set to 50 000 for each dataset. The results show that, as the batch size increases, the runtimes of *SPEP_PM* and *SS_BE* decrease. This is because a larger batch size will reduce the number of batches that need to be processed. We find that *SS_BE* is more time consuming with smaller batch sizes since *SS_BE* needs to maintain the information of a larger number of patterns.

Figure 2 shows the runtimes of the algorithms for a varying length of sliding window. In this experiment, the length of the processed stream is 720 000, the batch size is 6 000 and *maxNum* for *SPEP_PM* is set to 50 000 for each dataset. Generally, the runtime grows when the length of sliding window increases since there will be more frequent patterns in a batch with a larger sliding window. We find that the runtime of *SPEP_PM* is more stable as on the *BIBLE* and *KOSARAK* datasets, there is actually a jump in runtime for *SS_BE*. This occurs because *SS_BE* keeps much more patterns when using the sliding window length at the jump point.

[3]http://www.philippe-fournier-viger.com/spmf/datasets/BIBLE.txt

Table 5: Comparison of the False Positive Rates of the *SPEP_PM* and *SS_BE* algorithms

Batches	*FIFA*			*KOSARAK*			*BIBLE*			*ALARM*		
	Baseline	*SPEP*	*SS_BE*	Baseline	*SPEP*	*SS_BE*	Baseline	*SPEP*	*SS_BE*	Baseline	*SPEP*	*SS_BE*
10	270	270	279	52	55	78	188	188	196	69	78	108
20	285	285	293	56	56	75	198	198	203	39	55	90
30	286	286	289	53	53	72	196	196	206	35	45	88
40	282	282	295	54	54	75	189	189	193	30	40	74
50	285	285	297	52	52	74	170	170	181	25	35	68
60	289	289	301	53	53	74	171	171	180	23	37	70
70	288	288	300	53	53	73	171	171	176	20	28	65
80	288	288	299	52	52	72	N/A	N/A	N/A	16	25	60
90	N/A	N/A	N/A	53	53	72	N/A	N/A	N/A	18	23	54
100	N/A	N/A	N/A	52	52	74	N/A	N/A	N/A	19	27	59

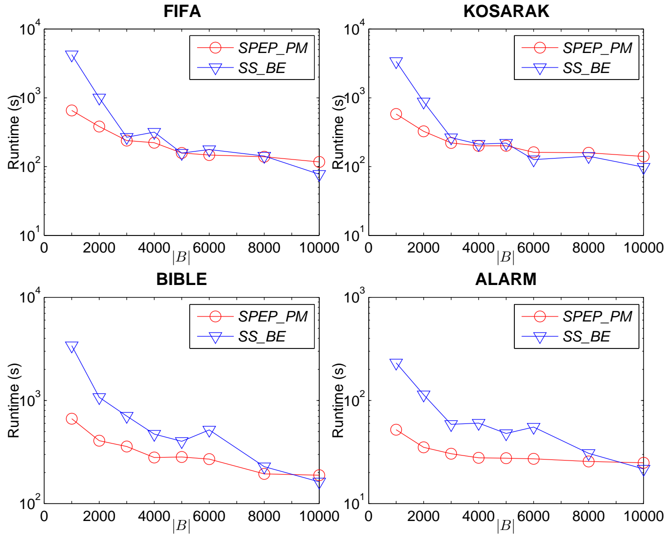

Figure 1: Impact of the batch size on the runtime

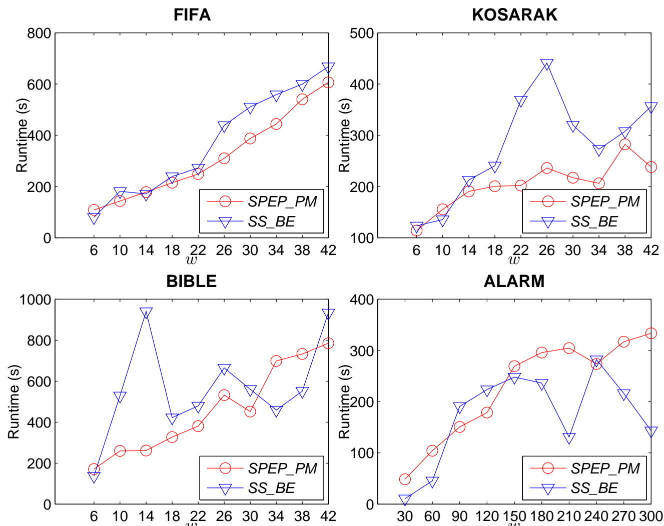

Figure 2: Impact of the sliding window length on the runtime

5.4. Evaluation Framework for the Predictor

The best known ways to measure the performance of a predictor are precision and recall:

$$precision = \frac{\text{Number of events predicted correctly}}{\text{Number of predicted events}}$$

$$recall = \frac{\text{Number of events predicted correctly}}{\text{Number of events that took place}}$$

The main metric we use for evaluation is the F_1-measure, which is defined as:

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall}$$

In our context, these definitions are not as trivial as they seem. We use a sliding window of length w to search for patterns, and yet we are able to make predictions at each possible time stamp for a user-chosen prediction *span*. Therefore, a prediction we make at time t is based on events that occurred in the time interval $[t - w + span + 1, t]$, and the prediction stays valid for the time interval $[t + 1, t + span]$.

For example, if the prediction span is 10, and the sliding window length 60, a prediction generated at time stamp 100 will be based on events occurring in interval $[51, 100]$. If we predict that event x will occur within the next 10 seconds, this prediction will be deemed correct if x occurs at any time stamp in interval $[101, 110]$.

In our experiments, we make new predictions at each time stamp. Obviously, the longer the sliding window, the higher the chance that some (or even most) predictions will be repeated from one time stamp to another. For example, assume we have identified a strong rule $a \Rightarrow b$, and that the window length is 60 and prediction span is 10. If event a occurred at time stamp 5, we will predict that b will occur in the time interval $[6, 15]$. However, at time stamp 6, if b has not occurred, we will probably again predict that b will occur in interval $[7, 16]$ (based on the same a that occurred at time stamp 5). We could, in theory, continue to make the same prediction based on the same a until time stamp 55, by which time we could again start predicting b based on another a or another rule altogether. It would be counterintuitive to count each such duplicate prediction as a separate prediction, as both the number of correct and incorrect predictions could end up being higher than the number of events that actually took place. We will therefore now formally describe our evaluation framework, that will be used to compute precision and recall in this context.

First of all, we define the *event count* at time t, EC_t, as the number of events that took place at time t, $EC_t = |(e, t) \in ES|$.

We say an event e that occurred at time t was predicted correctly if a prediction was made at any time stamp in the interval $[t - span, t-1]$ that event e would occur within time *span* from the prediction being made. At each time point, we will therefore need to keep track of the currently predicted items. To start with, we denote the set of predictions made at time t (through the top k selection) as PM_t. At time t, the set of valid predictions will consist of predictions made in the interval $[t-span+1, t-1]$ and the newly made predictions PM_t. We define the set of *existing valid predictions* at time t as

$$PP_t = \cup_{i\in[t-span,t-1]}(PM_i - \{(e,t') \in ES | t' \in [i+1, t]\}).$$

We then define the set of *currently valid predictions* at time t as $CP_t = PP_t \cup PM_t$. Note that the set of valid predictions includes all items that have been predicted at the last *span* time stamps, apart from those that have already occurred in the stream after they have been predicted. Therefore, in order to count how many predictions have actually been made, we need to compute how many new items have been added to the set of valid predictions at each time stamp. Note that, at time t, all predictions made at time stamp $t - span$ will expire. We define the set of *expired predictions* at time t as $EP_t = CP_{t-1} \cap NP_{t-span}$, where NP_{t-span} is the set of new predictions added to CP_{t-span} at time $t - span$. Recursively, we define the set of *new predictions* at time t as $NP_t = PM_t - (PP_t - EP_t)$. Having done this, we can compute the number of predictions made at time t as $PC_t = |NP_t|$. Finally, we define the *correct predictions count* at time t, CPC_t, to be equal to the number of events that took place at time t that were predicted correctly. Formally, $CPC_t = |\{(e,t) \in ES | e \in CP_{t-1}|$.

At any given moment, we can evaluate the performance of our predictor on the stream seen so far. To do this, we need to compute three values. The first is the total number of events that have occurred since we started predicting, $EC = \sum_{t\geq t_s} EC_t$, where t_s is the time stamp at which we started predicting future events. The second value we need is the total number of evaluated predictions, $PC = (\sum_{t\geq t_s} PC_t) - CP_{t_e}$, where t_e is the current time stamp, and CP_{t_e} the set of currently valid predictions, for which we still do not know whether they will be proved correct or incorrect, and we therefore disregard them from the computations. Finally, we need to compute how many correct predictions we have made, $CPC = \sum_{t\geq t_s} CPC_t$. Once we have computed these values, we can compute precision and recall along the lines described above,

$$precision_{t_e} = \frac{CPC}{PC} \quad \text{and} \quad recall_{t_e} = \frac{CPC}{EC}.$$

Example 4. *Consider an incoming stream $ES = aacab...$, and assume we are using a sliding window of length 4, and a prediction span of size 3. Further assume we have discovered two prediction rules, namely $a \Rightarrow b$, and $c \Rightarrow d$. Table 6 gives an overview of the evaluation process introduced in Section 5.3 as the stream progresses. At time 1, event a occurs, and we predict that event b will occur in the coming prediction interval of size 3, i.e., at time stamp 2, 3 or 4. We increase the event count and the prediction count by 1. At time 2, another a occurs. We once again predict that b will occur soon ($PM_2 = b$), but, since b is already in the set of currently valid predictions, we do not need to add another b, and, therefore, $NP_2 = \emptyset$. As a result, we only increase the event count by 1, while the prediction count for time 2 remains 0. At time 3, event c occurs, and we predict that event d will occur in time interval $[4, 6]$. We increase both the event count and the prediction count. At time 4, another a occurs. At this point, we know that the prediction we made at time 1 has been proved wrong. At that point, we predicted that event b would occur within three time units, and this prediction has now expired ($EP_4 = b$). The set of existing valid predictions (PP_4) now contains only d. However, since another a occurred, we once again make a prediction that b will happen, this time within time interval $[5, 7]$. Therefore, $NP_4 = b$, and the set of currently valid predictions now contains b and d. We once again increase both the event count and the prediction count. Finally, at time 5, event b occurs. Since b was in CP_4, we correctly predicted this occurrence. We increase both the event count and the correct prediction count, and remove b from the set of currently valid predictions.*

Table 6: An illustration of our evaluation framework

t	i	EP_t	PP_t	PM_t	NP_t	CP_t	EC	PC_t	CPC
1	a	-	-	b	b	b	1	1	0
2	a	-	b	b	-	b	2	0	0
3	c	-	b	d	d	bd	3	1	0
4	a	b	d	b	b	bd	4	1	0
5	b	-	d	-	-	d	5	0	1

Assume now that we want to evaluate the performance of our predictor at this point. We can see that five events have occurred, of which we only predicted one. Therefore, recall is computed to be equal to $1/5 = 0.2$. We can also see that we have made three predictions, one of which has come true. However, for one of those predictions, namely that d would occur within interval $[4, 6]$, we do not know whether it will prove correct or incorrect. We therefore have to leave this prediction out of our computations. Hence, we compute the precision to be equal to $1/2 = 0.5$.

5.5. Prediction Scoring Policies

In this set of experiments, we propose and evaluate five different scoring policies that can be used for making predictions. Our goal is to be able to, at each moment in time, generate reliable predictions as to which events can be expected to occur in the near future. To do this, we first check which of the discovered patterns apply. At time t, we select only those patterns whose prefix can be found in the recent past, but whose last element has not occurred since the last occurrence of the prefix. Once we have all such patterns, we can predict that the last element of each of these patterns will occur in the near future. However, at each time stamp we may only predict k events, so we need to rank the applicable patterns using a scoring function. The five proposed scoring policies are defined as follows:

(1) Support policy: $sup(p) = p.sumF$

The Support policy simply says that a prediction is most reliable if it is based on the most frequent pattern.

(2) Confidence policy: $conf(p) = p.sumF/\bar{p}.sumF$

The Confidence policy ranks the patterns according to their confidence. However, neither the Support nor the Confidence policy take the prediction context into account.

(3) Match policy: $match(p) = \frac{p.size}{p.minMatch} \times conf(p)$

The Match policy reduces the value of a prediction if the events that caused the prediction have occurred further in the past. For example, if we are predicting that event b will occur soon based on pattern $\langle a, b\rangle$, this prediction will score higher if a had just occurred, than if a had occurred, say, 10 time stamps ago, even if a is still within the relevance span.

(4) Fit policy: $fit(p) =$

$$\begin{cases} conf(p) & \text{if } p.minMatch \leq p.avgLen \leq p.maxMatch \\ \frac{p.avgLen}{p.minMatch} \times conf(p) & \text{if } p.avgLen < p.minMatch \\ \frac{p.maxMatch}{p.avgLen} \times conf(p) & \text{if } p.avgLen > p.maxMatch \end{cases}$$

Note that the Match policy can sometimes predict an event too early. Assume that the average length of a minimal occurrence of pattern $\langle a, b\rangle$ is 5, then it might not be a good idea to predict an occurrence of b just after event a had occurred. The Fit policy assigns a higher value to those patterns whose average occurrence length falls within the interval containing the possible minimal occurrence lengths of the current occurrence of the pattern, assuming that the predicted event occurs within the prediction span. Above, $p.maxMatch = p.minMatch + span - 1$ and $p.avgLen = p.wSum/p.wCount$.

(5) Combination policy: $comb(p) =$

$$\begin{cases} match(p) & \text{if } p.minMatch \leq p.avgLen \leq p.maxMatch \\ \frac{p.avgLen}{p.minMatch} \times match(p) & \text{if } p.avgLen < p.minMatch \\ \frac{p.maxMatch}{p.avgLen} \times match(p) & \text{if } p.avgLen > p.maxMatch \end{cases}$$

Finally, since policies (3) and (4) take different insights from the prediction context into account, we test whether they can also reinforce each other by combining them into a unified Combination policy.

Figure 3 shows the performance of different scoring policies, for varying values of k. In our experiments, we use the first 100 batches solely to mine patterns, and we start making predictions from the 101th batch, and continue all the way to the 200th batch (naturally, we still update the pattern list after each batch). We set *maxNum* to 50 000, *minConf* to 0.3 and *minSup* to 0 (meaning that we just use the current error bound ϵ_i as the minimum support threshold for the next batch which we want to predict) for all of the datasets. Additionally, we set batch size and the sliding window length to 2 000 and 10, respectively, for the three dense datasets, and to 86 400 (one day) and 120, respectively, for the *ALARM* dataset. Finally, in the *ALARM* dataset, we also set the optional *max_size* parameter to 5, in order to limit the size of the discovered patterns. We do this because in some dense batches, the patterns could grow very large, resulting in an unnecessary growth of the error bound, and a lot of useful (short) patterns being pruned. At the same time, long patterns are rarely matched in the prediction task, and are therefore less useful in this context. As can be seen in Figure 3, the Combination policy consistently showed the best performance, although Match was always only narrowly second. Therefore, we used the Combination policy in our remaining analyses in the coming sections.

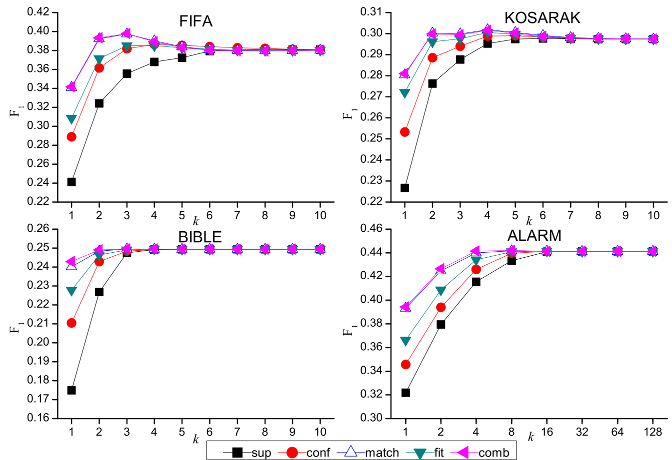

Figure 3: Performance of the different scoring policies

Taking into account the distribution of items in the four datasets, as shown in Table 4, we see that our predictor does a lot better than if we simply predicted the most frequent k items at each time stamp, which demonstrates the usefulness of the found patterns. Both our pattern miner and the predictor achieved satisfying runtimes. Table 7 shows the average runtime of our pattern miner to process a batch and the average runtime of the predictor to make a prediction, where the parameters are fixed at values defined in Section 5.5. To mention just two examples, it took on average 1.45 seconds to process a full day of incoming data in the *ALARM* dataset, while in the *BIBLE* dataset it took on average 29.78 *ms* to generate a single prediction.

Table 7: Runtime of mining a batch and making a prediction

Dataset	Mining a batch	Making a prediction
FIFA	1.34 s	26.96 ms
KOSARAK	1.27 s	24.41 ms
BIBLE	1.83 s	29.78 ms
ALARM	1.45 s	5.08 ms

5.6. Parameter Analysis for the Predictor

To further explore the performance of the predictor using the Combination policy, we conducted an analysis of the effect of varying parameter values on the precision and recall on the four datasets. The k parameter is set to 3, 4, 4 and 8 for *FIFA*, *KOSARAK*, *BIBLE* and *ALARM*, respectively, since these values of k produced the best predictor results, as shown in Figure 1. The remaining parameters are fixed at values defined in Section 5.5, except the parameter we varied in a given experiment.

5.6.1. Support threshold

We first experiment with different support thresholds (*minSup*). As shown in Figure 4, the precision typically improves

with increasing support thresholds while the recall decreases since there are fewer matched patterns with a higher support threshold. However, the quality of the matched patterns is better as a result.

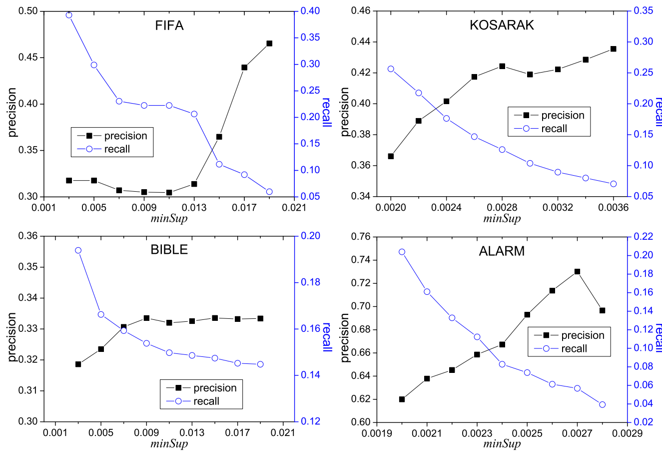

Figure 4: The impact of different support thresholds on precision and recall

5.6.2. *Confidence threshold*

Figure 5 shows the performance of our predictor at different minimum confidence thresholds (*minConf*). From Figure 5, we can see that the performance of the predictor is strongly related to the confidence threshold, as the precision increases and the recall decreases when the confidence threshold is raised.

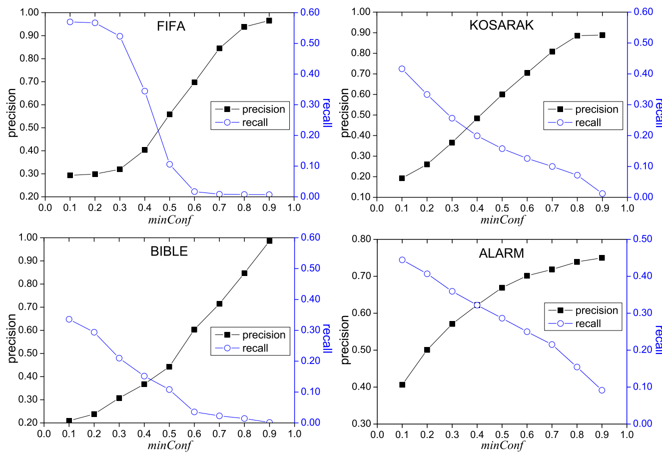

Figure 5: The impact of different confidence thresholds

5.6.3. *Number of patterns kept in memory*

Figure 6 shows the performance of our predictor with a varying maximum number of patterns kept in memory (*maxNum*). As can be seen, the performance of the predictor is quite stable with different maximum number of patterns for prediction. However, the precision decreases a little with ascending *maxNum* while the recall grows since there are more matched patterns when more patterns are kept in memory.

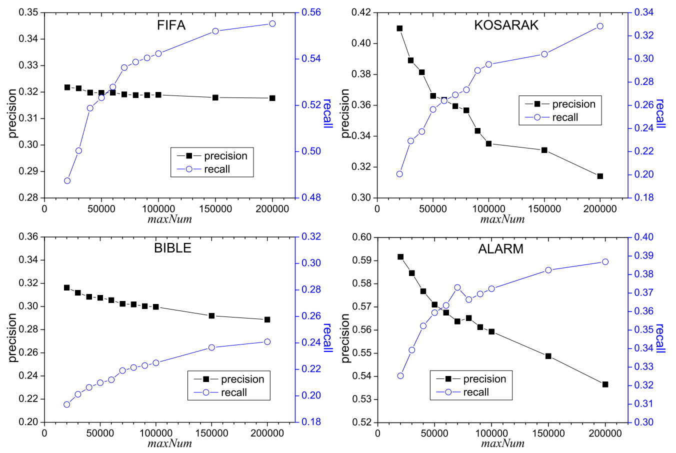

Figure 6: The impact of different maximum numbers of kept patterns

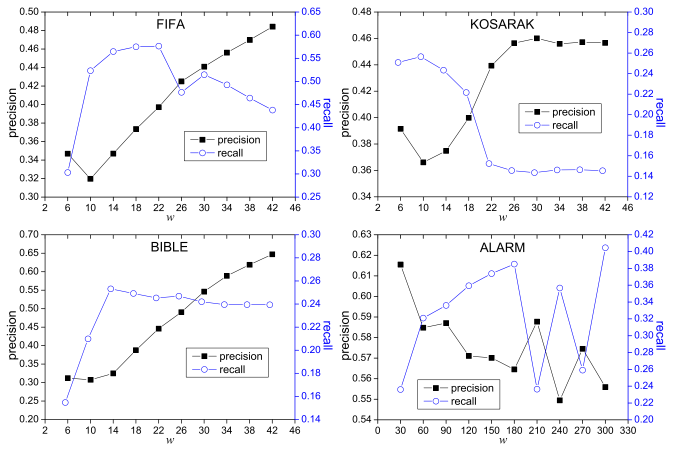

Figure 7: The impact of different sliding window lengths

5.6.4. *Sliding window length*

Figure 7 shows the precision and recall of the predictor under various settings of sliding window length w where the prediction span is set to $\lfloor \frac{w}{2} \rfloor$. From the figure we can see that the precision and recall often vary a lot with the changes of the sliding window length. We conclude that the window size should be set sensibly depending on the dataset, as neither increasing nor decreasing it guarantees better results.

5.6.5. *Prediction span*

Figure 8 shows the performance of our predictor under various settings of the prediction span where the sliding window lengths are 30, 30, 30 and 120 for *FIFA*, *KOSARAK*, *BIBLE* and *ALARM* respectively. We can see that increasing the prediction span, as expected, resulted in higher precision and recall, as the predictions remained valid for longer.

5.6.6. *SPADE Memory Limit*

Figure 9 shows the performance of our predictor under different memory limits of SPADE, which is used when mining sequential patterns from one batch of data. We conclude that while this parameter can result in a lower false positive rate

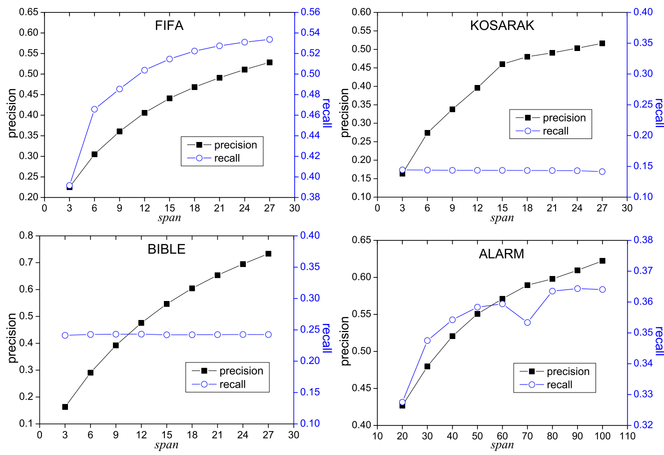

Figure 8: The impact of different prediction spans

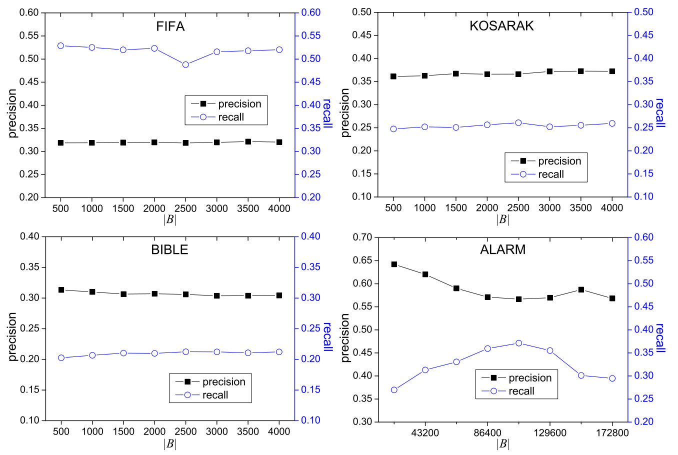

Figure 10: The impact of different batch sizes

when mining patterns, it has very little effect on the prediction performance.

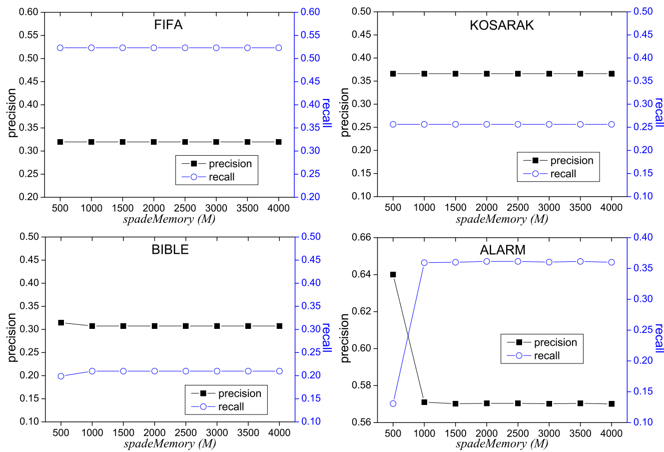

Figure 9: The impact of different memory limits used for SPADE

5.6.7. Batch Size

Figure 10 shows the performance of our predictor under different batch sizes. We find that changing the batch size had virtually no effect on either precision or recall, as long as the data is uniformly dense. If not, a very dense batch could result in a large error bound, and a loss of patterns, which would negatively affect the performance of the predictor, as was the case in the *ALARM* dataset.

6. Conclusions

Mining continuously massive data streams to discover up-to-date patterns is valuable for timely strategic decisions. This calls for the design of new mining methods to replace the traditional ones, since those would require the data to be first stored and then processed off-line using complex algorithms that make several passes over the data.

In this paper, we introduce a sequential pattern based event prediction framework for streaming data, which consists of a frequent sequential pattern miner and an event predictor. Due to the nature of stream mining, we propose an on-line sequential pattern miner based on Lossy Counting to dynamically determine the optimal error bound by maximising the memory usage. Through experimental evaluation, we show that our sequential pattern miner results in a lower false positive rate than existing methods. Furthermore, we use the discovered patterns to predict future events, and the extensive experimental results show that the predictor works well since we considered new quality measures for patterns to improve the predictor.

The proposed event prediction framework can be used to make a system that has the ability to learn patterns from event streams, to get new probabilistic associations over time, and to do real-time monitoring continuously on the forthcoming concerning events. We believe that the proposed framework highlights the directions in building real-time alerting services that predict significant events of interest.

There are still some limitations of our work. First, there are several user-chosen parameters for our predictor. However, tuning those parameters will increase the burden of a user. Second, we do not assign different importance to patterns based on how long before the prediction time they occurred in the historical data. However, it might be the case that a pattern that has not occurred often recently does not have a large predicting power.

Due to the above limitations, in future work, we will first attempt to reduce the number of user-chosen parameters that are currently needed by our predictor. Then, we will investigate how to build a new model that would give more value to patterns that occurred often in the recent past than to those that mostly occurred long ago when predicting future events based on them. Besides, we intend to explore ways to optimise memory usage further, in order to be able to achieve even higher accuracy. Additionally, we hope to be able to improve the data structure for storing the generated sequential patterns, by using, for example, a lexicographical tree.

Acknowledgements

Cheng Zhou is financially supported by the China Scholarship Council (CSC).

References

Arasu, A., Manku, G. S., 2004. Approximate counts and quantiles over sliding windows. In: Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. ACM, pp. 286–296.

Bala, A., Chana, I., 2015. Intelligent failure prediction models for scientific workflows. Expert Systems with Applications 42 (3), 980–989.

Casanova, I. J., Campos, M., Juarez, J. M., Fernandez-Fernandez-Arroyo, A., Lorente, J. A., 2015. Using multivariate sequential patterns to improve survival prediction in intensive care burn unit. In: Artificial Intelligence in Medicine. Springer, pp. 277–286.

Chen, G., Wu, X., Zhu, X., 2005. Sequential pattern mining in multiple streams. In: Fifth IEEE International Conference on Data Mining. IEEE, pp. 4–7.

Cheng, J., Ke, Y., Ng, W., 2008. A survey on algorithms for mining frequent itemsets over data streams. Knowledge and Information Systems 16 (1), 1–27.

Cho, C.-W., Wu, Y.-H., Yen, S.-J., Zheng, Y., Chen, A. L., 2011. On-line rule matching for event prediction. The VLDB Journal 20 (3), 303–334.

Chrobok, R., Kaumann, O., Wahle, J., Schreckenberg, M., 2004. Different methods of traffic forecast based on real data. European Journal of Operational Research 155 (3), 558–568.

Dimitropoulos, X., Hurley, P., Kind, A., 2008. Probabilistic lossy counting: an efficient algorithm for finding heavy hitters. ACM SIGCOMM Computer Communication Review 38 (1), 5–5.

Ezeife, C., Monwar, M., 2007. Ssm: a frequent sequential data stream patterns miner. In: IEEE Symposium on Computational Intelligence and Data Mining. IEEE, pp. 120–126.

Garofalakis, M., Gehrke, J., Rastogi, R., 2002. Querying and mining data streams: you only get one look a tutorial. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data. p. 635.

Gomariz, A., Campos, M., Marin, R., Goethals, B., 2013. Clasp: An efficient algorithm for mining frequent closed sequences. In: Advances in Knowledge Discovery and Data Mining. Springer, pp. 50–61.

Gu, X., Papadimitriou, S., Yu, P. S., Chang, S.-P., 2008. Online failure forecast for fault-tolerant data stream processing. In: IEEE 24th International Conference on Data Engineering. IEEE, pp. 1388–1390.

Gündüz, Ş., Özsu, M. T., 2003. A web page prediction model based on click-stream tree representation of user behavior. In: Proceedings of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, pp. 535–540.

Jalali, M., Mustapha, N., Sulaiman, M. N., Mamat, A., 2010. Webpum: A web-based recommendation system to predict user future movements. Expert Systems with Applications 37 (9), 6201–6212.

Koper, A., Nguyen, H. S., 2011. Sequential pattern mining from stream data. In: Advanced Data Mining and Applications. Springer, pp. 278–291.

Laxman, S., Tankasali, V., White, R. W., 2008. Stream prediction using a generative model based on frequent episodes in event sequences. In: Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, pp. 453–461.

Manku, G. S., Motwani, R., 2002. Approximate frequency counts over data streams. In: Proceedings of the 28th international conference on Very Large Data Bases. VLDB Endowment, pp. 346–357.

Mannila, H., Toivonen, H., Verkamo, A. I., 1997. Discovery of frequent episodes in event sequences. Data Mining and Knowledge Discovery 1 (3), 259–289.

Martin, F., Méger, N., Galichet, S., Becourt, N., 2012. Forecasting failures in a data stream context application to vacuum pumping system prognosis. Transactions on Machine Learning and Data Mining 5 (2), 87–116.

Mendes, L. F., Ding, B., Han, J., 2008. Stream sequential pattern mining with precise error bounds. In: Eighth IEEE International Conference on Data Mining. IEEE, pp. 941–946.

Merah, A. F., Samarah, S., Boukerche, A., Mammeri, A., 2013. A sequential patterns data mining approach towards vehicular route prediction in vanets. Mobile Networks and Applications 18 (6), 788–802.

Metwally, A., Agrawal, D., El Abbadi, A., 2005. Efficient computation of frequent and top-k elements in data streams. In: Database Theory-ICDT. Springer, pp. 398–412.

Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., Hsu, M.-C., 2004. Mining sequential patterns by pattern-growth: The prefixspan approach. IEEE Transactions on Knowledge and Data Engineering 16 (11), 1424–1440.

Raıssi, C., Poncelet, P., Teisseire, M., 2005. Need for speed: Mining sequential patterns in data streams. BDA05: Actes des 21iemes Journees Bases de Donnees Avancees.

Tanbeer, S. K., Ahmed, C. F., Jeong, B.-S., Lee, Y.-K., 2008. Efficient frequent pattern mining over data streams. In: Proceedings of the 17th ACM conference on Information and knowledge management. ACM, pp. 1447–1448.

Wang, Y., Ma, E. W., Chow, T. W., Tsui, K., 2014. A two-step parametric method for failure prediction in hard disk drives. IEEE Transactions on Industrial Informatics, 419–430.

Zaki, M. J., 2001. Spade: An efficient algorithm for mining frequent sequences. Machine learning 42 (1-2), 31–60.

Zhu, H., Wang, P., Wang, W., Shi, B., 2011. Stream prediction using representative episode rules. In: IEEE 11th International Conference on Data Mining Workshops. IEEE, pp. 307–314.