Mining Sequential Patterns of Event Streams in a Smart Home Application

Marwan Hassani, Christian Beecks, Daniel Töws, and Thomas Seidl

Data Management and Data Exploration Group
RWTH Aachen University, Germany
{hassani,beecks,toews,seidl}@cs.rwth-aachen.de

Abstract. Recent advances in sensing techniques enabled the possibility to gain precise information about switched-on devices in smart home environments. One is particularly interested in exploring different patterns of electrical usage of indoor appliances and using them to predict activities. This in turns results with many useful applications like inferring effective energy saving procedures. The necessity to derive this knowledge in the real time and the huge size of generated data initiated the need for a precise stream sequential pattern mining approach. Most available approaches are less accurate due to their batch-based nature. We present a smart home application of the *PBuilder* algorithm which uses a batch-free approach to mine sequential patterns of a real dataset collected from appliances. Additionally, we present the *StrPMiner* which uses the PBuilder to find sequential patterns within multiple streams. We show through an extensive evaluation over a smart home real dataset the superiority of the StrPMiner algorithm over a state-of-the-art approach.

1 Introduction

Careful usage of indoor electrical devices is an important topic in the field of energy saving and sustainability. Understanding the usage patterns of appliances during a typical day is the key to induce savings of electrical energy. If a domain expert finds anomalies in the electricity usage of one house, which consumes a lot of energy, he can help the householder by suggesting lesser consuming patterns. Recent advances in sensing techniques enabled the possibility to gain precise information about different switched-on devices in a smart home environment. This information contains the time and the duration when a particular appliance was turned on. Gaining knowledge about correlation patterns between the activation of different devices is possible with an offline visualization of a small-sized data collected from a limited number of appliances (cf. Figure 1). This tends to be sophisticated when one requires an instant knowledge about the usage needed during the collection time. Additionally, the number of devices and

 In: R. Bergmann, S. Görg, G. Müller (Eds.): Proceedings of the LWA 2015 Workshops: KDML, FGWM, IR, and FGDB. Trier, Germany, 7.-9. October 2015, published at http://ceur-ws.org

even the houses should usually be big enough to gain useful patterns. This signals the necessity to apply data mining methods to collect handy usage patterns.

A data stream produces an infinite and continuous flow of data. Regularities can often be found in those streams, which give information about the connection between the events in the data. To find this hidden information, sequential pattern mining algorithms can be used over the data stream. A suitable algorithm is able to reveal electric devices that are often used with or implied by each other. Sequential pattern mining is a special case of frequent item set mining, where patterns have to be frequent subsequences of the stream. Each pattern has to appear a certain number of times within a part of the stream (called *batch*) to count as a sequential pattern.

Additional challenges arise when looking at multiple streams at once, as patterns can be part of one or multiple streams. This is the case in a smart home environment, as each electric device provides a different data stream, feeding us with new information. For this a special treatment of data is needed, so that a useful connection among multiple electric devices can be found.

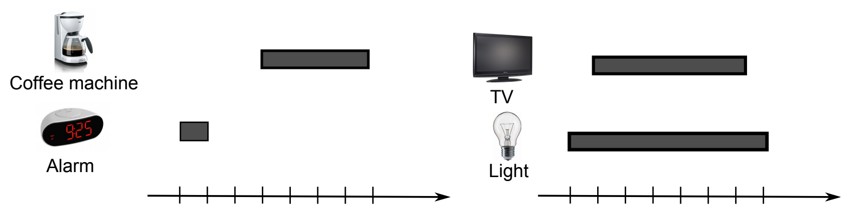


Fig. 1. If a person drinks a coffee every morning, the data would contain a connection between the alarm clock and the coffee machine. In particular the alarm would imply the coffee machine. In the second example the time frame, in which the TV is used, would be contained by the time frame, in which the light is used.

Multiple algorithms were proposed in the literature to mine sequential patterns from data streams. Most of them use a batch approach, like the *SS-BE* algorithm [11]. The batch approach is a simple and efficient solution to mine sequential patterns in a stream. However, it leaves a room for errors. Sequential patterns are, by definition, very sensitive to the order of items. This order can not be found when searched patterns are located between two consecutive batches. A batch-based algorithm will fail to detect such patterns. Moreover, single items might have a duration as in the case of the interval-based events in our smart home application (cf. Fig. 1). These items might also span multiple batches.

In this work, we present an application over a real smart home dataset using two algorithms [14] that avoid the above mentioned errors. The first algorithm is the Pattern Builder *PBuilder* which mines sequential patterns for given data using a batch-free approach. The second algorithm is the Streaming Pattern Miner *StrPMiner* which uses the *PBuilder* to find sequential patterns within multiple streams arriving from multiple indoor appliances and keeps track of their quality.

The remainder of this paper is organized as follows: Section 2 presents some related work. Section 3 looks at the preliminaries of sequential pattern mining. Additionally it will highlight the problem with the batch approach. In Section 4 two algorithms are presented. The algorithm *StrPMiner* is then tested against the *SS-BE* algorithm in Section 5, where we will also prove its superior accuracy. The paper is concluded with a summary and an outlook in Section 6.

2 Related Work

Optimizing sequential pattern mining is an important task in the streaming data mining field, which leads to a lot of different algorithms. A base algorithm for many approaches [11],[13],[15],[2], is the *PrefixSpan* algorithm [12]. The *PrefixSpan* algorithm was designed for a static data environment. Because of this it can use the apriori assumption [1], that every part of a frequent pattern also has to be frequent. In the *PrefixSpan* patterns are generated bottom up. Starting with a frequent item, each pattern will be checked for its frequency. If it is frequent, it will be used as a prefix for other frequent items to generate longer patterns. All algorithms using the *PrefixSpan* in a stream environment collect data in a batch instead of evaluating each item as soon as it arrives.

Since the streaming approach allows to only look at data once, algorithms have to make compromises in order to provide fast results. [11] proposes two algorithms with different pruning strategies, the *SS-BE* and *SS-BM* algorithms. These algorithms restrict memory usage but are able to find all true sequential patterns and allow an error bound on the false positives. The patterns are saved in a new designed tree structure, the T_0 tree. The tree will be frequently checked and pruned. Patterns that did not reappear frequently in the past will be deleted, so that only current frequent items are contained in the tree.

In a static data set, all information needed for the algorithm is provided from the beginning, while in the streaming approach new data arrives every second, thus, patterns that were not frequent in the beginning may become frequent later on. Yet, it is impossible to save every pattern and its information. The *FP-stream* [3] solves this issue by saving information in different time granularities. The newer the information, the more accurate it will be displayed. Another way to solve the memory problem is by using a sliding window model, in which only the most recent data is being looked at. The *MFI-TransSW* algorithm [10] optimizes this concept. The algorithm works in three steps: window initialization, window sliding and pattern generation. Previously described algorithms only provide solutions for one stream. In cases of multiple streams in parallel, the *MSS-BE* algorithm [8] is an idea to find sequential patterns in an multiple-stream environment, where pattern elements can be part of different streams.

The algorithms mentioned above only provide solutions for frequent pattern mining or find sequential patterns by using batches. The stream pattern miner (StrPMiner) algorithm which uses the PBuilder was first introduced in [14]. It uses a sliding window approach instead of the batch method while efficiently mining sequential patterns of the streams. The algorithm was successfully used in an application within the humanities domain, for analysis of translation data, where subjects are translating English texts into German. The two streams in that case were the eye gazes of the translators and their collected keystrokes during the translation session [4,5,14].

3 Preliminaries: Sequential Pattern Mining

We are given a set $\mathcal{S} = \{S^1, S^2, \ldots, S^{|\mathcal{S}|}\}$ of $|\mathcal{S}|$ different streams arriving from different observed parameters collected from the smart home. Each stream S^k is represented by streaming, time-stamped interval-based events that evolve over the time. Thus, the first n items of stream S^k are represented as $S^k = \{s_1^k, s_2^k, \ldots, s_n^k\}$ where s_i^k is an observed event that occurs at time t_i where $t_i < t_{(i+1)}$ for all $i = 1, \ldots, n$. Each event is additionally described by its label. A sequential pattern is a combination of multiple events that follow each other. These patterns can be used to find correlations in the data.

We are asked to obtain the different frequent patterns that appear within a *single* stream S^k and also within *multiple* streams from $\mathcal{S}$ (also called multimodal streams). The sequential pattern mining problem differs from the normal frequent item set mining in the fact that the order of items (events) matters. The problem of mining sequential patterns is defined as follows: Let $I = \{i_1, i_2, \ldots, i_{|I|}\}$ be a set of $|I|$ items, each item consists of a timestamp and a duration. A pattern is represented here by a sequence, which is an ordered list of items from I denoted by $\langle p_1, p_2, \ldots, p_k \rangle$. Thus, a sequence $p = \langle a_1, a_2, \ldots, a_q \rangle$ is a subsequence of a sequence $p' = \langle b_1, b_2, \ldots, b_r \rangle$ if there exists integers $i_1 < i_2 < \cdots < i_q$ such that $a_1 = b_{i_1}, a_2 = b_{i_2}, \ldots, a_q = b_{i_q}$.
This definition of sequential pattern mining is very feasible for the continuously emerging characteristics of stream data. A stream S^k in this context is an arbitrarily large list of sequences p_i. A sequence p in the data stream S^k *contains* another sequence p' from S^k if p' is a subsequence of p. The *count* of a sequence p, denoted as $count(p)$, is defined by the number of sequences that contain p in the stream S^k. If the frequency of a pattern (p) within a window w of the stream S^k is greater or equal to a user defined threshold min_supp, then the sequence p is a frequent sequence or a sequential pattern in that window of S^k.

Following the apriori principle [1], given two subsequences $p = \{p_1, p_2, ..., p_n\}$ and $p' = p \backslash \{p_n\}$, it holds that $supp(p') \geq supp(p)$ due to the anti-monotonicity property. Thus, if p is a sequential pattern, p' is also a sequential pattern.

To provide different views on the data, three different window concepts are used by the *StrPMiner*. The algorithm works with the Landmark Window, the Sliding Window and the Damped Window concept. In the Landmark Window, a point in time is defined as the *landmark*. All data is then collected starting from the *landmark*. This concept allows to look at big parts of the data. The Sliding Window concept uses a fixed window size and slides it over the data. Thus, only a snapshot of the data will be monitored at any given time. An advantage is that old patterns will be forgotten eventually, which leaves only current information. The Damped Window weights the objects to reflect their age. New items will be

more important than old ones. This allows a compromise between the Landmark Window and the Sliding Window concept. A good solution to find sequential patterns in a streaming environment is the batch approach. It allows to use the Apriori principle, since each batch provides a static data set. However it comes at a cost. Given a support threshold of 2, meaning a pattern has to appear two times within one batch to be counted as frequent, a batch size of 3 and following sequence: $(A, B, C, A, C, C, A, D, C, A...)$ with A, B, C, D being items of a stream. The online component would cut the data stream in following batches:
1. (A, B, C) 2. (A, C, C) 3. (A, D, C) 4. (A, ...) 5. ...

In this case, no pattern would be frequent. Looking at the whole data without cutting it into batches would reveal that the pattern C, A appears three times, which is over the support threshold of 2. This would lead to a frequent pattern. Additionally, all items except for C in the second batch, would be pruned away, although the item A and C appear in every batch. This leads to two reasons for errors through the batch approach: First: Patterns that appear between batches will not be found. Second: Items and patterns that do not appear often in one batch will be pruned, although they are frequent in the whole data set. The *StrP-Miner* was designed to avoid the batch approach because of these two reasons which result into false statistics for sequential patterns.

4 The *StrPMiner* and the *PBuilder* Algorithms

Since the *PrefixSpan* algorithm only scales well when the candidates for sequential patterns can be pruned, the *StrPMiner* reverses the idea of the *PrefixSpan* and uses a new algorithm called the Pattern Builder (*PBuilder*). This allows the *StrPMiner* to work on each data item step by step as it comes in.

To provide a more focused view on the order of the items, the definition of sequential patterns was changed slightly. As stated previously, a sequential pattern is a frequent subsequence. We redefine subsequences, and sequential patterns, as only allowed to be a list of ordered items that directly follow each other. Thus, p is considered a subsequence of q if $p = (p_1, p_2, ..., p_n)$, $q = (q_1, q_2, ..., q_m)$ and there exist integers $i_1 < i_2 < ... < i_m$ such that $p_1 = q_{i_1}, p_2 = q_{i_2}, ..., p_n = q_{i_n}$ for $n < m$ *and* for all k, l with $l, k < m$ and $l = k + 1$.

The *StrPMiner* handles arriving data from multiple streams at once. For this, we assume that at each point in time only one item can arrive per stream. If multiple items from multiple streams arrive at the same time, they will be put into an ordered list and the algorithm handles each item after another. First an item will be compressed, as only the label and the timestamp are relevant for creating sequential patterns. Then the *StrPMiner* passes the item to the *PBuilder*. The *PBuilder* then uses this data to create sequential pattern candidates. After this, the *StrPMiner* saves the candidates in the T_0 tree structure and keeps track of those candidates and their corresponding statistics. Currently this is the count value, which allows to calculate the support and confidence value of a pattern. The tree will be updated with the new count values and if a pattern was not part of the tree a new node will be created. This approach allows full accuracy, and flexibility in the output, as the support threshold can be changed at every output request. This is not possible when using the *PrefixSpan*, since the threshold has to be previously set.

4.1 The *PBuilder*

The *PBuilder* creates only patterns that contain the newly arrived item. Since it is the last arrived item, all created patterns will end with this item. Given an item A as the newly arrived item, the *PBuilder* starts with this item as a pattern of length one. After this, the algorithm recursively adds older items as a prefix to the previously created postfix. To ensure that the *StrPMiner* only finds direct sequential patterns, the prefix is a direct predecessor of the postfix. As visible in the pseudocode, visible in Agorithm 1, the ItemList only contains the latest items ordered by their appearance. The newest item is the last item in the list. In the first iteration, the currentPattern parameter is empty. Line 7 will then recursively add a prefix to our current pattern. The resulting pattern will be inserted into the tree, as visible in Line 9. This will be repeated, until the complete ItemList was included.

For each created pattern, the *PBuilder* algorithm calls the update function of the T_0 tree. An example of the tree can be seen in Figure 2.

Algorithm 1: The *PBuilder* explained with pseudo code

```
1  PBuilder
   Data: ItemList, currentPattern
2  //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
3  int index = ItemList.length;
4  //create patterns until maxPatternLength is reached
5  while currentPattern.length ≤ ItemList.length do
6  |   //add the next item to the pattern
7  |   currentPattern = ItemList.get(index-currentPattern.length) +
   |   currentPattern;
8  |   //update the tree with the new pattern
9  |   updateTree(currentPattern);
10 end
```

4.2 Maximum Pattern Length as a Solution for Exponential Growth

In contrast to a static database, where all information is available from the beginning, the streaming approach does not have any information on what future items and their frequency might look like. This means that any item and pattern that is currently not frequent in a stream, can become frequent at any later point in time. The support of every pattern changes with every new arriving item. To ensure that at every time the user requests an output all sequential patterns are part of the output, every possible pattern and its information have to be saved. This causes an exponential increase of the calculation time, as with every new arriving item more patterns can be created. Additionally, the memory space will eventually collapse, as the amount of data that has to be saved also increases exponentially.

To stop the exponential growth, the *StrPMiner* introduces a parameter called *maxPatternLength*, as an upper Bound for the pattern length. This variable restricts the *PBuilder* to only look at the last *maxPatternLength* items. A *maxPatternLength* of five, will cause patterns to maximally contain five items, as only those are given to the algorithm. Given this bound, the calculation time in each step only scales with the size of the *maxPatternLength* parameter. Additionally this parameter bounds the maximum growth of the required memory space. On the one hand, as the parameter will not change over the time, the calculation time for each new arriving item will be constant. On the other hand this upper bound filters patterns, before they have been created. Sequential patterns that have a length higher than the given bound, will not be found. With this in mind, a careful selection of the upper bound is important, as it provides a trade off between the calculation time and accuracy.

4.3 Different Window Models

As previously mentioned, the *StrPMiner* uses the T_0 tree introduced by [11]. For the algorithm slight adaptations were made, regarding the saved information. The *StrPMiner* saves the label of the item and the time stamps, at which it appeared, of the pattern in each node. The count of each pattern is then determined by the number of time stamps saved in the corresponding node. An example is shown in Figure 2.

The sliding window model helps to provide another view on the data, as it only contains knowledge of recent data and forgets old data. This helps in cases, where the data changes drastically over the duration of the stream. The landmark window would still show old patterns even though they did not reappear for a long time. In general, the whole algorithm works the same, as in the landmark window, except for an extra pruning step. For this the time stamp of the corresponding item and the patterns created with it have to be deleted from the T_0 tree, which is one path.

5 Experimental Results

Because of the problems that come with the batch approach, the *StrPMiner*, unlike the *SS-BE* algorithm, does not use the *PrefixSpan*. Instead it uses the *PBuilder*, which handles each newly arriving item immediately, without using the batch approach. In this section we compare the presented algorithm to the *SS-BE*, since it is a current state of the art algorithm that finds sequential patterns in a stream environment. Other algorithms we looked at did not fulfill both of these criteria.

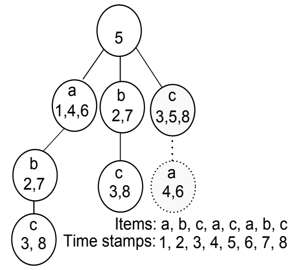


Fig. 2. An example of the T_0 tree. The dotted node represents the pattern (c,a).

For the experimental evaluation of both algorithms we used the REDD dataset [9]. This dataset contains information about the usage of electric devices in Smart Homes. For analyzing those information we preprocessed the data to an event stream. Each stream represents one electronic device, where the items contain the information about the on and off time of the objects. For example, if the oven is turned on at time t, the corresponding item at time t will be labeled *oven +* and *oven -* if it is turned off. Following this code, the patterns of the examples in Figure 1 would be *alarm +, alarm -, coffee +, coffee -* and *light +, tv +, tv -, light -*.

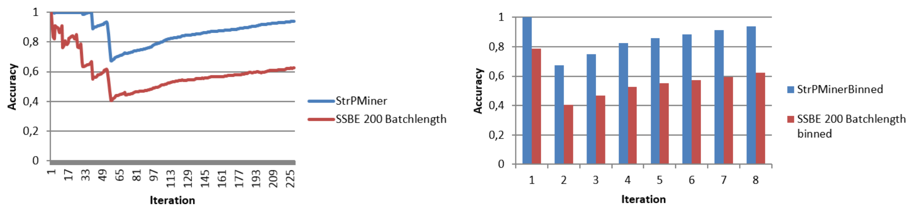
Since we are only interested of direct sequential patterns, we adapted the *PrefixSpan* in such a way that it will only create direct sequential patterns. The adaption will additionally effect the results output by the *SS-BE* algorithm, as it is dependent on the results produced by the *PrefixSpan*.

The support threshold, the only parameter used by both algorithms, was set to 1%.

For the *StrPMiner* we set the maximum pattern length at 200. As explained in 4.2, this parameter strongly influences the patterns that we find and our runtime. The runtime of the *StrPMiner* is slower than the runtime of the *SS-BE*, but with this parameter setting we still ensure real time results. Our assumption is, that, with this setting, the *PBuilder* will find every pattern that is shorter than 200. This result into full accuracy for those patterns. In this evaluation we only want to look at the strong accuracy of the *StrPMiner*, we will only use the Landmark Window here. The Sliding Window and the Damped Window show similar results.

The parameters we set for the *SS-BE* algorithm were the significance threshold ϵ with 0.0099 and the pruning period δ to 10. Those settings are close to those used by the authors [11]. This means, that after ten batches the algorithm will prune the t_0 tree. The batch length is either set to 200 or to 300. Those settings ensure that we will compare both algorithms to similar patterns and similar output.

In a first evaluation we compared both algorithms against a ground truth, which contains all patterns with a support of at least 1%. As the *SS-BE* algorithm uses the batch approach, an output can only be generated after batch



(a) Comparing each time stamp.

(b) The evaluated time stamps were binned and then compared.

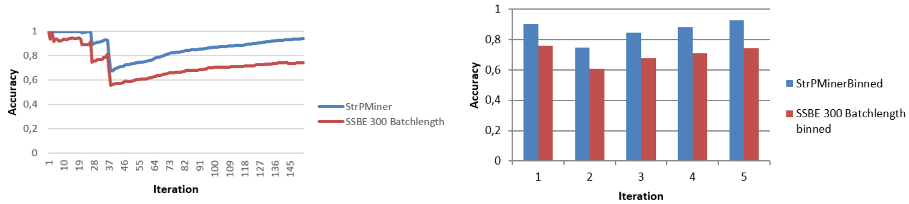
Fig. 3. A comparison of the *StrPMiner* to the *SS-BE* algorithm after evaluating one house. The y-axis displays the accuracy, while the x-axis shows the time. The batch length was set to 200.

length amount of items were evaluated. This means in our case, that only after each 200 or 300 items, an output is available. In contrast to this, the *StrPMiner* can produce a valid output after each item, as it will treat each item directly. In Figure 3 we compared the result of both algorithms to the output after each 200 items. Additionally another comparison is created, where we bin the single time steps. As visible in this figure, the *StrPMiner* has a significantly higher accuracy, which is 30% points higher at each single time step for the given data. Two other things are also visible in this figure. First, the accuracy of the *StrPMiner* stays 100% for the first few time steps, as long as there are no frequent patterns found with a higher length than 200. Second, there is a noticeable drop in the accuracy during the first third of the evaluation. A closer look into the data reveals, that during this time the amount of patterns, that have a higher length than 200, is rising. But, all of those patterns are single stream patterns, with a switching on and off event of one single device, happening in a few seconds. The binning is used to smooth out those abrupt changes and provide a focused view on the general direction of the results.

Although the accuracy of the *SS-BE* algorithm rises with a higher batch length, all three observations are still visible in Figure 4. We tested the algorithms against multiple houses, in which the accuracy of the algorithms changed slightly, but the general direction was the same, revealing the higher accuracy of the *StrPMiner*. In houses with less noisy data, we were even able to maintain full accuracy with the *StrPMiner*, as there were no frequent patterns with a high batch length.

In most of the evaluation the higher batch length setting shows to be more accurate, but still has a lower accuracy of nearly 20% points.

In a next step we wanted to prove our assumption. Only looking at the most important patterns, meaning the top 100 patterns with the highest support, reveals that the *PBuilder* has a full accuracy for all patterns with a length lower than the maximum pattern length. A comparison to the *SS-BE* algorithm is visible in Figure 5. This figure shows, how many of the hidden patterns in the data could be found. In this case, the *SS-BE* algorithm has a high accuracy of over 90%, but is still beaten by the full accuracy of the *StrPMiner*.

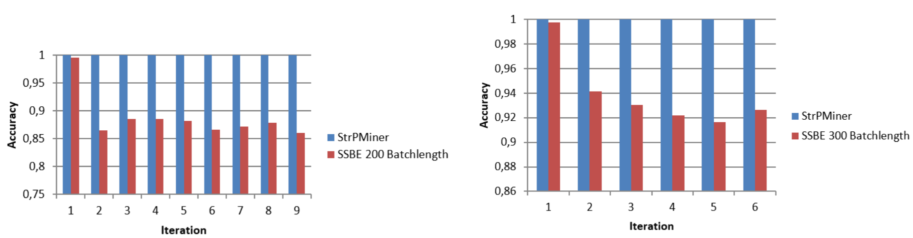


(a) (b)

Fig. 4. A similar comparison as in Figure 3, but with a batch length of 300 for the *SS-BE* algorithm.

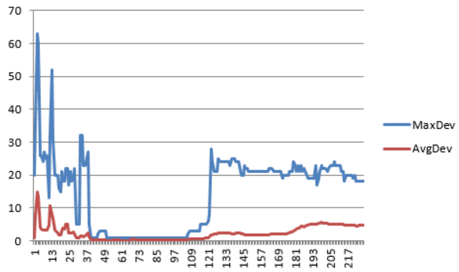
Taking a closer look at the order of the top 100 reveals, that, due to the full accuracy, the *StrPMiner* is able to show all important patterns in the correct order, sorted by their support value. The *SS-BE* algorithm is not able to keep the correct position of the patterns. Figure 6 shows the deviation of the patterns at each time step. The figure shows the mean deviation over all patterns, and the maximal deviation of one pattern.

Although these results show the higher accuracy for the *StrPMiner*, they only represent the average case, formed by looking at all patterns. The open question is, how can these results help in an application case, where we want to find and keep track of specific interesting patterns? The open assumption we want to test is, that both algorithms are able to find meaningful patterns. This means, patterns that show an existing connection between the items contained in it. To test this assumption, we created a correlation matrix for the devices in the data set. A snapshot of it is shown in Table 1, which gives information about how often the items were turned on or off *together*. A higher value means that the on and off time of those two items is close to each other. With this correlation matrix we may not gain information about the specifics of the connection of two items, but we can safely say, that there is a connection between those items.

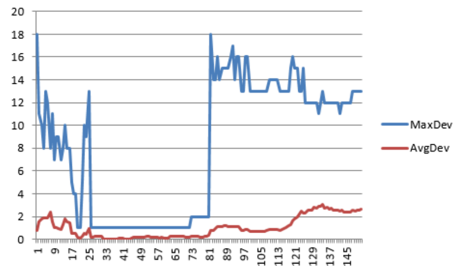


(a) (b)

Fig. 5. For the most important patterns, the top 100, both algorithms show a higher accuracy. Notable is, that the *StrPMiner* provides full accuracy.



(a) The deviation for a batch length of 200 reaches up to 60.



(b) The deviation for a batch length of 300 reaches only to 20.

Fig. 6. This figure shows the deviation between the top 100 patterns created by the *SS-BE* algorithm, compared to the ground truth. A maximal deviation of 10 means, that a pattern a, that appeared at position x in the ground truth, will appear at position $x + 10$ or $x - 10$ in the results of the *SS-BE* algorithm.

	oven	oven	refrigerator	dishwasher	k_outlets	k_outlets	lighting
oven	1	0.828	0.046	0.387	0	0	0.006
oven	0.828	1	0.051	0.307	0	0	0.005
refrigerator	0.046	0.051	1	0.022	0	0	0.011
dishwasher	0.387	0.307	0.022	1	0	0	0
k_outlets	0	0	0	0	1	0	0
k_outlets	0	0	0	0	0	1	0
lighting	0.006	0.005	0.011	0	0	0	1

Table 1. A part of the correlation matrix between some appliances.

This is also reflected in the results of the *StrPMiner*, as patterns between two items with a high correlation, are the multimodal patterns with the highest support. Item combinations with a correlation of over 0.6 are part of the frequent patterns. These patterns, like *oven 3+, oven 4+* and *oven 3-, oven 4-* show that both items are often used with each other. Six of those multimodal patterns have a higher support than 1% in the ground truth and can be found with full accuracy in the results of the *StrPMiner*. In contrast to this, the *SS-BE* algorithm can find three of those with an error rate of over 5%. The other 3 items are not part of the results at all, as they were pruned out of lost between batches of *SS-BE*.

6 Conclusion and Future Work

In this paper we have presented a smart home application over a recent algorithm, the *PBuilder* [14], that is able to mine sequential patterns in data streams. The *StrPMiner* [14] uses the *PBuilder* for the pattern calculation in multiple streams. The results are saved in the T_0 tree. Three different window concepts allow to present the data in different perspectives, which helps users to analyze the data more effectively. Additionally the algorithm can create the output in a much more flexible way than other algorithms, that use the *PrefixSpan*. For each output request any support threshold can be given and the output can be created correctly. The usefulness of the algorithm is tested with the big smart home

REDD dataset. We compared the *StrPMiner* against the *SS-BE* algorithm. In our experimental evaluation we showed, that our algorithm has a significantly higher accuracy than the competitor. Additionally, we showed that the algorithm is capable of running over big real datasets.

In the future we plan to improve the time efficiency of our algorithm. Although our algorithm is able to calculate the results in real time, it is slower than the *SS-BE* algorithm. We found the bottleneck in the insertion step of the data into the T_0 tree. First changes could improve the runtime significantly. We would like additionally to test our approach in distributed, multi-source sensor streaming environments [7] and in anytime environments [6].

References

1. R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.
2. Y.-C. Chen, C.-C. Chen, W.-C. Peng, and W.-C. Lee. Mining correlation patterns among appliances in smart home environment. In *PAKDD*, pages 222–233. 2014.
3. C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu. Mining frequent patterns in data streams at multiple time granularities. *Next gen. DM*, 212:191–212, 2003.
4. M. Hassani. *Efficient Clustering of Big Data Streams*. PhD thesis, RWTH Aachen University, 2015.
5. M. Hassani, C. Beecks, D. Töws, T. Serbina, M. Haberstroh, P. Niemietz, S. Jeschke, S. Neumann, and T. Seidl. Sequential pattern mining of multimodal streams in the humanities. In *BTW*, pages 683–686, 2015.
6. M. Hassani, P. Kranen, and T. Seidl. Precise anytime clustering of noisy sensor data with logarithmic complexity. In *SensorKDD Workshop @KDD*, pages 52–60, 2011.
7. M. Hassani, E. Müller, P. Spaus, A. Faqolli, T. Palpanas, and T. Seidl. Self-organizing energy aware clustering of nodes in sensor networks using relevant attributes. In *SensorKDD Workshop @KDD*, pages 39–48, 2010.
8. M. Hassani and T. Seidl. Towards a mobile health context prediction: Sequential pattern mining in multiple streams. In *MDM*, pages 55–57. IEEE, 2011.
9. J. Z. Kolter and M. J. Johnson. Redd: A public data set for energy disaggregation research. In *SustKDD Workshop @KDD*, 2011.
10. H.-F. Li and S.-Y. Lee. Mining frequent itemsets over data streams using efficient window sliding techniques. *Expert Sys. w. App.*, 36(2):1466–1477, 2009.
11. L. F. Mendes, B. Ding, and J. Han. Stream sequential pattern mining with precise error bounds. In *ICDM.*, pages 941–946, 2008.
12. J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE*, pages 0215–0215, 2001.
13. A. F. Soliman, G. A. Ebrahim, and H. K. Mohammed. Speds: A framework for mining sequential patterns in evolving data streams. In *Communications, Computers and Signal Processing (PacRim), 2011*, pages 464–469. IEEE, 2011.
14. D. Töws, M. Hassani, C. Beecks, and T. Seidl. Optimizing sequential pattern mining within multiple streams. In *BTW*, pages 223–232, 2015.
15. S.-Y. Wu and Y.-L. Chen. Mining nonambiguous temporal patterns for interval-based events. *KDE*, pages 742–758, 2007.