BFSPMiner: an effective and efficient batch-free algorithm for mining sequential patterns over data streams

Marwan Hassani¹ · Daniel Töws² · Alfredo Cuzzocrea³ · Thomas Seidl⁴

Received: 1 June 2017 / Accepted: 6 December 2017 / Published online: 26 December 2017
© The Author(s) 2017. This article is an open access publication

Abstract

Supporting sequential pattern mining from data streams is nowadays a relevant problem in the area of data stream mining research. Actual proposals available in the literature are based on the well-known *PrefixSpan* approach and are, indeed, able to effectively bound the error of discovered patterns. This approach foresees the idea of dividing the target stream in a collection of manageable *chunks*, i.e., pieces of stream, in order to gain into effectiveness and efficiency. Unfortunately, mining patterns from stream chunks indeed introduce additional errors with respect to the basic application scenario where the target stream is mined continuously, in a non-batch manner. This is due to several reasons. First, since batches are processed individually, patterns that contain items from two consecutive batches are lost. Secondly, in most batch-based approaches, the decision about the frequency of a pattern is done locally inside a single batch. Thus, if a pattern is frequent in the stream but its items are scattered over different batches, it will be continuously pruned out and will never become frequent due to the algorithm's lack of the “complete-picture” perspective. In order to address so-delineated pattern mining problems, this paper introduces and experimentally assesses *BFSPMiner*, a Batch-Free Sequential Pattern Miner algorithm for effectively and efficiently mining patterns in streams without being constrained to the traditional batch-based processing. This allows us, for instance, to discover frequent patterns that would be lost according to alternative batch-based stream mining processing models. We complement our analytical contributions by means of a comprehensive experimental campaign of *BFSPMiner* against real-world data stream sets and in comparison with current batch-based stream sequential pattern mining algorithms.

Keywords Sequential pattern mining · Data streams · Batch-free

1 Introduction

Among the emerging data mining research areas, sequential pattern mining [2,27,31] is gaining the momentum and it plays a leading role in next-generation big data applications where order-sensitive patterns are mined, extracted and successfully used to derive “actionable knowledge”. The most distinctive characteristic of sequential pattern mining consists in the presence of time in the observed items, as in contrast with traditional frequent pattern mining problems. Indeed, in traditional frequent pattern mining, the main activities are: (i) finding all item subsets that occur *together* in the target transaction database; (ii) discovering all mining rules that describe the correlation of the *presence* of an item subset *with* the presence of another item subset in the target transaction database. Therefore, this approach has traditionally turned to be useful in supporting pattern mining in a wide family of application scenarios, ranging from basket data analysis to cross-marketing, from catalog design to loss

✉ Marwan Hassani
m.hassani@tue.nl

Daniel Töws
daniel.toews@fkie.fraunhofer.de

Alfredo Cuzzocrea
alfredo.cuzzocrea@dia.units.it

Thomas Seidl
seidl@dbis.fwi.lmu.de

¹ Architecture of Information Systems Group, Eindhoven University of Technology, Eindhoven, The Netherlands

² Fraunhofer Institute for Communication, Information Processing and Ergonomics FKIE, Wachtberg, Germany

³ DIA Department, University of Trieste and ICAR-CNR, Trieste, Italy

⁴ Database Systems Group, LMU Munich, Munich, Germany

leader analysis, and so forth. On the other hand, sequential pattern mining finds all item subsets that occur frequently in a specific *sequence* of the target transaction database, and discovers all mining rules that describe the correlation of the *order* of an item subset *after* the order of another item subset in the target transaction database. It thus follows that the order of items plays a critical role in such problems, and, in addition to this, not only the combinations of items but even all the possible permutations of items discriminate the sequence's frequency, hence determining if a target item sequence is frequent or infrequent. Therefore, this approach has been successfully applied to innovative data mining application scenarios, ranging from bioinformatics [1] to web mining [7], from text mining [32] to sensor data mining [4,5,16,28], and so forth.

In the related *static* sequential pattern mining context, *PrefixSpan* [27] has been proposed and elected as one of the state-of-the-art solutions to the investigated problem. *PrefixSpan* is indeed able to effectively mine the entire pattern set, and it also adds efficiency by significantly reducing the computational effort due to candidate subsequence generation. As a consequence, several stream-based algorithms [9,16,23,26,29,33] propose to re-adapt the *PrefixSpan*'s idea for targeting sequential pattern mining problems in the streaming case. Basically, these algorithms argue to: (i) divide the target stream into batches; (ii) collect suitable statistics on these batches that can also be updated in an effortless mode (e.g., via ad-hoc summary data structures); (iii) finally apply *PrefixSpan* over these statistics (see Sect. 2).

Unfortunately, mining patterns from stream chunks introduce additional errors with respect to the basic application scenario where the target stream is mined continuously, in a non-batch manner. This is due to several reasons. First, since batches are processed individually, patterns that contain items from two consecutive batches are lost. Secondly, in most batch-based approaches, the decision about the frequency of a pattern is done locally inside a single batch. Thus, if a pattern is frequent in the stream but its items are scattered over different batches, it will be continuously pruned out and will never become frequent due to the algorithm's lack of the "complete-picture" perspective.

Consider the example in Fig. 1. Here, an application of a translation session from a source text to a target text [11] is addressed. The eye gazes and the keystrokes of users are collected during the whole translation session. Texts are carefully selected, such that they contain problematic translation tasks according to the source and target languages. Interesting behavioral combinations of eye gazes and keystrokes might be indicators of cognitive processes of the translator, which is an interesting research direction for psycholinguists [3]. Dealing with the input data as streams allows to observe the evolution of certain rules over the translation session, in order to track, for instance, the effect of learning from

Fig. 1 An example of an application for mining sequential patterns over multiple streams. Translators perform eye gazes over particular locations in source and target texts. Keystrokes of translators are also collected. Both inputs are separate streams, providing sequential data. Certain combinations (patterns) of both streams constitute cognitive processes, which can be found by mining sequential patterns

mistakes. When applying any of the above-mentioned batch-based stream approaches, some sequence of items, like a long eye gaze followed nine DELETE keystrokes and then a *verb* typed, might be distributed over two consecutive batches, leaving the algorithm unable to discover the resulting pattern. Additionally, when focusing on the “frequent” patterns generated within a single batch merely, some patterns might be continuously pruned because they are locally non-frequent, although they would be frequent when considering the bigger picture.

In order to address so-delineated problems, this paper introduces and experimentally assesses *BFSPMiner*, a Batch-Free Sequential Pattern Miner algorithm for effectively and efficiently mining patterns in streams without being constrained to the batch-based processing. This allows us, for instance, to discover frequent patterns that would be lost according to alternative batch-based stream mining processing models. The algorithm builds upon the T_0 tree structure [26] that addresses the above-mentioned issues and efficiently overcomes the scalability issues that could appear from using a batch-free stream management method. We introduce a novel pruning approach to safely eliminate non-promising patterns from the candidate tree.

We complement our analytical contributions by means of a comprehensive experimental campaign of *BFSPMiner* against real-world data stream sets and in comparison with current batch-based stream sequential pattern mining algorithms. In more details, we show through a list of extensive evaluations over seven real-world data sets the superiority of *BFSPMiner* w.r.t. the accuracy and the completeness of the discovered patterns. To highlight the importance of a batch-free streaming sequential pattern mining, we also provide a novel prediction system that uses the found patterns as rules for predicting near-future items. In order to evaluate the prediction part, we consider two additional real-world data sets and we show again the high predictability of the

patterns retrieved by *BFSPMiner* when compared with those extracted by state-of-the-art batch-based algorithms.

This paper significantly extends the short paper [21]. With respect to the early version, we provide the following contributions:

- we provide much more conceptual content, motivations and description of *BFSPMiner*, by also proposing significant examples and case studies;
- we add a detailed explanation of our approach by extending the pseudo code with two additional algorithms and by elaborating the third one;
- we improve and extend related work analysis;
- we provide a more rich and comprehensive experimental assessment and analysis of *BFSPMiner*, also considering real-world data stream sets.

The remaining part of the paper is organized as follows. Section 2 focuses on related work and its correlation with the *BFSPMiner* proposal. In Sect. 3, we discuss some important preliminaries that are needed to introduce our approach. Section 4 introduces the innovative *BFSPMiner* algorithm, with details on the different components and modules. In Sect. 5, we provide the experimental assessment and analysis of *BFSPMiner* against seven real-world data sets and in comparison with state-of-the-art algorithms. Finally, Sect. 6 contains conclusions and future work of our research.

2 Related work

The problem of finding sequential patterns in a static database has been introduced by Agrawal and Srikant over 20 years ago [2]. During this time, different algorithms providing feasible solutions to this annoying problem have been proposed by the research community. Notable algorithms, which also impacted sequential pattern mining in data streams, are: *SPADE* [31], and *PrefixSpan* [27].

These algorithms have also been applied to the leading data stream context, even though they were designed to target static databases. This approach argues to collect multiple items in a batch, in order to improve the stream processing phase. These small batches are then treated as individual static data sets, on top of which *SPADE* and *PrefixSpan* are applied in order to find sequential patterns. *SS-BE* [26] is a popular algorithm that makes use of this approach, along with several other alternatives [23,29,33].

Despite this main framework, mining sequential pattern in data streams introduces complex problems that cannot be solved by analyzing batches separately, and then adding-up intermediate results. First, if an item appears as new in the target data stream, the item and patterns containing it will not be classified as frequent, as they did not occur enough. It

would be a naive strategy to dismiss items like this, as they could re-appear more often in the future. Second, patterns that contain items of two consecutive batches cannot be found, since the batches are analyzed independently.

SS-BE algorithm tries to solve the first issue via introducing different so-called *support thresholds*. This conveys in the fact that new patterns, which are potentially promising, get the necessary time to become frequent. If this is not the case, these patterns will be discarded from the final result set. Nevertheless, this still does not solve the investigated issue, as there may be patterns that do not appear to be promising in one batch, but are indeed frequent over the whole stream.

Another suggestion to solve this problem has been proposed by Koper and Nguyen [23]. Their proposal, called *SS-LC*, makes use of an adapted version of *PrefixSpan*. Here, items that are not frequent in a batch are still considered if they are frequent with respect to the *entire* stream. This seems to solve the problem but, unfortunately, the proposed adaptation does not consider patterns properly. Therefore, frequent sequential patterns will not be displayed completely accurate, because batches where these patterns are not classified as frequent still remain.

SPEP [33] is another approach that aims at minimizing the error that occurs when between-batches information is lost. To this end, authors look at a snapshot of data that are close to the borders of two batches and collect new sequential patterns there. This would limit the sequential pattern problem difficulties of comparison approaches.

In order to deal with the described challenges, we introduce *Batch-Free Sequential Pattern Miner* algorithm, *BFSPMiner* in short. *BFSPMiner* avoids those issues via analyzing the target data stream in a continuous, *batch-free* way. In addition to this, we make use of the generated sequential patterns to predict new upcoming items in the data stream. With respect to the target application (i.e., language translational), for instance, *BFSPMiner* mainly looks into data that are containing eye gazes on the reference computer screen, with the goal of using eye gazes to predict the next action of the user. Predicting user behavior is an important research task that has been considered in stream data mining problems for over a decade [8], even in different-but-related contests such as prediction of system failures [25]. This wide range of real-life applications further demonstrates the benefits that derives from our proposed *BFSPMiner* algorithm.

3 Preliminaries: sequential pattern mining

In this Section, we provide preliminary concepts and definitions that are needed to introduce the *BFSPMiner*'s approach.

Following the canonical definition by Agrawal and Srikant [2], sequential patterns are defined as frequent sequences that are contained by the target database. Formally, given

sequences that contain items, in contrast to sets, a sequence $A = (a_1, a_2, \dots, a_n)$ contains a sequence $B = (b_1, b_2, \dots, b_m)$, if there exists an ordering with integers $i_1 < i_2 < \dots < i_m$, such that $b_1 = a_{i_1}, b_2 = a_{i_2}, \dots, b_m = a_{i_m}$.

A data stream $S = s_1, s_2, \dots$ is defined as an infinite set of items, where each item s_i is characterized by an arrival time t_i . In the data stream context, a sequence $p = (p_1, p_2, \dots, p_n)$ is contained by the target stream, or a subsequence of the target stream, if there exists an ordering with integers $i_1 < i_2 < \dots < i_n$, such that $p_1 = s_{i_1}, p_2 = s_{i_2}, \dots, p_n = s_{i_n}$, being $p_1, p_2, \dots, p_n$ defined as *pattern* and p_i , as *pattern item*.

Given a data stream S and support threshold x , such that $0 \leq x \leq 1$, a subsequence p is classified as frequent, and thus as a sequential pattern, if the following condition holds:

$$\frac{\text{count}_p}{n} \geq x \quad (1)$$

wherein (i) count_p is the number of occurrences of p in S ; (ii) n is the number of items that already appeared in S .

Based on (1), a subsequence is classified as sequential patterns if it contains (frequent) items that appear one after each other, without the constraints of these items to be *consecutive* one after each other. However, since we focus on the specific data stream setting where the order of items is, by the contrary, relevant, we will slightly adapt the definition (1). To increase the meaningfulness of discovered patterns, we superimpose that only subsequences where items are consecutively following each other can be classified as sequential patterns. For this reason, we introduce an additional constraint to the ordering of sequential patterns. Thus, the subsequence p is considered as sequential pattern, if it exists an ordering $i_1 < i_2 < \dots < i_n$, such that:

$$\begin{cases} p_1 = s_{i_1}, & p_2 = s_{i_2}, \dots, p_n = s_{i_n} \\ \forall k \in \{1, 2, \dots, n\}, l = k + 1 \rightarrow i_l = i_k + 1 \end{cases} \quad (2)$$

Efficient in-memory representation of so-defined patterns requires a specific data structure. The T_0 tree is a popular structure introduced by Mendes et al. [26], which is used by *SS-BE* as well as *SS-LC* algorithm. Such a tree contains information about all sequential patterns, as well as promising candidates that may become sequential patterns in the future. In more details, the root of T_0 contains the number of items that already appeared in the data set. Each other node in T_0 represents a pattern p and contains its corresponding information. This includes, for instance, the number of occurrences, the time it first appeared, and the item's name. The pattern that is represented by a specific node can be determined by following the path from the root to the node and retrieving the names of items that are contained in each node along the path. Figure 2 shows a T_0 tree example.

Fig. 2 An example of the T_0 tree. The dotted node represents the pattern c, a and contains the information of its occurrences

In order to predict new items in the target data stream, rules that exploit the context of the latest items are necessary. Those patterns, if frequent, suggest that, if the first few items appeared, then the next few items will appear with a certain confidence. *BFSPMiner* makes use of an approach similar to the one used by *SPEP* algorithm for assessing the quality of prediction. Prediction rules are indeed not limited to sequential pattern mining problems only. More complex structures like *episode rules* can also be adopted by algorithms to predict items, with the main goal of achieving higher accuracy thanks to more advanced rules [6,34]. The complex nature of those rules results in higher computational overhead of target sequential pattern mining algorithms.

4 The BFSPMiner

In Fig. 3, the necessary steps of the algorithm are modeled. In this work, we treat the preprocessing as a black box that provides the algorithm with the important information of the data stream. For creating sequential patterns, *BFSPMiner* requires only an item label and time stamp. After receiving the newest item, the algorithm will determine new patterns that are sequential pattern candidates. Those patterns will be saved in an adapted version of the T_0 tree. After the algorithm determined the current sequential patterns, these will be passed to the predictor, which uses them to predict the next upcoming item. As a last step, the algorithm eventually prunes non-promising patterns from the tree. Notable is that each of those components can be swapped out with other subroutines, especially the predictor and the pruning are optional components which can be left out depending on the requirements of the application.

Fig. 3 A rough sketch of *BFSPMiner* **Algorithm 1:** The whole process of our evaluation framework

```

Data: item
/* the new incoming data from the stream */
Result: Updated sequential patterns and predictions
/* Each item will be evaluated separately */
1 for incoming item do
2   itemPreProcessed = preprocessing(item) ItemList[Item]
   list.remove(oldest) ItemList[Item]
   list.add(itemPreProcessed) /* This method is
   explained in Algorithm 2 */
3   List[Pattern] patterns = useBFSPMiner(list, item) /* This
   is explained in Algorithm 3 */
4   List[Item] predictions = predict(patterns, itemPreProcessed,
   list, k) print(predictions)
  
```

4.1 Creating sequential patterns

We create patterns in a continuous way, which corresponds to the natural flow of data streams. Therefore, we analyze each item directly after it appears in the stream. All patterns that are created at this step will then have this new item as its last part.

Given an item a as the newly arrived item, the *BFSPMiner* starts with this item as a pattern of length one. In the next step the predecessor of the new item, will be attached in front of the starting item. This step will be repeated recursively. For example, the last items that appeared in the stream were c and then b . Now the new item a will be processed. First the pattern a will be created, then b, a and then c, b, a . As the future of those patterns is not clear, they will all be saved in our adapted T_0 tree.

The complete process of our proposed framework is explained with pseudo code in Algorithm 1. The pattern creation part is detailed in Algorithm 2 and the prediction part is described in Algorithm 3. An open source implementation of *BFSPMiner* is available under [30].

4.2 Maximum pattern length as a solution for the quadratic growth of patterns

The support of every pattern changes with every new arriving item. To ensure that all sequential patterns are output at user request, every possible pattern and its information have to be saved. However, if all possible patterns are regarded and

Algorithm 2: Creating patterns with the *BFSPMiner*

```

Data: ItemList, currentPattern
/* ItemList contains the latest compressed
items and is limited by
maxPatternLength. The newly arrived item
is at the last position */
Result: The new patterns that can be created with the new item
1 int index = ItemList.length;
2 list patterns = new list;
/* create patterns until maxPatternLength
is reached */
3 while currentPattern.length ≤ ItemList.length do
/* add the next item to the pattern */
4   currentPattern = ItemList.get(index-currentPattern.length) +
   currentPattern;
5   patterns.add(currentPattern);
/* update the tree with the new patterns */
6 updateTree(currentPattern);
7 if itemCount % pruningCount == 0 then
8   pruneTree()
9
  
```

saved, the amount of patterns increases quadratically and so do both the calculation time and memory consumption.

To stop this growth, *BFSPMiner* uses a parameter, the *maxPatternLength* parameter, to limit the length of patterns that are generated. A *maxPatternLength* of five will cause patterns to maximally contain five items. With such a set limit, the time required to generate patterns is constant and independent of the data stream and scales linearly with the amount of arriving data. The algorithm will find all sequential patterns that are smaller than *maxPatternLength*. Saving those found patterns in the tree structure will result in full accuracy for those patterns. However, this parameter will hinder *BFSPMiner* from finding sequential patterns that are longer than the set limit. In the experimental evaluation though, we will show that this is not a big concern in real-world data sets. We will argue in Figs. 9, 10 and 11 using two different real data sets that setting it in general to a specific value is maximizing the accuracy while spending a reasonable amount of time.

4.3 The inverted T_0 tree

The original T_0 tree provides an efficient data structure for saving sequential patterns. In this tree each node represents its own sequential pattern. We can find each pattern by following a predetermined path and adding the labels of the

nodes together. Additionally the support of the pattern is represented by the timestamps of when the patterns occurred. Looking at the left tree in Fig. 4, the node b beneath the node a will represent the pattern a, b . In this node all the timestamps of the occurrences of this pattern are saved. Additionally, if an output is requested and node t_1 is representing a non-frequent item the whole subtree can be skipped from the output. This is possible, since each node of the subtree rooted in t_1 represents a longer and therefore even rarer pattern than t_1 itself.

However, the original form of the tree is problematic to use for *BFSPMiner*, as it is a tree, where each node represents the prefix of its children. For example, if node t_2 is a child of node t_1 , then the pattern represented by the node t_1 is a prefix of the pattern represented by node t_2 . However, the patterns generated by *BFSPMiner* generated in a specific time step are postfixes of each other. Looking back at the example above, where item a was processed, it generated patterns a and b, a and c, b, a . Here the smaller patterns are postfixes and not prefixes of the longer patterns. This is due to the fact that the algorithm generates pattern with the newest item and adds in items that occurred earlier. Therefore, the algorithm faces a problem concerning the input of the new generated patterns into the T_0 tree. For each generated pattern, it has to search for a completely different path in the tree to insert the new patterns. Using the normal T_0 tree with *BFSPMiner* will result in a huge calculation overhead in each step of the tree update.

This problem can be solved by simply inverting the T_0 tree. This means that each node will now represent the pattern that can be determined by reading the path from the current node to the root, from bottom to the top. With this simple change, in each tree update step, the algorithm can now update the changed patterns along one path instead of updating multiple paths. Figure 4 visualizes the difference between the update procedure for the T_0 tree and the inverted T_0 tree. In this new structure the rightmost path represents the pattern a, b, d and its occurrences.

The importance of this small change is crucial when working with data streams that contain multiple unique items and long sequential patterns. If the stream contains ten different items, then each node may have up to ten different children. Finding a specific path of a long pattern is a time-consuming task. Using the inverted T_0 tree, the algorithm has to only find one path and can update the nodes along the way, instead of finding multiple paths and updating only one node in each of them.

4.4 Pruning non-promising patterns

Assuming that the data stream contains n unique items, the amount of patterns that have to be saved in the inverted T_0 tree is bounded by $n^{\max PatternLength}$. As the data stream is usually expected to contain an infinite amount of unique items, the amount of patterns to be saved will be infinitely

Fig. 4 After processing item d , and generating the new patterns with *BFSPMiner*, the algorithm has to update the data structure with the new generated patterns. When updating the T_0 tree, on the left, the algorithm has to update along three different paths. In the inverted T_0 tree, here on the right, the algorithm can update along only one path

many. Therefore, we are not able to save each and every pattern indefinitely and are forced to prune out non-frequent patterns eventually.

The *BFSPMiner* algorithm utilizes the pruning strategy that is also used by the *SS-BE* algorithm [26]. This pruning strategy contains following values and parameters: 1. **TID**: contains the timestamp of the batch that first contained this pattern. This information is saved in the specific node in the tree. 2. **B**: the number of batches passed since the last pruning, before the node was inserted in the tree. 3. **batchCount**: the number of batches that contain the pattern. This information is contained in the specific node in the tree. 4. α and ϵ : two different support thresholds that determine promising patterns. It holds that $0 \leq \alpha \leq \epsilon \leq supportthreshold$. 5. **L**: the batch size. it determines how many items will be part of one batch. 6. δ : the pruning period. After δ batches have passed, the pruning will be repeated.

Given those parameters, a node and its subtree will be pruned if $count + (B - batchCount) * (\alpha L - 1) \leq \epsilon * B * L$. A more detailed explanation of the parameters and a proof stating the completeness of this pruning strategy can be found in [26].

Since *BFSPMiner* does not use batches, the algorithm will simply keep track of the amount of items that have passed the stream and then assign a new batch number every L items. Additionally, the algorithm will repeat the pruning process every $L * \delta$ items.

4.5 Predicting new items

The list of frequent patterns and their count values are passed on to the predictor, where it can use those patterns as rules for

making new predictions. First the predictor searches through the list to find rules matching the context given by the latest items that appeared in the stream. For example, if a, b, c are currently the latest items in our data stream, the predictor searches through the list of frequent patterns to find the patterns c and b, c and a, b, c . Afterward the predictor searches through the list to find all patterns that match the context and contain one additional item. For example c, d and a, b, c, e would match this description, but not b, c, d, a . With the count value of the patterns that match the context and the ones with the additional item, we can determine the confidence of the association rule. The confidence of an association rule $A \rightarrow b$ is determined by $conf_{A,b} = \frac{count_{A,b}}{count_A}$. The confidence value tells us, how often the item b appeared, after we found the pattern A . If c appeared 5 times and c, d 3 times, the confidence of the association rule $c \rightarrow d$ would be $\frac{3}{5}$.

Algorithm 3: Predicting new Items using Sequential Pattern Mining

```

Data: patterns, item, itemList, k
/* patterns contains the current sequential
   patterns, item the newest item, itemList
   contains the last few items to determine
   the context and k the number of
   requested predictions. */
Result: List of predictions
/* Find the patterns, that are matching the
   context of our last few items */
1 List[Pattern] relevantPatterns = findRelevantPatterns(itemList,
   patterns)
2 List[Pattern] sorted = sortByConfidence(relevantPatterns)
3 return List[Pattern] predictions = topK(sorted, k)

```

After we determined the confidence of all rules that match our context, we predict the item that is part of the rule with the highest confidence. Looking at the example above, the predictor would predict d to be the next item, if this is the rule with the highest confidence. If we receive a completely new item, then the predictor cannot find any applicable rule and we predict the item with the highest support value.

Similar to the prediction done by Zhou et al. [33], we use two different parameters to relax the predictions. We use the parameter k , to predict up to k different items for the next time step. Additionally we use the parameter $span$ to determine how long we want to hold on to a prediction. In our past examples we wanted to predict one item for the next time step, which corresponds to the parameter setting $k = 1$ and $span = 1$. If we set our parameter $k = 2$, we use the two association rules with the highest confidence to predict that the next item will be one of the two. If we set $span = 3$, a prediction we make at time step t is not only correct for time step $t + 1$, but also for the time steps up to $t + 3$. If an item that was predicted at time step t is

predicted again during $span$ time steps, it will not be added to the list of pending predictions. This means each item can only be predicted once during $span$ time steps and has to be predicted again afterwards. This ensures that the amount of predicted items does not exceed the amount of items from the stream.

We introduce a different parameter, the *contextLength* parameter, which determines the length of the used rules. The parameter limits the amount of items appearing on the left side of the association rule. Intuitively, and also based on our experimental evaluation on multiple real data sets, we observed that patterns with the highest confidence values are usually smaller patterns. Because of this, the prediction quality does not increase after a certain context length. But a longer context means that more rules have to be checked, which results in a significantly higher runtime.

5 Experimental results

In the beginning of this section, we will take a deeper look in the prediction method by showing its attributes through an experimental evaluation. After this, we show the superior prediction quality of *BFSPMiner*, while maintaining a competitive runtime compared to current popular algorithms. For this we compare *BFSPMiner* against the *SS-BE* [26] and *SS-LC* [23] algorithms. For the first evaluation part, where we take a look at the predictor, we use the REDD data set [22], that contains usage information of electrical devices in smart home environments [10]. Additionally, we use Eye-Tracking data that were produced by persons who translated a text on the computer. To evaluate *BFSPMiner*, we use multiple data sets containing click stream data and data extracted from books.¹ An overview of the used datasets is detailed in Table 1. An open source implementation of our *BFSPMiner* algorithm is available under [30].

For the comparison, we set the support threshold to 1%, the ϵ value to 0.00999 and the α value to 0.00995, as these were the suggested values by Mendes et al. [26]. Additionally we set the pruning period to 5 and the initial batch length to 300. We used a computer with 8 GB RAM and a processor that had four cores running with 3.2 GHz.

For evaluating our prediction model we use the recall, precision and $F1$ scores.

$$\text{recall} = \frac{|\text{correct predictions}|}{|\text{items}|} \quad (3)$$

$$\text{precision} = \frac{|\text{correct predictions}|}{|\text{predictions}|} \quad (4)$$

$$F1 = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}} \quad (5)$$

¹ <http://www.philippe-fournier-viger.com/spmf/datasets>.

Table 1 The data sets used to analyze the described algorithms

Data set	Number of items
EyeTracking [11] and Fig. 1	5511
REDD [22]	45924
Leviathan	387000
Kosarak	396000
MSNBC	846000
FIFA	1476000
Bible	1566000

Kosarak, MSNBC and FIFA contain click stream data. Leviathan contains the words of the book written by Thomas Hobbes as items. EyeTracking data set [11] is explained in Sect. 1 and Fig. 1. The REDD data set [22] contains the on/off recording of smart home electrical appliances

Fig. 5 Increasing the *span* parameter allows the recall, precision and *F1* scores to rise, as each prediction has more chances to turn out true, $k = 1$, REDD data set

The recall (cf. Eq. 3), for instance, is obtained by dividing the number of correct predictions over the amount of processed data. Intuitively, the recall of our predictions rises if we increase the k parameter as well as the *span* parameter, since they allow us to make more predictions that can stay valid for a longer time.

Increasing the *span* parameter, as shown in Fig. 5, will increase the value of all three evaluation measures, as the number of predicted items stay the same, but are valid for a longer time.

The same statement is true for the parameter k . If the number of used rules increases, the recall increases too, but the precision drops significantly, pulling the *F1* measure with it. This can be seen in Fig. 6. Although the recall gets higher, increasing the k parameter results with a guessing game, as too many items are predicted for the next time steps.

To study the parameter sensitivity of *BFSPMiner* w.r.t. the *maxPatternLength*, we started by observing the sensitivity of the competitor w.r.t. their comparative parameter (i.e., the support threshold). Figure 7 depicts the prediction quality of *SS-BE* w.r.t. the support threshold using REDD data set. Intuitively, the recall is decreasing with the increase in

Fig. 6 Higher k values increase the recall, as more items can improve the prediction. However, many more items are predicted, which drops the precision significantly. Higher span value slows the drop of the precision value. (REDD data set). **a** $span = 1$, **b** $span = 3$

Fig. 7 The prediction quality of *SS-BE* w.r.t. the support threshold using REDD data set. Increasing the support threshold allows for faster computation time, but the quality of the predictions decreases. Figure 8 and Table 2 reflect the effect on the pattern generation time and the prediction time, respectively

the support threshold. Comparing Fig. 7 with both Fig. 8 and Table 2 highlights the expected tradeoff. Higher support thresholds result with less generated patterns and thus fewer frequent patterns that can be used as rules for the predictor. As a result, both the pattern generation runtime Fig. 8 and the prediction runtime (Table 2) of the *SS-BE* algorithm increase.

Fig. 8 The pattern generation time of two versions of the *SS-BE* algorithm w.r.t. the support threshold using the REDD data set. Despite some anomalies, a decreasing runtime is obvious that becomes stable for support thresholds above 13%

Table 2 The prediction times of *SS-BE* w.r.t. the support threshold using the REDD data set

Threshold (%)	Prediction time	Items (ms)
1	2122	2.60
2	987	5.60
3	677	8.13
4	559	9.86
5	393	14.02
6	432	12.75
7	393	14.02
8	326	16.90
9	294	18.74
10	265	20.79

Higher support thresholds allow for faster processing of the items, since fewer rules have to be checked. This results in a lower prediction quality (cf. Fig. 7)

ning times. Due to the continuous increase in the running time, we would like to set the *maxPatternLength* to the lowest possible value that maximizes the prediction quality.

For this, we study the effect over the prediction runtime and the prediction quality. The maximal context length (*contextLength*) parameter reflects the maximum length of the used rules. During the prediction, this parameter is upper-limited by the *maxPatternLength*. Thus, studying its maximum value is equal to studying *maxPatternLength*. Figure 10 depicts the effect of the maximal *contextLength* parameter of both the prediction quality of and the prediction runtime of *BFSPMiner* using the REDD data set. The bigger the left side of the association rule, the higher the prediction quality. But this is only true until a certain limit is reached and the most confident rules are included in the length. Afterward, increasing the context does not provide higher recall values, but keeps increasing the runtime for no pay off. This tradeoff is visualized in Fig. 10. Setting the maximal *contextLength* (and thus *maxPatternLength*) to 5 results with the optimal setting for the tradeoff.

To avoid having a data set-specific value of *maxPatternLength*, we check the effect over the prediction quality (recall) of our algorithm w.r.t. the maximal *contextLength* algorithm using the MSNBC data set. Figure 11 depicts the results. Therefore, we can safely use a maximum pattern length of 10 in all data sets. This assumption is also supported by Figs. 10 and 11, showing that interesting and insightful patterns, which are for example usable for prediction, have a length of at most 5. This is also true for the REDD data set, which was the only data set containing some sequential patterns with length higher 10 (due to a continuous turning on and off of a washing machine), but the prediction quality did not increase with context lengths higher than 5 (Table 3).

Table 4 shows the runtime of each algorithm, using multiple data sets. Depending on the amount of unique items

Fig. 9 The pattern generation time of *BFSPMiner* w.r.t. the *maxPatternLength* parameter using the REDD data set

Figure 9 represents the effect of the *maxPatternLength* parameter over the pattern generation time of *BFSPMiner*. Different to the comparable support threshold parameter, higher values of *maxPatternLength* result with higher run-

Fig. 10 **a** The prediction quality (recall) and **b** the prediction time of *BFSPMiner* using the REDD data set. The quality does not increase by rules that contain more items than 5. Increasing the length of the context negatively affects the runtime

Fig. 11 Changing the context length effects the recall of the MSNBC value only until a *contextLength* of 2

Table 3 Basic information for the used data sets

Data set	# Items	# Unique items	# Seq. Pat. (Ground truth)
Leviathan	387000	9027	393543
Kosarak	396000	1408	414352
MSNBC	846000	19	2789570
FIFA	1476000	2992	781992
Bible	1566000	13907	1540632

The number of sequential patterns is determined using an offline *PrefixSpan* and is used as the ground truth for later comparisons

Table 4 Runtime results in milliseconds of each algorithm using different data sets. For each dataset, the shortest runtime is highlighted in bold

Data set	BFSPMiner	SS-BE	SS-LC	BFSPMiner Pruned
Leviathan	18360	2138	8603	36515
Kosarak	26477	1315	1878	45488
MSNBC	3834	12061	13264	15972
FIFA	31495	3732	35864	279795
Bible	83835	7740	106856	405553

Table 5 The overall number of occurrences of sequential patterns found over different data sets

Data set	BFSPMiner	SS-BE	SS-LC	BFSPMiner pruned
Leviathan	393543	352884	363009	393543
Kosarak	414352	404613	405670	414352
MSNBC	2789570	2424469	2440867	2789570
FIFA	781992	779656	780784	781992
Bible	1540632	1366633	1530228	1540632

The largest number of found occurrences are in bold, and are in all of the cases found by *BFSPMiner*. Note also that the pruning of *BFSPMiner* does not affect the number of found sequential patterns

in the data stream and for how long the algorithm needs to be used, the pruning step of the *BFSPMiner* can be turned on and off. Working without the pruning results in a more flexible and faster algorithm that can output results without a fixed support threshold. At each output request, the user can choose to use a different support threshold. If the algorithm is used indefinitely, the pruning has to be used.

Additionally, the runtime of each algorithm differs completely depending on the data set. The difference is especially visible for the MSNBC data, where the *BFSPMiner* is significantly faster than the other data sets, while being slower in most of the other cases. To track the reason behind that, we compared the MSNBC data set with the Bible data set. Two big differences could be revealed. First, while the Bible data sets contain 13907 unique items, the MSNBC set contains only 19 different items (cf. Table 3). Second, although it is larger, the Bible data set has only 17 different sequential patterns after the analysis, in contrast to more than 100 different sequential patterns discovered in the MSNBC data set. This proves the general differences between *BFSPMiner* and the other two algorithms. The *PrefixSpan* algorithm, used by *SS-BE* and *SS-LC*, excels when multiple items can be pruned early, which happens when there are not many single frequent items as in the Bible data set. If the amount of sequential patterns is more dense, as with the MSNBC data set, which contains not many unique items, then more patterns will be found and less items can be pruned. Here is the advantage of

BFSPMiner, where the calculation time for sequential patterns generation stays constant.

Table 5 shows the amount of patterns found by two variants of *BFSPMiner*, *SS-BE* and *SS-LC* (support threshold of 1%). For each sequential pattern, we accumulated their occurrences to determine the found number of patterns. Sequential pattern mining algorithms do not detect usually false positive patterns. Thus, in Table 5 we follow the assumption that finding more sequential patterns results with a higher accuracy in representing the data. In other words, the closer the number of detected patterns to the number of ground truth patterns detected using *PrefixSpan* (cf. Table 3), the better the quality of the sequential pattern mining algorithm. The

results in Table 5 reflect the superiority of *BFSPMiner* and the variant of *BFSPMiner* with pruning compared to the competitor algorithms (cf. Table 3). *BFSPMiner* is able to find all the sequential patterns, that have a *maxPatternLength* = 10. Since *BFSPMiner* is able to find all sequential patterns, Figs. 12, 13 and 14 visualize the relative amount of sequential patterns found by the different algorithms over different periods of time when compared to the ones found statically by *PrefixSpan* at the same periods.

Looking at the top found patterns for a data set should reveal the most important patterns. Yet, as seen in Table 6, the competitor algorithms fail to perfectly find the most popular patterns in their correct ordering. Since the statistics for

Fig. 12 The relative percentage of the number of found patterns by all algorithms w.r.t. the number of patterns found by *PrefixSpan* (the ground truth). The competitor algorithms can only find between 85 and 90% of the sequential patterns in most of the time, while *BFSPMiner* is able always to find the same number of patterns found by *PrefixSpan*. Additionally, the pruning step of the *BFSPMiner* seems not to drop any important patterns. **a** Leviathan data set, **b** Kosarak data set

Fig. 13 A comparison like in Fig. 12 for MSNBC and Fifa data sets. **a** MSNBC data set, **b** Fifa data set

each pattern are flawed, the patterns deviate in their order. Some are not even part of the top pattern set. For this statistic we compared the top patterns of the competitor algorithms to the top patterns of *BFSPMiner*, again by following our first observation that our algorithm finds the patterns lost between the batches of the competitor algorithms. In the table we visualized the resulting deviations. The *max* column reflects the maximum number of missed top patterns by the competitor algorithms when comparing their different outputs of the

batches with the output of *BFSPMiner*. The *avg* column is the final value after averaging over all batches.

To compare the predictability of *BFSPMiner* against that of the *SS-BE* and *SS-LC* algorithm, we used the same prediction model for all three of them. Since the batch-based competitors will not produce new rules in each step, they will only update their set of rules after each batch and predict the items in between with rules from the previous batch. The recall (cf. Eq. 3) results can be seen in Table 7. For this experiment we have set $k = span = 1$, as we hypothesize,

Fig. 14 A comparison like in Fig. 12 for the Bible data set

Table 6 The maximum and the average deviations of the top patterns found by the competitor algorithms w.r.t. the top patterns found by *BFSPMiner*

Data set	Number of top patterns	SS-BE max—avg	SS-LC max—avg
Leviathan	33	2—0.48	7—0.63
Kosarak	21	2—0.24	2—0.24
MSNBC	100	28—3.68	30—4.16
FIFA	5	2—0.08	2—0.08
Bible	17	2—0.32	2—0.32

Table 7 The averaged recall (cf. Eq. 3) of the different algorithms using the EyeTracking and REDD data sets, $k = 1$, $span = 1$

Data set	BFSPMiner	SS-BE	SS-LC
EyeTracking	0.54	0.24	0.24
REDD	0.5	0.13	0.13

Best values are in bold

that those settings reflect the nature of predicting the next item the best. *BFSPMiner* outperforms both competitors on both data sets.

One of the weakest points of the competitors when compared with *BFSPMiner* is being batch-based. Since these algorithms collect 300 new items every time before analyzing them and generating new sequential patterns, the old results become outdated. The prediction is then based on patterns that are not reflecting the current state of the stream. The batch-based algorithms fail to adapt to changes in the data stream. These results are even stronger when put into the context of the data set. The REDD data set contains 29 unique items, this means that guessing the next item would result in

Fig. 15 The figures show a comparison over time between the *BFSPMiner* and *SS-BE* algorithm on different data sets. **a** Bibel data set, $k = 1$ and $span = 1$. **b** Kosarak data set, $k = 1$ and $span = 1$

a 3.5% chance of being correct. Another way of predicting the next item would be to guess for the item with the highest support value, to be the next item. In the case of the REDD data, the most frequent item has a support value of 16.7%. In other words, guessing this item at every step would result in a higher prediction rate than using the described prediction method with the *SS-BE* and *SS-LC* algorithm.

We evaluated the prediction algorithm on different data sets. The different results of our prediction method are visual-

Fig. 16 The figures show a comparison over time between the *BFSPMiner* and *SS-BE* algorithm on different data sets. **a** EyeTracking data set, $k = 1$ and $span = 1$. **b** MSNBC data set, $k = 1$ and $span = 1$. **c** REDD data set, $k = 1$ and $span = 1$

ized in Figs. 15 and 16. The figures show how the *F1* Measure (cf. Eq. 5) changes over time during evaluation. Figures 17 and 18 contain additionally the results from another variant of the *SS-BE* algorithm. Multiple observations are clearly visible in those graphs. First, the *SS-BE* algorithm always starts with 0 for *F1*, since during the time the first batch is collected, the algorithm does not provide any possible predictions at all. Only after that, the batch-based approach algorithm increases its *F1* score. Second, *BFSPMiner* seems to have always a higher prediction quality on the used data sets with $k = 1$ and $span = 1$. Third, the data sets seem to provide different qualities of patterns. While the EyeTracking, MSNBC and REDD data sets allow for a good predictability, having results between 58% and 16%, the Kosarak and Bible data sets seem less easy to predict. The patterns collected from

Fig. 17 EyeTracking data set. By increasing *span*, the *F1* scores increase for all algorithms, ($k = 1$)

the last two data sets are less helpful in getting insights on how the stream behaves in the future. This might be also due to the amount of unique items in the data stream. The first three data sets contain only 19 to 29 different items, while the Bible data set contains over 13900 different items (cf. Table 3). With this information, it is rather obvious that a prediction of this data set is significantly harder.

Figures 17 and 18 show how varying the k and $span$ parameters affects the *F1* score of the algorithms using the EyeTracking data set. The results shown in those figures are inline with the results on the other data sets. Increasing the $span$ parameter tends to increase the *F1* score, since the predictions made are valid for a longer time. This increases the recall value and therefore the *F1* score is increased.

Similar to the study in Figs. 5 and 6, the results of the evaluation in Figs. 19 and 20 show the changes of the precision (cf. Eq. 4), recall (cf. Eq. 3) and *F1* (cf. Eq. 5) scores over REDD and MSNBC data sets. Each of the evaluations in Figs. 19 and 20 is reflecting the prediction quality of *BFSPMiner* against *SS-BE* while varying k and $span$ parameters in more details. Although increasing the k value will also increase the recall score, the *F1* Measure drops. This happens because the amount of new predictions lowers the precision value stronger than it increases the recall value. This is similar to the observation in Figs. 5 and 6. An interesting observation is, however, that *BFSPMiner* is having higher recall scores than *SS-BE* almost all the time on REDD data set and higher precision, recall and *F1* scores all the time on the MSNBC data set. Additionally, it is obvious that all algorithms tend to scale their prediction quality in the same way. A parameter change affects *BFSPMiner* similarly to the *SS-BE* and *SS-BE2* algorithms.

6 Conclusion and future work

In this work, we presented *BFSPMiner*, a sequential pattern mining algorithm designed for data streams. The algorithm manages to achieve a considerable increase in prediction

Fig. 18 EyeTracking data set. By increasing k , the $F1$ scores drop for all algorithms, ($span = 3$)

Fig. 19 Comparing the precision (blue), recall (green) and $F1$ (orange) scores on the REDD data set for both *BFSPMiner* (solid bars) and *SS-BE* (chess bars) (color figure online)

Fig. 20 Comparing the precision (blue), recall (green) and $F1$ (orange) scores on the MSNBC data set for both *BFSPMiner* (solid bars) and *SS-BE* (chess bars) (color figure online)

quality w.r.t. state-of-the-art batch-based algorithms, while being able to analyze data in real time and having in some cases a reasonable running time. This is due to its continuous pattern generation, instead of using batches. The algorithm excels and outperforms state-of-the-art algorithms in data streams with fewer unique items and a higher density of sequential patterns. Additionally, we presented a predictor that is able to predict the next items in the data stream.

The empirical results have shown that the predictability of *BFSPMiner* is significantly stronger than the predictability of competitor algorithms.

In the future, we plan to introduce an adaptive threshold for the maximal pattern length, which is able to increase the prediction quality for possible data sets that contain patterns with higher and undefined maximal pattern length. Additionally, we want to improve our prediction method by using

other statistical approaches, including events with overlapping intervals [14,15,24].

Another interesting research direction that we want to explore is using the considerable deviations in the top k streaming sequential patterns or their statistics as indications for possible drifts in the underlying distributions of the stream [12,13,18–20]. We want to study also the application of our approach to discover causalities in event streams for producing process models [17].

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Abouelhoda, M., Ghanem, M.: String Mining in Bioinformatics, pp. 207–247. Springer, Berlin (2010)
2. Agrawal, R., Srikant, R.: Mining sequential patterns. In: 11th International Conference on Data Engineering (1995)
3. Alves, F., Pagano, A., Neumann, S., Steiner, E., Hansen-Schirra, S.: Units of translation and grammatical shifts: towards an integration of product- and process-based research in translation. In: Translation and Cognition, pp. 109–42 (2010)
4. Beecks, C., Hassani, M., Brenger, B., Hinnell, J., Schüller, D., Mittelberg, I., Seidl, T.: Efficient query processing in 3D motion capture gesture databases. *Int. J. Semant. Comput.* **10**(1), 5–26 (2016)
5. Beecks, C., Hassani, M., Hinnell, J., Schüller, D., Brenger, B., Mittelberg, I., Seidl, T.: Spatiotemporal similarity search in 3d motion capture gesture streams. In: Advances in Spatial and Temporal Databases—14th International Symposium, SSTD 2015, Hong Kong, China, 26–28 August 2015. Proceedings, pp. 355–372 (2015)
6. Cho, C.-W., Wu, Y.-H., Yen, S.-J., Zheng, Y., Chen, A.: On-line rule matching for event prediction. In: VLDB, pp. 303–334 (2011)
7. Ezeife, C., Lu, Y.: Mining web log sequential patterns with position coded pre-order linked wap-tree. *Data Min. Knowl. Discov.* **10**(1), 5–38 (2005)
8. Gündüz, S., Özsu, M.T.: A web page prediction model based on click-stream tree representation of user behavior. In: KDD, pp. 535–540 (2003)
9. Hassani, M.: Efficient Clustering of Big Data Streams. Ph.D. thesis, RWTH Aachen University (2015)
10. Hassani, M., Beecks, C., Töws, D., Seidl, T.: Mining sequential patterns of event streams in a smart home application. In: LWA (2015)
11. Hassani, M., Beecks, C., Töws, D., Serbina, T., Haberstroh, M., Niemietz, P., Jeschke, S., Neumann, S., Seidl, T.: Sequential pattern mining of multimodal streams in the humanities. In: BTW, pp. 683–686 (2015)
12. Hassani, M., Kranen, P., Saini, R., Seidl, T.: Subspace anytime stream clustering. In: Proceedings of the 26th Conference on Scientific and Statistical Database Management, SSDBM' 14, p. 37 (2014)
13. Hassani, M., Kranen, P., Seidl, T.: Precise anytime clustering of noisy sensor data with logarithmic complexity. In: Proceedings of the 5th International Workshop on Knowledge Discovery from Sensor Data, SensorKDD '11 @ KDD '11, pp. 52–60. ACM (2011)
14. Hassani, M., Lu, Y., Seidl, T.: Towards an efficient ranking of interval-based patterns. In: Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, 15–16 March 2016, Bordeaux, France, pp. 688–689 (2016)
15. Hassani, M., Lu, Y., Wischniewsky, J., Seidl, T.: A geometric approach for mining sequential patterns in interval-based data streams. In: 2016 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2016, Vancouver, BC, Canada, 24–29 July 2016, pp. 2128–2135 (2016)
16. Hassani, M., Seidl, T.: Towards a mobile health context prediction: Sequential pattern mining in multiple streams. In: MDM, pp. 55–57. IEEE (2011)
17. Hassani, M., Siccha, S., Richter, F., Seidl, T.: Efficient process discovery from event streams using sequential pattern mining. In: 2015 IEEE Symposium Series on Computational Intelligence, pp. 1366–1373 (2015)
18. Hassani, M., Spaus, P., Cuzzocrea, A., Seidl, T.: Adaptive stream clustering using incremental graph maintenance. In: BigMine 2015 at KDD'15, pp. 49–64 (2015)
19. Hassani, M., Spaus, P., Gaber, M.M., Seidl, T.: Density-based projected clustering of data streams. In: Proceedings of the 6th International Conference on Scalable Uncertainty Management, SUM '12, pp. 311–324 (2012)
20. Hassani, M., Spaus, P., Seidl, T.: Adaptive multiple-resolution stream clustering. In: Proceedings of the 10th International Conference on Machine Learning and Data Mining, MLDM '14, pp. 134–148 (2014)
21. Hassani, M., Töws, D., Seidl, T.: Understanding the bigger picture: batch-free exploration of streaming sequential patterns with accurate prediction. In: Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, 3–7 April 2017, pp. 866–869 (2017)
22. Kolter, J.Z., Johnson, M.J.: Redd: A public data set for energy disaggregation research. In: SustKDD Workshop @KDD (2011)
23. Koper, A., Nguyen, H.S.: Sequential pattern mining from stream data. In: Advanced Data Mining and Applications, pp. 278–291 (2011)
24. Lu, Y., Hassani, M., Seidl, T.: Incremental temporal pattern mining using efficient batch-free stream clustering. In: Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, 27–29 June 2017, pp. 7:1–7:12 (2017)
25. Martin, F., Méger, N., Galichet, S., Nicolas, B.: Forecasting failures in a data stream context application to vacuum pumping system prognosis. In: Transactions on Machine Learning and Data Mining, pp. 87–116 (2012)
26. Mendes, L.F., Ding, B., Han, J.: Stream sequential pattern mining with precise error bounds. In: ICDM, pp. 941–946 (2008)
27. Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., Hsu, M.-C.: Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In: ICDE, pp. 0215–0215 (2001)
28. Schüller, D., Beecks, C., Hassani, M., Hinnell, J., Brenger, B., Seidl, T., Mittelberg, I.: Automated pattern analysis in gesture research: similarity measuring in 3D motion capture models of communicative action. *Digit. Hum. Q.* **11**(2), 1 (2017)
29. Soliman, A.F., Ebrahim, G.A., Mohammed, H.K.: SPEDS: A framework for mining sequential patterns in evolving data streams. In: Communications, Computers and Signal Processing (PacRim), pp. 464–469. IEEE (2011)
30. Töws, D., Hassani, M.: Open Source Implementation of BFSP-Miner. <https://github.com/Xsea/BFSPMiner>
31. Zaki, M.J.: SPADE: an efficient algorithm for mining frequent sequences. *Mach. Learn.* **42**, 31–60 (2001)

32. Zhong, N., Li, Y., Wu, S.-T.: Effective pattern discovery for text mining. *IEEE Trans. Knowl. Data Eng.* **24**(1), 30–44 (2012)
33. Zhou, C., Cule, B., Goethals, B.: A pattern based predictor for event streams. In: *Expert Systems with Applications*, pp. 9294–9306 (2015)
34. Zhu, H., Wang, P., Wang, W., Shi, B.: Stream prediction using representative episode rules. In: *IEEE 11th International Conference on Data Mining Workshops*, pp. 307–314 (2011)