Information Systems 90 (2020) 101451

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Detection and removal of infrequent behavior from event streams of business processes

Sebastiaan J. van Zelst [a,b,*], Mohammadreza Fani Sani [b], Alireza Ostovar [c], Raffaele Conforti [c], Marcello La Rosa [c]

[a] Fraunhofer Institute for Applied Information Technology, Sankt Augustin, Germany
[b] RWTH Aachen University, Aachen, Germany
[c] University of Melbourne, Melbourne, Australia

ARTICLE INFO

Article history:
Received 31 January 2019
Received in revised form 31 May 2019
Accepted 10 September 2019
Available online 9 October 2019
Recommended by Gottfried Vossen

Keywords:
Process mining
Event streams
Filtering
Outlier detection
Anomaly detection

ABSTRACT

Process mining aims at gaining insights into business processes by analyzing the event data that is generated and recorded during process execution. The vast majority of existing process mining techniques works offline, i.e. using static, historical data, stored in event logs. Recently, the notion of online process mining has emerged, in which techniques are applied on live event streams, i.e. as the process executions unfold. Analyzing event streams allows us to gain instant insights into business processes. However, most online process mining techniques assume the input stream to be completely free of noise and other anomalous behavior. Hence, applying these techniques to real data leads to results of inferior quality. In this paper, we propose an event processor that enables us to filter out infrequent behavior from live event streams. Our experiments show that we are able to effectively filter out events from the input stream and, as such, improve online process mining results.

1. Introduction

Modern information systems record the execution of the business processes they support. Common examples include order-to-cash and procure-to-pay processes, typically tracked by ERP systems. Process mining [1] aims to turn such event data into valuable and actionable knowledge, i.e. allowing us to identify and rectify process performance and/or compliance issues. Different process mining techniques are available, including techniques for automated *process discovery*, *conformance checking* and *process enhancement*. In process discovery, we aim to reconstruct the underlying structure of the business process, in the form of a process model. In conformance checking, we assess to what degree the recorded data aligns with a normative process model. In process enhancement, we enhance the view we have of the process, as represented by a process model, by incorporating performance information, data-driven decision points, etc.

Most process mining techniques are defined in an *offline setting*, i.e. they work over historical data of completed process executions, e.g. over all orders fulfilled in the past six months, and have been proven successful in a variety of successful case studies [2]. However, they are typically inadequate to work in *online settings*, i.e. analyzing live streams of events rather than historical data. Hence, they cannot be used for operational support and monitoring, rather, only for a-posteriori analysis. At the same time, the notion of online process mining provides a wealth of opportunities. For example, applying conformance checking techniques in online settings allows us to detect compliance deviations as soon as they occur, and potentially predict them in advance. In turn, the corresponding insights gained are valuable in order to rectify the affected process executions on the fly, i.e. avoiding the deviations to occur altogether.

Recently, several process mining techniques have been designed to work in the online setting, i.e. with the aim to analyze the events executed in the context of a business process at the moment they occur. For example, these newly developed techniques include concept drift detection [3–5], automated process discovery [6–8], conformance checking [9–11] and predictive process monitoring [12]. These techniques directly analyze the *event stream* produced by an information system, rather than the conventionally used static event logs. However, these techniques typically assume the event stream to be free of noise and anomalous behavior. In reality, several factors cause this assumption to be wrong, e.g. the supporting system mistakenly triggers the execution of an inappropriate activity that does not belong to the process, or a system overload results in logging errors. The existence of these anomalies in event streams easily leads

* Corresponding author at: Fraunhofer Institute for Applied Information Technology, Sankt Augustin, Germany.
E-mail addresses: sebastiaan.van.zelst@fit.fraunhofer.de, s.j.v.zelst@pads.rwth-aachen.de (S.J. van Zelst).

https://doi.org/10.1016/j.is.2019.101451
0306-4379/

to unreliable results. For example, in drift detection, sporadic stochastic oscillations caused by noise negatively impact drift detection accuracy [3,5].

In this paper, we propose a general-purpose event stream filter, designed to detect and remove *infrequent behavior* from event streams. The approach presented, relies on a time-evolving subset of behavior of the total event stream, out of which we infer an incrementally-updated model that represents this behavior. In particular, we build a collection of *probabilistic automata*, which are dynamically updated to filter out spurious events. Using a corresponding prototypical implementation, we evaluated accuracy and performance of the filter by means of multiple quantitative experiments. To illustrate the applicability of our approach w.r.t. existing online process mining techniques, we assessed the benefits of our filter when applied prior to drift detection.

This paper extents earlier work, presented in [13], in the following dimensions. We have extended the algorithm by adding an *emission delay* and an *automaton voting scheme*, in order to more accurately detect unwanted infrequent behavior. We have conducted a new set of large scale experiments, including the aforementioned newly added functionality. Within these newly conducted experiments, we have extended the amount of noise present in the input data, as well as the range of filtering thresholds used. Finally, we have added a noise-oriented taxonomy of process mining behavior in the context of (online) process mining, and highlighted what classes of the taxonomy are covered by the proposed filter.

The remainder of this paper is structured as follows. In Section 2, we discuss related work. In Section 3, we present background concepts, introducing (online) process mining and filtering. In Section 4, we present our approach, which we extensively evaluate in Section 5. In Section 6, we discuss the proposed filter and indicate what types of noise are detected by the filter, in terms of a broader control-flow-oriented behavioral taxonomy. We conclude the paper and discuss several avenues for future work in Section 7.

2. Related work

For the purpose of this paper, we primarily focus on work conducted in the area of *online process mining* as well as *process mining oriented noise filtering*. For an extensive overview of process mining, we refer the reader to [1]. Similarly, for an extensive overview of general purpose outlier detection, we refer to [14,15].

In the area of online process mining, the majority of work concerns automated process discovery algorithms. For example, Burattin et al. [7] propose a basic algorithm that lifts an existing offline process discovery algorithm [16] to the online setting. Additionally, in [17], Burattin et al. propose an online process discovery technique for the purpose of discovering declarative models. In [6], Hassani et al. extend [7] by proposing the use of indexed prefix-trees in order to increase memory efficiency. Finally, in [8], van Zelst et al. extend and generalize [6,7] for a large body of different classes of existing process discovery algorithms. More recently, event streams have been used for online conformance checking [9–11] and online concept drift detection [3–5]. In the context of online conformance checking, Burattin et al. [9] propose an approach that uses an enriched version of the original process model to detect deviant behavior. In [10], van Zelst et al. propose to detect deviant behavior by incrementally computing prefix-alignments. More recently, in [11], Burattin et al. present a framework that allows one to compute conformance indicators on the basis of behavioral patterns described by the given process model. In the context of online concept drift detection, Ostovar et al. [3] propose to detect drifts on event streams by monitoring the distribution of behavioral abstractions (i.e. α^+ relations [18]) of the event stream across adjacent time sliding-windows. In follow-up work [4], Ostovar et al. extend [3] to allow for concept drift characterization.

With respect to noise filtering and/or outlier detection in the context of event logs, several approaches are described in literature [19–23]. The approach proposed by Wang et al. [19] relies on a reference process model to repair a log whose events are affected by labels that do not match the expected behavior of the reference model. The approach proposed by Conforti et al. [20] removes events that cannot be reproduced by an automaton constructed using frequent process behavior recorded in the log. In [21], Fani Sani et al. propose an approach that uses conditional probabilities between sequences of activities to remove events that are unlikely to occur in a given sequence. In [22], Fani Sani et al. propose to repair infrequent trace fragments by more frequent fragments on the basis of conditional probabilities occurring in the log. Finally, in [23], Fani Sani et al. propose to exploit (in)frequent patterns, i.e. as defined in the domain of *sequence mining*, in order to identify potential outlier behavior.

Existing noise filtering techniques have shown to improve the quality of process mining techniques, yet they are not directly applicable in an online context. Similarly, online process mining techniques do not address the problem of noise in event streams. The technique presented in this paper, therefore, bridges the gap between these techniques.

3. Background

In this section, we present basic preliminary background material, covering the notions of event logs, event streams and probabilistic automata.

3.1. Basic notation

Let X denote an arbitrary set. We let $\mathcal{P}(X)$ denote the power set of X, i.e. $\mathcal{P}(X) = \{X' \mid X' \subseteq X\}$. We let $\mathbb{N} = \{1, 2, \ldots\}$ denote the positive integers, $\mathbb{N}_0$ includes 0. We let $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ denote the set of Boolean values. A *binary relation* R on set X, written as (X, R), is a set of ordered pairs of elements of X, i.e. $R \subseteq X \times X$. In case $(x, y) \in R$, we alternatively write xRy. A *strict partial order* $\prec$ on set X, is a binary relation on X, for which $x \not\prec x$, $x \prec y \implies y \not\prec x$ and $x \prec y \wedge y \prec z \implies x \prec z$ hold, for all $x, y, z \in X$. A *multiset* generalizes the notion of a set, i.e. it allows its members to have multiple appearances. We define a multiset M over a set X as a function $M: X \rightarrow \mathbb{N}_0$. We write a multiset as $[x^i, y^j, \ldots, z^k]$, where $M(x) = i$, $M(y) = j, \ldots, M(z) = k$. If $M(x) = 0$, we omit it from multiset notation, and, if $M(x) = 1$, we omit its superscript, e.g. $[x^2, y]$ contains 2 times x, one y element and zero z elements. We let $\mathcal{M}(X)$ denote the universe of all possible multisets over base-set X.

A *sequence* σ of length n over set X, relates n positions to elements of X, i.e. it is a function $\sigma: \{1, 2, \ldots, n\} \rightarrow X$. The set of all possible sequences over set X, of arbitrary length, is denoted as X^*. Given $\sigma \in X^*$, we let $elem: X^* \rightarrow \mathcal{P}(X)$, where $elem(\sigma) = \{x \in X \mid \exists 1 \leq i \leq |\sigma| \, (\sigma(i) = x)\}$. We furthermore let $parikh: X^* \rightarrow \mathcal{M}(X)$, where $parikh(\sigma)(x) = |\{i \in \{1, \ldots, |\sigma|\} \mid \sigma(i) = x\}|$, i.e. the *parikh*-function counts the number of occurrences of a certain element $x \in X$, in the given sequence σ.

3.2. Event data

As indicated, process mining aims to gain insights into business processes for their improvement, by means of analyzing the data generated during process execution. Modern information systems track, often in great detail, what specific activity is

Table 1
Example event log fragment.

Event-id	Case-id	Activity	Resource	Time-stamp
…	…	…	…	…
6711	*4123*	*decide (e)*	*Fred*	*2018-12-10 13:47*
6712	*4173*	*register request (a)*	*Wilma*	*2018-12-10 14:14*
6713	*4123*	*reject request (g)*	*Fred*	*2018-12-10 14:18*
6714	*4173*	*examine cause (b)*	*Barney*	*2018-12-10 14:33*
6715	*4173*	*check ticket (d)*	*Betty*	*2018-12-10 14:06*
6716	*4173*	*decide (e)*	*Fred*	*2018-12-10 14:51*
6717	*4173*	*pay compensation (f)*	*Betty*	*2018-12-10 15:03*
6718	*5043*	*register request (a)*	*Wilma*	*2018-12-10 15:05*
…	…	…	…	…

performed for a running instance of the process (also referred to as a *case*) at a certain point in time. Traditional process mining techniques aim to analyze such data, i.e. *event logs*, in a static/a-posteriori setting, i.e. for completed cases only. Consider Table 1, depicting an example of an event log, related to a fictional business process for the purpose of filing a compensation request for concert tickets.

Each line in Table 1, refers to the execution of an activity, i.e. an *event*, in the context of a process instance, which is identified by means of a *case-id*. In this example, the case-id equals the id of the ticket for which a compensation request is filed. In general, the case-id depends on the process under study, e.g. a customer type or product-id are often used as a case-id.

Consider the events related to case-id 4173. The first event, i.e. with id 6712, describes that *Wilma* executed a *register request* activity. Subsequently, *Barney* performed a *causal examination* of the request (event 6714). In-between event 6712 and 6714, event 6713 is executed that relates to a case with id 4123, which reflects that several process instances run in parallel. The next activity executed for case-id 4713 is the *check ticket* activity, executed by *Betty*. A *decision* is made for the case by *Fred*, after which *Betty* handles the *compensation payment*.

As reflected by Table 1, an event describes a multitude of different data attributes, e.g. the *event-id*, *case-id*, *activity*, *resource* and *time-stamp*. However, we primarily focus on the *control-flow dimension*, i.e. the sequential ordering of activities in the context of the different process instances. Therefore, let $\mathcal{E}$ denote the universe of events, $\mathcal{C}$ the universe of case identifiers and $\mathcal{A}$ the universe of activities. We assume that we are able to assess the case identifier and executed activity of an event by means of two projection functions, i.e. $\pi_{\mathcal{C}}: \mathcal{E} \rightarrow \mathcal{C}$ and $\pi_{\mathcal{A}}: \mathcal{E} \rightarrow \mathcal{A}$, respectively. For example, for the first event of Table 1 (which we write as e_{6711}), we have $\pi_{\mathcal{C}}(e_{6711}) = 4123$ and $\pi_{\mathcal{A}}(e_{6711}) =$ decide. We formalize the notion of event logs, as well as traces, i.e. as typically used in offline process mining, in Definition 1.[1]

Definition 1 (*Event, Event Log, Trace*)**.** Let $\mathcal{E}$ denote the universe of events, let $\mathcal{A}$ denote the universe of activities and let $\mathcal{C}$ denote the universe of case identifiers. An *event* $e \in \mathcal{E}$ describes the (partial-)execution of an activity, in the context of some process instance. Projection functions $\pi_{\mathcal{A}}: \mathcal{E} \rightarrow \mathcal{A}$ and $\pi_{\mathcal{C}}: \mathcal{E} \rightarrow \mathcal{C}$, allow us to access the activity and case identifier, respectively, of an event e. Given a collection of events $E \subseteq \mathcal{E}$, an *event log* L is a strict partial order of events, i.e. $L = (E, \prec)$. A *trace* related to case $c \in \mathcal{C}$ is a sequence $\sigma \in E^*$ for which:

1. $\forall 1 \leq i \leq |\sigma| (\pi_{\mathcal{C}}(\sigma(i)) = c)$; Events in σ relate to case c.
2. $\forall e \in E (\pi_{\mathcal{C}}(e) = c \Rightarrow \exists 1 \leq i \leq |\sigma| (\sigma(i) = e))$; Each event related to c is in σ.
3. $\forall 1 \leq i < j \leq |\sigma| (\sigma(i) \neq \sigma(j))$; All events in σ are unique.
4. $\forall 1 \leq i < j \leq |\sigma| (\sigma(j) \not\prec \sigma(i))$; Events in σ respect their order.

The strict partial order of the events of an event log is typically imposed by means of recorded times-stamps. A log is a strict partial order due to, for example, inherent parallelism of the process and/or mixed time-stamp granularity. A *trace* is a sequence of events related to the same case identifier which respects the strict partial order. Reconsider case 4173 in Table 1, which we write as a *trace* as $\langle$(6712, 4173, register request), (6714, 4173, examine cause), (6715, 417, check ticket), (6716, 4173, decide), (6717, 4173, pay compensation)$\rangle$, or simply $\langle$(6712, 4173, a), (6714, 4173, b), (6715, 4173, d), (6716, 4173, e), (6717, 4173, f)$\rangle$ using short-hand activity names. Using the control-flow perspective, the example trace simply represents the activity sequence $\langle a, b, d, e, f \rangle$. Note that, when adopting the control-flow perspective, a multitude of cases exist which project onto the same sequence of activities.

We adopt the notion of online/real-time *event stream*-based process mining, in which the data is assumed to be an infinite sequence of events. Since in practice, several instances of a process run in parallel, we have no guarantees w.r.t. the arrival of the events related to the same case. Thus, the events related to a certain case of the process are likely to be emitted onto the stream in a dispersed manner.

Definition 2 (*Event Stream*)**.** An event stream S is a (possibly infinite) sequence of unique events, i.e. $S \in \mathcal{E}^*$ s.t. $\forall 1 \leq i < j \leq |S| (S(i) \neq S(j))$.

Consider Fig. 1, in which we depict a few of the (short-hand) events that are also presented in Table 1.

The first event depicted represents (6711, 4123, decide), the second event is (6712, 4173, register request), and so on. We assume that one event arrives per unit of time. Moreover, we assume that the order of event arrival corresponds to the order of execution.

3.3. Probabilistic automata

The filtering approach, presented in this paper, builds on top of the notion of *probabilistic automata* (*PA*). Such automata extend *conventional non-deterministic automata*, i.e. a basic mathematical model describing a collection of (possibly accepting) states and associated transitions that allow us to change the state of the model. Probabilistic automata allow us to assign a probability of occurrence to each transition in the model.

Definition 3 (*Probabilistic Automaton*)**.** A probabilistic automaton (*PA*) is a 6-tuple $(Q, \Sigma, \delta, q_0, F, \gamma)$, where Q is a finite set of states, Σ is a finite set of symbols, $\delta: Q \times \Sigma \rightarrow \mathcal{P}(Q)$ is a transition relation, $q^0 \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states and $\gamma: Q \times \Sigma \times Q \rightarrow [0, 1]$ is the transition probability function.

Additionally we require:

1. $\forall q, q' \in Q, a \in \Sigma \left(q' \in \delta(q, a) \Leftrightarrow \gamma(q, a, q') > 0\right)$: if an arc labeled a connects q to q', then the corresponding probability is non-zero (and vice-versa).
2. $\forall q \in Q \setminus F \left(\exists q' \in Q, a \in \Sigma \left(q' \in \delta(q, a)\right)\right)$: non-accepting states have outgoing arc(s).
3. $\forall q \in Q \setminus F \left(\sum_{\{(a,q') \in \Sigma \times Q | q' \in \delta(q,a)\}} \gamma(q, a, q') = 1\right)$: the sum of probabilities of outgoing arcs of a non-accepting state equals one.
4. $\forall q \in F \left(\sum_{\{(a,q') \in \Sigma \times Q | q' \in \delta(q,a)\}} \gamma(q, a, q') < 1\right)$: the sum of probabilities of outgoing arcs of an accepting state is smaller than one.

[1] We omit projection functions to access the time-stamps, resources and/or other types of data attributes, as these are not used in the remainder of the paper.

$$\cdots\ (6711, 4123, e), (6712, 4173, a), (6713, 4123, g), (6714, 4173, b),\ \cdots$$

Fig. 1. Example event stream S.

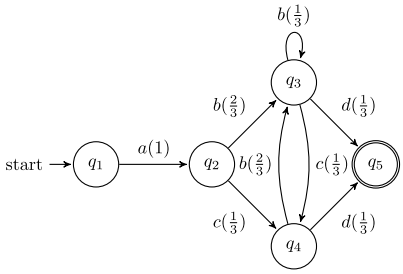


Fig. 2. Example probabilistic automaton. Probabilities are listed in brackets, e.g. $b(\frac{2}{3})$.

For a given state $q \in Q$ and label $a \in \Sigma$, we denote the conditional probability of observing label a, whilst being in state q, as $P(a \mid q)$, where:

$$P(a \mid q) = \sum_{\{q' \in Q \mid q' \in \delta(q,a)\}} \gamma(q, a, q')$$

Consider Fig. 2, in which we depict an example probabilistic automaton.

Observe that, in state q_2, label b occurs with probability $\frac{2}{3}$, whereas label c occurs with probability $\frac{1}{3}$. Furthermore, observe that, according to Definition 3, an *accepting state* is allowed to have outgoing arcs. We accordingly define the probability of not observing any label for such an accepting state q as $P(\varnothing \mid q) = 1 - \sum_{a \in \Sigma} P(a \mid q)$.

4. Filtering infrequent behavior from event streams

In this section, we present a probabilistic automaton-based approach to identify and filter infrequent behavior from event streams. We first present the general architecture of the proposed filter, after which we explain in detail how to use probabilistic automata for the purpose of control-flow oriented event filtering.

4.1. Architecture

In this section, we present the basic architecture of the proposed event filter. Consider Fig. 3, in which we depict a high-level overview of the architecture of the filter. We assume that the input event stream S contains both proper and noisy behavior. In this context, noisy behavior is referred to as events that are improperly executed, i.e., events that are observed on the stream but should not have been observed.

We maintain a sliding window w on the basis of the input stream. A new event e, arriving at the event stream, is immediately incorporated in the behavioral representation of the stream, as maintained by the event filter f. *As such, the filter f always reflects the behavior captured within the sliding window w.* Thus, when events are removed from the underlying sliding window, as caused by the addition of the newly received event e, i.e. visualized by event e'' in Fig. 3, we process the event removal within the filter.

We propose to instantiate filter f by constructing an *ensemble of different probabilistic automata*, which describe the behavior captured within the sliding window.[2] A state within a probabilistic automaton maintained by the filter, represents *a view* on recently observed behavior of a specific process instance as represented by its corresponding case identifier. For example, a state of an automaton can represent the three most recent activities performed for a certain process instance. The outgoing transition probabilities of a state are based on observed behavior for that state, as temporarily described by the sliding window. When applying filtering on an event, we assess the state of the process instance described by the event and check, based on the distribution as defined by that state's outgoing arcs, whether the new event is infrequent, according to the probability distribution of that state.

Note that, actually filtering an event is optionally delayed, i.e. the events are temporarily buffered in some secondary storage component, prior to be evaluated in the filter. Finally, after such optional delay, the filter f either decides to emit the event onto output stream S', or, to discard it. Observe that, each event, eventually filtered or not, is always incorporated in the internal representation of the filter. We mainly do so because of the fact that typically, concept drift, i.e. changes in the predominant underlying distribution of behavior, initially seem to relate to outlier behavior and only over time they become mainstream behavior

4.2. Constructing prefix-based automata

As indicated, we instantiate filter f by constructing an *ensemble of different probabilistic automata*, on the basis of the behavior present in the underlying sliding window. The states in the automata represent different views on recently observed behavior of a specific process instance. In the remainder of this section, we describe how to construct probabilistic automata on the basis of event streams.

As an example of maintaining probabilistic automata on the basis of observed process behavior, consider Fig. 4(a).

In the two example automata, for each observed process instance, we obtain the corresponding state by applying the *elem* function on the two most recently received events, e.g. if we have observed $\langle a, b, c \rangle$ for some process instance, the corresponding state is $elem(\langle b, c \rangle) = \{b, c\}$. Since we use a view size of 2, we only obtain meaningful states if a trace is at least of length 2. Therefore, from the initial state, we use a uniform distribution of τ-labeled transitions to states $\{a, c\}$ and $\{a, b\}$ respectively. The automaton depicted in Fig. 4(a) is based on four traces of observed behavior, i.e. $\langle a, b, c, d, e \rangle$, $\langle a, c, b, d, f \rangle$, $\langle a, c, d, e \rangle$ and $\langle a, c \rangle$. Note that some of these traces, e.g., $\langle a, c \rangle$, are likely to be incomplete, i.e., this is an inherent property of stream based process mining. Therefore, within the automaton, we only mark a state as being accepting, if it has no outgoing arcs. For each state within the example automaton, based on previously observed process instances, we record the probability of observing a certain activity.

The probabilistic automata that we construct, contain states that represent recent control-flow oriented behavior for the process instances currently captured within the sliding window. As such, each state refers to a (partial) prefix of the process instance's most recent behavior, and hence, we deem these automata *prefix-based automata*. Therefore, in prefix-based automata, a state q represents an abstract view on a *prefix of executed activities*, whereas outgoing arcs represent those activities $a \in \mathcal{A}$ that are likely to follow the prefix represented by q, and their associated probability of occurrence. We define two types of parameters, that allow us to deduce the exact state in the corresponding prefix automaton based on a prefix, i.e.:

[2] Note that, the automata used in this paper can be regarded as extended/decorated variants of the transition systems described in [24].

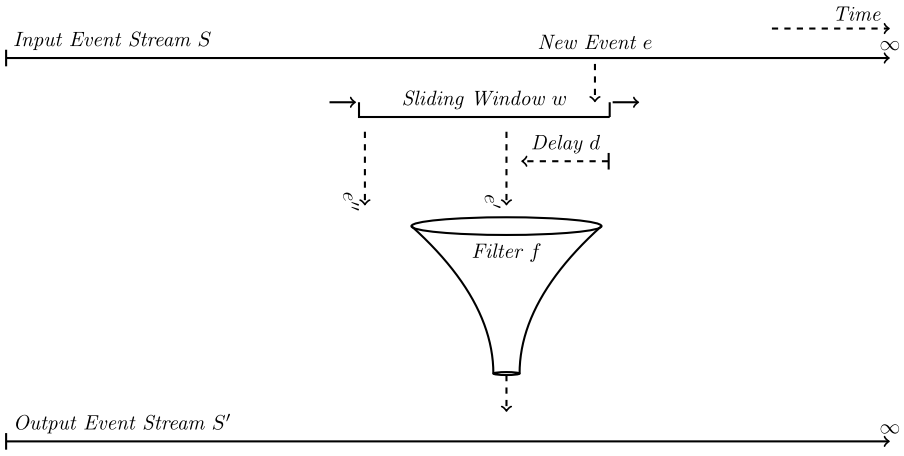
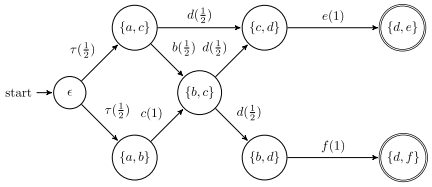
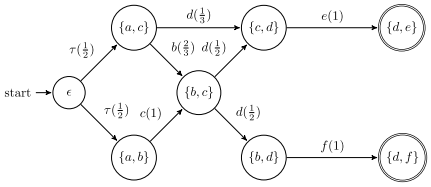


Fig. 3. Schematic overview of the proposed filtering architecture.



(a) Example automaton based on four (possibly incomplete) observed fragments of traces, i.e. $\langle a, b, c, d, e\rangle$, $\langle a, c, b, d, f\rangle$, $\langle a, c, d, e\rangle$ and $\langle a, c\rangle$.



(b) Similar automaton to Figure 4a, after receiving a new event b for the (incomplete) trace $\langle a, c\rangle$ (yielding trace $\langle a, c, b\rangle$). As a consequence, the outgoing arcs of state $\{a, c\}$ get different probabilities.

Fig. 4. Example of maintaining a prefix-based automaton, using a window size of 1 (the same for all abstractions).

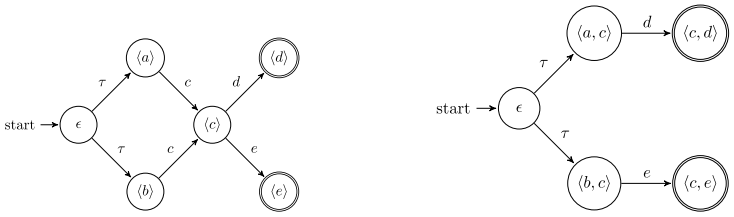
1. *Maximal abstraction view size*: Represents the size of the prefix to take into account when constructing states in the automaton. For example, if we use a maximal size of 5, we only take into account the five most recent events present in the sliding window for the process instance under consideration as being recent behavior.
2. *Abstraction*: Represents the abstraction that we apply on top of the derived recent historical behavior, i.e. subject to the *maximal abstraction view size* parameter, in order to define a state. We propose three abstractions:
   - *Identity*: Given view size $k\in\mathbb{N}$ and a trace $\sigma\in\mathcal{A}^*$, the identity abstraction $\vec{id}^k$ yields the prefix as a state, i.e. $\vec{id}^k : \mathcal{A}^* \rightarrow \mathcal{A}^*$, where $\vec{id}^k(\sigma) = \langle\sigma(|\sigma| - (k + 1)), \ldots \sigma(|\sigma|)\rangle$.
   - *Set*: Given view size $k\in\mathbb{N}$ and a trace $\sigma\in\mathcal{A}^*$, the set abstraction indicates the presence of $a\in\mathcal{A}$ in the last k elements of σ, i.e. we apply $elem(\langle\sigma(|\sigma| - (k + 1)), \ldots \sigma(|\sigma|)\rangle)$.
   - *Parikh*: Given view size $k\in\mathbb{N}$ and a trace $\sigma\in\mathcal{A}^*$, the Parikh abstraction yields a multiset describing the number of occurrences of $a\in\mathcal{A}$ within σ, i.e. we apply $parikh(\langle\sigma(|\sigma| - (k + 1)), \ldots \sigma(|\sigma|)\rangle)$.

The maximal abstraction view size influences the degree of generalization of the automaton. For example, when using a maximal view size of 1, we obtain a strictly smaller automaton than the one depicted in Fig. 4, allowing for much more behavior. Moreover, for the case of maximal view size 1, each of the abstractions mentioned, i.e. *identity*, *set* and *Parikh*, yields the same automaton. Observe that, increasing the maximal window size is likely to generate automata of larger size, i.e. we are able to distinguish a wider variety of states, and is thus likely to be more memory intensive. Hence, there is a trade-off between precision of the automata w.r.t. training data and memory complexity.

4.3. Incrementally maintaining collections of automata

In this section, we indicate how to maintain a ensemble of automata which we use to filter. Prior to this, we motivate the need for using multiple automata within filtering.

Consider that we observe an event stream, on which we observe only two traces, i.e. $\langle a, c, d\rangle$ and $\langle b, c, e\rangle$. Note that, within the data, there is a *dependency* between, on the one hand, a and d, and, b and e, on the other hand. Consider Fig. 5, in which we present two automata constructed on the basis of the two simple traces. In both automata we use an identity abstraction, yet in Fig. 5(a), we use a maximal view size of 1, whereas in Fig. 5(b), we use a maximal view size of 2. For simplicity, we have omitted the probabilities of the edges of the automata. Note that, when only using the automaton depicted in Fig. 5(a), we do not observe the dependency. As a result, whenever an event describes the occurrence of activity e after earlier observed prefix $\langle a, c\rangle$, we are not able to identify this as being infrequent, i.e. both $\langle a, c\rangle$ and $\langle b, c\rangle$ are translated into state $\langle c\rangle$. In the automaton in Fig. 5(b), this is however possible. Hence, we aim to use automata using different window sizes, which allows us to generalize on the one hand (smaller window sizes), yet, also allows us to detect certain long-distance patterns (larger window sizes).



(a) Automaton (probabilities omitted) describing the behavior of the process, i.e. traces $\langle a, c, d\rangle$ and $\langle b, c, e\rangle$, using a window size of 1.

(b) Automaton (probabilities omitted) describing the behavior of the process, i.e. traces $\langle a, c, d\rangle$ and $\langle b, c, e\rangle$, using a window size of 2.

Fig. 5. Two automata, using different window lengths, i.e. length 1 and 2, describing the behavior of the process, i.e. traces $\langle a, c, d\rangle$ and $\langle b, c, e\rangle$. Only using a window length of 2 allows us to observe the *long-term dependencies* as described by the data.

As new events are emitted on the stream, we aim to keep the automata up-to-date in such way that they reflect the behavior present in the sliding window. For each case identifier $c \in \mathcal{C}$, after receiving i events, we keep track of the events present in the underlying sliding window at corresponding time i, i.e. we denote this sequence as $\sigma_c^i \in \mathcal{E}^*$. Note that we assume the order of the events in σ_c to comply with the order of the events as stored in the underlying sliding window w. Let $k > 0$ represent the maximal abstraction view size we take into account when building automata. We maintain k prefix-automata, where for $1 \leq j \leq k$, automaton $PA_j = (Q_j, \Sigma_j, \delta_j, q_j^0, F_j, \gamma_j)$ uses maximal abstraction window size j to define its state set Q_j. Upon receiving a new event, we incrementally update the k maintained automata.

Consider receiving the ith event $S(i) = e$, with $\pi_{\mathcal{C}}(e) = c$ and $\pi_{\mathcal{A}}(e) = a$. To update automaton PA_j, we apply the abstraction of choice on the j preceding events of the newly received event, i.e. $\langle \sigma_c^i(|\sigma_c^i| - j), \ldots, \sigma_c^i(|\sigma_c^i| - 1)\rangle$, to deduce the corresponding previous state $q \in Q_j$. The newly received event influences the probability distribution as defined by the outgoing arcs of state q, i.e. it describes that q can be followed by activity a. Therefore, instead of storing the probabilities of each γ_j, we store weighted outdegree of each state $q_j \in Q_j$, i.e. $\deg_j^+(q_j)$. Moreover, we store the individual contribution of each $a \in \mathcal{A}$ to the outdegree of q_j, i.e. $\deg_j^+(q_j, a)$. Observe that $\deg_j^+(q_j) = \sum_{a \in \mathcal{A}} \deg_j^+(q_j, a)$, and, that deducing the empirical probability of activity a in state q_j is trivial, i.e. $P(a \mid q_j) = \frac{\deg_j^+(q_j, a)}{\deg_j^+(q_j)}$.

As an example of updating a single automaton, reconsider the example automaton in Fig. 4, and consider that we receive an event related to activity b, which in turn belongs to the same case as the trace $\langle a, c\rangle$. Hence, we obtain a new trace $\langle a, c, b\rangle$ for the corresponding case identifier. As we use a window size of 2, in combination with the set abstraction, we deduce the new state in the automaton related to that case is $\{a, c\}$. We observe a total of 3 traces that describe an action out of state $\{a, c\}$, two of which describe activity b. Only one of the traces describes activity d after state $\{a, c\}$. Hence, we deduce empirical probability $\frac{2}{3}$ for activity b and $\frac{1}{3}$ for activity d.

Updating the automata based on an event that is removed from the sliding window, is performed as follows. We let $\Delta_c(i) = |\sigma_c^{i-1}| - |\sigma_c^i|$, $\forall c \in \mathcal{C}$. Again, assume that we receive a new event e at some point in time $i > 0$ related to a process instance identified by some case identifier c. Observe that for any case identifier $c \in \mathcal{C}$, that *does not relate to the newly received event*, we have $\Delta_c(i) \geq 0$. This is the case since events are potentially dropped for such case, yet no new events are received, hence $|\sigma_c^i| \leq |\sigma_c^{i-1}|$. In a similar fashion, for the process instance identified by case c that relates to the new event e, we have $\Delta_c(i) \geq -1$. This is the case since either $|\sigma_c^i| = |\sigma_c^{i-1}| + 1$, or, $|\sigma_c^i| = |\sigma_c^{i-1}|$. Thus, to keep the automata in line with the events stored in the event window, in the former case we need to update the automata if $\Delta_c(i) > 0$, i.e. at least one event is removed for the corresponding case identifier, whereas in the latter case we need to update the automata if $\Delta_c(i) \geq 0$.

To update the collection of k maintained automata, given $j = \min(k, |\sigma_c^{i-1}| - 1)$, we generate sequences $\langle \sigma_c^{i-1}(1)\rangle$, $\langle \sigma_c^{i-1}(1), \sigma_c^{i-1}(2)\rangle, \ldots, \langle \sigma_c^{i-1}(1), \ldots, \sigma_c^{i-1}(j)\rangle$. For each generated sequence, we apply the abstraction of choice to determine corresponding state q, and subsequently reduce the value of $\deg^+(q)$ by 1. Moreover, assume that state q corresponds to sequence $\langle \sigma_c^{i-1}(1), \sigma_c^{i-1}(2), \ldots, \sigma_c^{i-1}(j)\rangle$ with $1 \leq j \leq \min(k, |\sigma_c^{i-1}| - 1)$, we additionally reduce $\deg^+(q, a)$ by 1, where $a = \sigma_c^{i-1}(j+1)$.

4.4. Filtering events

After receiving an event and subsequently updating the collection of automata, we determine whether the new event is spurious or not. To determine whether the newly arrived event is likely to relate to outlier behavior, we assess to what degree the empirical probability of occurrence of the activity described by the new event is an outlier w.r.t. the probabilities of other outgoing activities of the current state. Given the set of k automata, for automaton $PA_j = (Q_j, \Sigma_j, \delta_j, q_j^0, F_j, \gamma_j)$ with prefix-length j ($1 \leq j \leq k$), we characterize an automaton specific filter as $f_j : Q_j \times \Sigma_j \to \mathbb{B}$.[3] Note that an instantiation of a filter f_j often needs additional input, e.g., a threshold value or range. The exact characterization of f_j is a parameter of the approach, however, we propose and evaluate the following instantiations:

- *Fractional*: Considers whether the probability obtained is higher than a given threshold, i.e., $f_j^F : Q_j \times \Sigma_j \times [0, 1] \to \mathbb{B}$, with $f_j^F(q_j, a, \kappa) = 1$ if $P(a \mid q_j) < \kappa$.
- *Heavy Hitter*: Considers whether the probability obtained is higher than a fraction of the maximum outgoing probability, i.e., $f_j^H : Q_j \times \Sigma_j \times [0, 1] \to \mathbb{B}$, with $f_j^H(q_j, a, \kappa) = 1$ if $P(a \mid q_j) < \kappa \cdot \max_{a' \in \mathcal{A}} P(a' \mid q_j)$.
- *Smoothed Heavy Hitter*: Considers whether the probability obtained is higher than a fraction of the maximum outgoing probability subtracted by the uniform distribution over the number of outgoing arcs. Let $NZ = \{a \in \Sigma_j \mid P(a \mid q_j) > 0\}$, we define $f_j^{SH} : Q_j \times \Sigma_j \times [0, 1] \to \mathbb{B}$, with $f_j^{SH}(q_j, a, \kappa) = 1$ if $P(a \mid q_j) < \kappa \cdot \left(\max_{a' \in \mathcal{A}} P(a' \mid q_j) - \frac{1}{|NZ|}\right)$.

[3] It is also possible to have $rng(f_i) = [0, 1]$, i.e., indicating the probability of an event being spurious, however, the filters we propose in this paper all map to Boolean values.

Recall that we are able to optionally *delay* the actual filtering of the event. For an event that we aim to filter, each automaton, combined with a filter of choice yields a Boolean result indicating whether or not the new event is an outlier. In general, we are able to define a weight to each automaton and compute a combined filter score for an event. However, in the remainder, we use two alternative schemes. We symbol an event to be a potential outlier when any of the k maintained automata signals an event to be a potential outlier. Alternatively, we signal an event to be a potential outlier, if the majority of the automata signals this. Finally, note that maintaining/filtering the automata can be performed in parallel, e.g., we maintain an automaton on each node within a cluster.

5. Evaluation

In this section, we evaluate the proposed filter in two ways. First, we assess filtering accuracy on randomly generated event data, based on *synthetic* process models. Second, we assess the applicability of our filter in combination with an existing class of online process mining techniques, i.e., *concept drift detection techniques*. In the latter experiment, we consider both synthetic and real-life datasets.[4] The source code of the filter is available through http://svn.win.tue.nl/repos/prom/Packages/StreamBasedEventFilter.

5.1. Stable process behavior

In these experiments we use a wide variety of stable process behavior, i.e., a stream generated out of a process free of concept drift, with differing levels of noise. We first present the experimental setup, after which we discuss the obtained accuracy results.

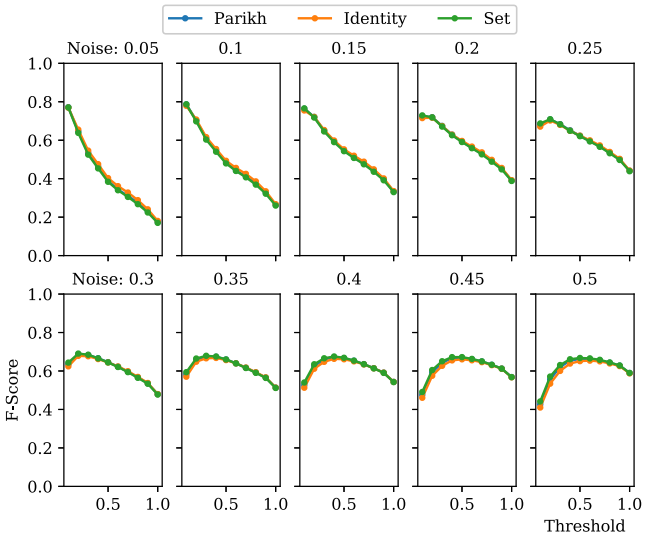
5.1.1. Experimental setup

To evaluate the proposed method on stable process behavior, we automatically generated several process models with different probabilities (ranging from 0 to 0.20 in steps of 0.05) of inserting parallel structures. To obtain a more robust analysis, for each insertion probability, we generated 40 different process models. Afterwards, for each process model, subject to different noise injection probabilities (from 0 to 0.5 with steps of 0.05), we generated an event log containing a total of 3000 traces of process behavior. We additionally checked that the generated noisy events are indeed noise w.r.t. the original model. Consider Table 2, which presents a schematic overview of the experimental setup.

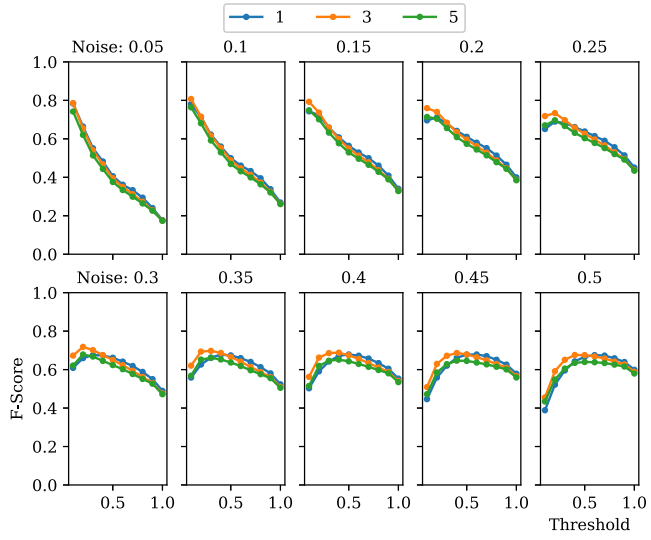
5.1.2. Results

We measured the F-Score obtained by the algorithm, under different parameter configurations. In this context, a *true positive filtering outcome* is defined as a noisy event that is correctly identified as such by the filter. Since the process models and corresponding event logs used in the experiments only describe stable behavior, the impact of applying an emission delay is expected to be negligible. Indeed, upon inspection, the corresponding results show a negligible influence of applying filtering delay on the filtering accuracy under stable event stream behavior.

We assess the influence of four different parameters, as presented in this paper, i.e., the abstraction, maximal view size, filtering technique and the voting scheme used, which we present in Figs. 6 and 7 respectively (excluding noise-free data).



(a) Average F-score per abstraction used.



(b) Average F-score per abstraction view size used.

Fig. 6. Average F1-score for different abstractions and abstraction view sizes. The identity abstraction slightly outperforms the Parikh/set abstraction for small noise levels. Maximum view size of 3 provides an adequate balance between under- and overhitting and outperforms sizes 1 and 5 for smaller threshold values.

Consider Fig. 6(a), in which we present the accuracy results for the different types of abstractions presented in this paper, i.e., the *Parikh*, *identity* and *set* abstraction. For the majority of the results, the Parikh and set abstraction result in the same accuracy values. This is most likely explained by the fact that repeated execution of activities in the underlying event data is limited. For low noise levels (0.05–0.15), combined with high threshold values, we observe that the identity abstraction slightly outperforms both the Parikh and set abstraction, in terms of obtained F-score. Upon inspection, we observe that this is caused by slightly higher precision values for the identity abstraction. Hence, for these lower noise ranges, limited generalizing power of the identity abstraction enables us to more adequately distinguish actual noisy behavior from proper behavior, i.e., by obtaining a slightly lower amount of false negatives. For higher noise levels

[4] As already indicated in [13], in an earlier performed set of experiments, we measured an average event handling time of $\sim$ 0.017 ms, leading to handling $\sim$ 58.8 events per ms., which confirm that automaton-based filtering is suitable to work in real-time/event stream based settings. Therefore, in this paper, we mainly focus on the quality of the filter in terms of filtering accuracy.

Table 2
Parameters of *Data Generation* and *Experiments* with *Synthetic Data*.

Artefact/Parameter	Value
Data generation	
Parallel construct probabilities	{0, 0.05, . . . , 0.20}
Number of models per construct prob.	40
Probability of spurious event injection, per model	{0, 0.05, . . . , 0.5}
Number of traces, generated per model/noise combination	3000
Filter parameters	
Sliding window size	{2500, 5000}
Maximal abstraction view size	{1, 3, 5}
Abstraction	{`Identity`, `Parikh`, `Set`}
Filter	{`Fractional` (f^F), `HeavyHitter` (f^H), `SmoothedHeavyHitter` (f^{SH})}
Filter Threshold (κ)	{0.05, 0.1, . . . , 0.5}
Delay	{0, 500, 1000, 2000}

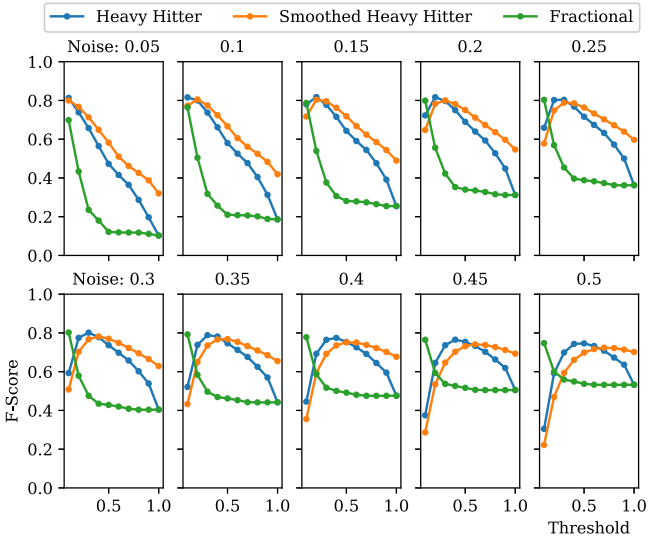
($\geq$ 0.35), combined with low threshold values, we observe that the Parikh and set abstraction slightly outperform the identity abstraction, in terms of obtained F-score. Again, this is due to variety in the precision levels. In this case, the limited generalization power of the identity abstraction leads to reverse effects, i.e., the noise generates infrequent states that have small outgoing distributions, which do no longer allow us to identify noisy behavior.

In Fig. 6(b), we present the accuracy results obtained for the different possible abstraction view sizes used. Interestingly, using a larger size (size 5), i.e., using a larger collection of automata yielding less behavioral generalization, leads to poorer accuracy in the majority of the noise-threshold combinations. We furthermore observe that using a view size of 1 tends to lead to better accuracy results for larger threshold values. Using a window size of 3 tends to lead to better accuracy values for lower threshold values. Hence, we conclude, that using a too large window, having limited generalizing power within the filter, does not allow us to obtain good results. Depending on the threshold used, either the extreme generalizing power of view size 1 (higher threshold values) or the less generalizing view size of 3 (lower threshold values) is appropriate.

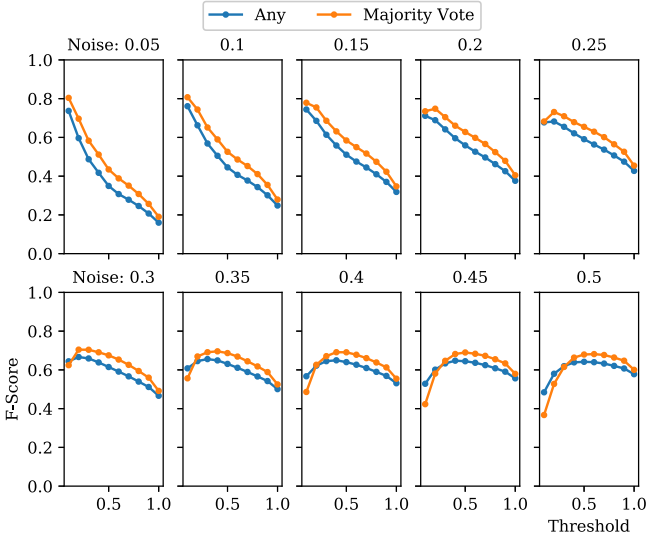
Consider Fig. 7, in which we present accuracy results obtained by using a different filtering technique as well as using different voting mechanisms among the different automata.

In Fig. 7(a), we show the obtained accuracy results for the different filtering techniques proposed in this paper. We observe that the fractional filter leads to suboptimal accuracy in the majority of cases. When observing the underlying recall and precision values, we observe that the low F-measure values are primarily caused by extremely low precision values, as opposed to very high recall values. This is explained by the fact that the fractional filter works on a global level, i.e., anything below the given threshold is considered noise. As such, only very prominent behavior is considered not to be noise, leading to high recall values and low precision values. Both the heavy hitter and smoothed heavy hitter, which can be considered as local filtering techniques, i.e., subject to the state of the respective automaton, perform significantly better. For low levels of noise, the smoothed heavy hitter outperforms the heavy hitter filtering techniques. For higher levels of noise, the smoothed heavy hitter outperforms the heavy hitter filtering technique only for higher threshold values. Upon inspection, we observe that the smoothed heavy hitter filtering technique results in lower recall values, yet higher precision values than the heavy hitter filtering technique.

Finally, in Fig. 7(b), we present the obtained accuracy results by the two different voting mechanisms proposed in this paper. We observe that the majority vote scheme tends to outperform the any scheme, i.e., in which any automaton signaling noise leas to filtering. Upon inspection, this difference is mainly explained

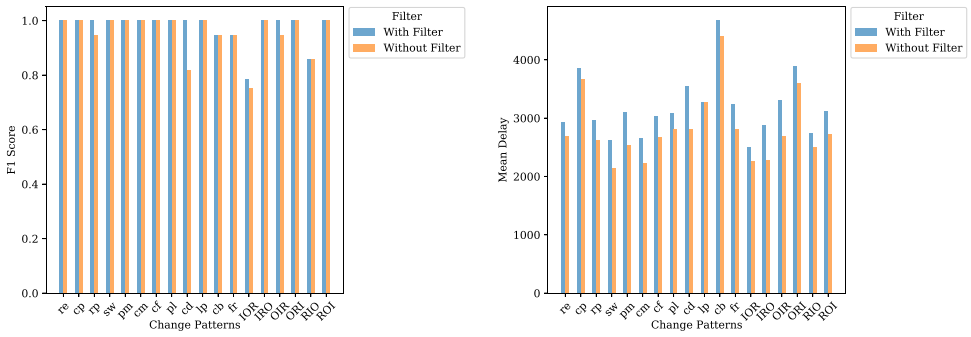


(a) Average F-score per filtering technique applied.

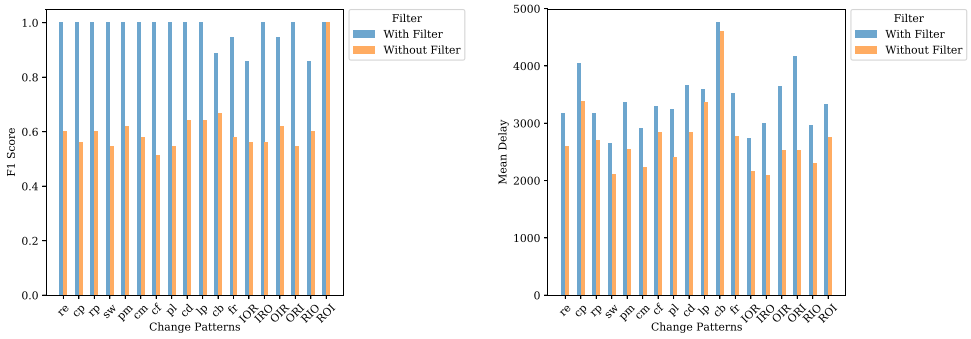


(b) Average F-score per voting scheme adopted.

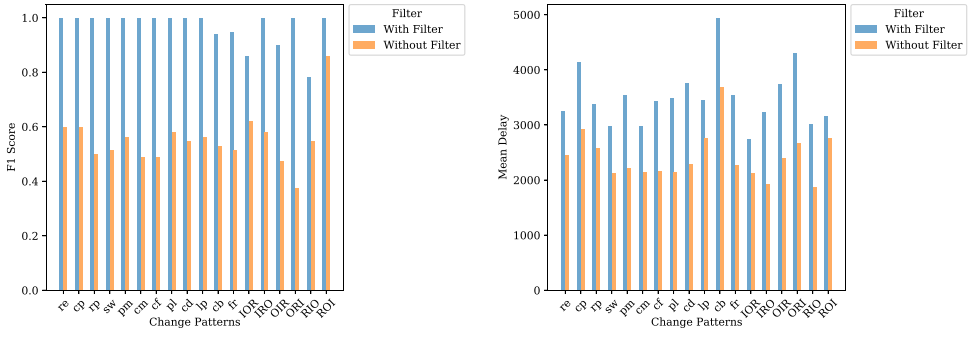
Fig. 7. Average F1-score for different filtering techniques and voting schemes. Smoothed heavy hitter outperforms fractional and heavy hitter filtering in the majority of the cases. Similarly, majority voting outperforms using any negative automaton outcome as a noise classification.



(a) Noise ratio = 0%.



(b) Noise ratio = 2.5%.



(c) Noise ratio = 5%.

Fig. 8. Drift detection F1 score and mean delay (in number of events) per change pattern, obtained from the drift detection technique in [3] over filtered versus unfiltered event streams.

by a large difference in precision, for which the majority voting scheme consistently results in higher values. The alternative scheme consistently leads to higher recall values, yet, the difference is smaller w.r.t. the difference in obtained precision values. Hence, the majority voting scheme seems to allow us to reduce the effect of falsely highlighting proper events as noisy events.

5.2. Filtering with concept drift

In a second set of experiments, we evaluate the impact of our filter on the accuracy of process drift detection. For this, we use a state-of-the-art technique for drift detection that works on event streams [3]. We apply our filter to the event streams generated from a variety of synthetic and real-life logs, with different levels of noise, and compare drift detection accuracy with and without the use of the proposed filter. We first discuss the experimental setup, after which we compare drift detection results obtained with and without the use of our filter.

5.2.1. Experimental setup

For these experiments, we used the 18 event logs proposed in [3] as a basis. The event data are generated by simulating a model featuring 28 different activities (combined with different intertwined structural patterns). Additionally, each event log contains nine drifts obtained by injecting control-flow changes into the model. Each event log features one of the twelve *simple change patterns* [25] or a combination of them. Simple change patterns may be combined through the insertion ("I"), resequentialization ("R") and optionalization ("O") of a pattern. This produces a total of six possible *nested change patterns*, i.e. "IOR", "IRO", "OIR", "ORI", "RIO", and "ROI". For a detailed description of each change pattern, we refer to [3].

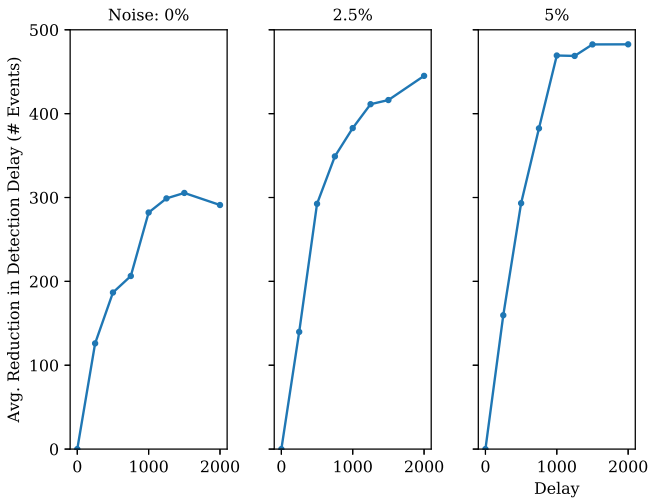


Fig. 9. Average reduction in mean drift detection delay for increasing levels of filtering delays, with different noise percentage levels. Temporarily buffering events has a positive impact on the mean drift detection delay.

Starting from these 18 event logs, we generated 36 additional event logs i.e. two for each original event log. One of the two generated event logs contains 2.5% noise and the other contains 5% of noise. Noise is generated by means of inserting random events into traces of each log. Hence, the final corpus of data consists of 54 event logs, i.e. 12 simple patterns and 6 composite patterns with 0%, 2.5%, and 5% noise, each containing 9 drifts and approximately 250,000 events.

5.2.2. Results on synthetic data

In this experiment, we evaluate the impact of the proposed filter on the accuracy of the drift detection technique proposed in [3]. We use the previously described corpus of data for the experiments. Fig. 8 on page 26 illustrates the F1 score and mean delay of the drift detection, before and after the application of our filter over each change pattern.

The filter, on average, successfully removes 95% of the injected noise, maintaining and even improving the accuracy of the drift detection (with F1 score of above 0.9 in all but two change patterns). This is achieved whilst delaying the detection of a drift by less than 720 events on average (approximately 28 traces).

When considering noise-free event streams (cf. Fig. 8(a)), the filter preserves the accuracy of the drift detection. For some change patterns ("rp", "cd", "IOR", and "OIR"), our filter improves the accuracy of the detection by increasing its precision. This is due to the removal of sporadic event relations, that cause stochastic oscillations in the statistical test used for drift detection. Figs. 8(b) and 8(c) show that noise negatively affects drift detection, causing the F1 score to drop, on average, to 0.61 and 0.55 for event streams with 2.5% and 5% of noise, respectively. This is not the case when our filter is applied, where an F1 score of 0.9 on average is achieved. In terms of detection delay, the filter on average increases the delay by 370, 695, and 1087 events (15, 28, and 43 traces) for the logs with 0%, 2.5%, and 5% noise, respectively. This is the case since changes in process behavior immediately following a drift are treated as noise.

As a final experiment using synthetic data, we investigate the potential effect of temporal buffering of events within the filter, i.e., by means of delaying the actual moment of filtering the received events. We do so under similar levels of noise as used in the experiments reported on in Fig. 8. We present the corresponding results in Fig. 9 In the experiments performed, we measure the impact of using filtering delay of 250, 500, 750, 1000, 1250, 1500 and 2000 events. For reference, when using no delay, the mean drift detection delay under 0% of noise is 2945.6 events, under 2.5% of noise is 3158 events and under 5% of noise is 3303.67 events respectively. Interestingly, we observe that in all cases, the mean drift detection delay, i.e., is reduced when measured on the output stream. At the same time, the average decrease in delay is usually less than the actual filtering delay. Additionally, we observe that the impact of the filtering delay on the drift detection delay stagnates for higher delay values. Furthermore, upon inspection, we observe that in the cases of 2.5% and 5% of noise, using the filtering delay positively contributes to the filtering accuracy in terms of f-score. From these experiments we conclude that the proposed delay in the filter is able to achieve higher filtering accuracy in the presence of both noise and concept drift.

5.2.3. Results on real-life data

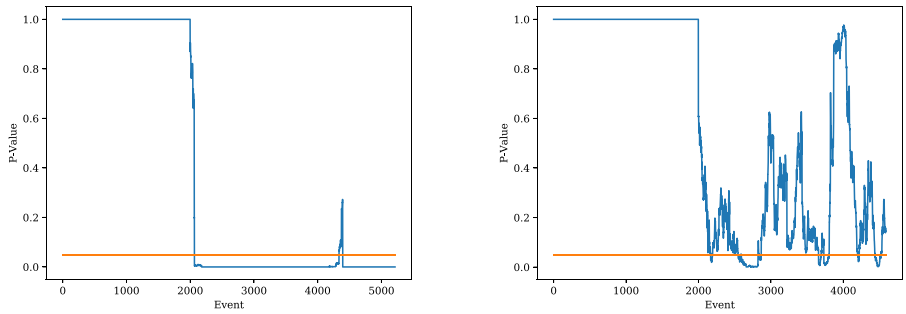
In this experiment, we assess whether the positive effects of our filter on drift detection, observed on synthetic data, translate to real-life data. For this, we used an event log containing cases of Sepsis (a life-threatening complication of an infection) from the ERP system of a hospital [26]. The event log contains 1050 cases with a total of 15,214 events belonging to 16 different activities.

For this experiment, we attempt to detect concept drift over the last 5214 events, as the first 10,000 events are used to train the filter. Fig. 10 plots the significance probability *p-value* curves of the statistical tests used for drift detection, both without (Fig. 10(a)) and with (Fig. 10(b)) the use of our filter. In order to detect a drift, the p-value of the drift detection technique needs to be below a user-specified significance probability threshold, commonly set to 0.05. Moreover, the p-value needs to be lower than the threshold for a given window of ϕ events. In the unfiltered case, cf. Fig. 10(a), we see two clear regions of p-values below the threshold, i.e. after the 2067th event and after the 4373th event. In the case when applying the filter, cf. Fig. 10(b), we observe that there is much more oscillation in the p-value and we do not detect a clear drift.

In the experiments with synthetic logs, we observed that the filter reduced the number of false positives (drift detected when it did actually not occur). To verify if this is also the case for the real-life event log, we profiled the direct-follows dependencies occurring before and after the drifts. The profiling indicates that while direct-follows dependencies "IV Antibiotics $\longrightarrow$ Admission NC" and "ER Sepsis Triage $\longrightarrow$ IV Liquid" are observed several times across the entire event stream, the infrequent direct-follows dependencies "Admission NC $\longrightarrow$ IV Antibiotics" and "IV Liquid $\longrightarrow$ ER Sepsis Triage" appear only in the proximity of the two drifts. These two infrequent dependencies cause a change in the underlying $\alpha+$ relations between the activities, which we use to detect the drifts (in this case changing from causal to concurrent). This change in the relation results in the detection of the drifts. These infrequent dependencies are removed when applying the filter, which in turn does not lead to a clear concept drift. In light of these insights, we can argue that the two drifts detected over the unfiltered event stream are indeed false positives, confirming what we already observed on the experiments with synthetic logs, i.e. that our filter has a positive effect on the accuracy of drift detection.

6. Discussion

As discussed in the related work section, several authors have proposed different process mining specific filtering techniques (in an offline setting). Most of these techniques, like the technique proposed in this paper, learn some model, often probabilistic, of predominant normative behavior and subsequently use that



(a) P-values obtained without applying filtering.

(b) P-values obtained with applying filtering.

Fig. 10. P-values without filtering and with the proposed filter, for the Sepsis event log [26].

model to identify and remove behavior. All these techniques have difficulties in distinguishing *noise*, i.e., true unrelated behavior, from correct yet *infrequent* behavior. Even more so, a proper corresponding taxonomy of noisy behavior and corresponding rationale of idealized filtering behavior in terms of such taxonomy is lacking.

Therefore, we introduce a *control-flow-oriented taxonomy of process behavior* and discuss to what degree our filter can deal with different undesired behavioral categories as described by the taxonomy. Specifically, we identify three major data characteristics, which we then use to classify process execution data.

- *Trustworthiness*: Indicates to what degree the recorded behavior corresponds to reality, i.e. what actually happened during the execution of the process.
- *Compliance*: Indicates to what degree the recorded behavior is in accordance with a predefined set of rules or expectations. These rules may derive from a regulatory framework, business rules or normative process model and dictate that explicit forms of behavior are required. In other cases, service level agreements prescribe certain expectations on the process behavior. When the process behavior complies with such rules/expectations, we consider the behavior as compliant.
- *Frequency*: Indicates the relative frequency of the behavior, i.e. compared to other observed executions of the same process.

In Fig. 11, we present a graphical overview of the different characteristics, and their relation. Depending on the type of process mining task performed someone may desire to include certain types of behavior in the analysis. However, observe that, when behavior is untrustworthy, in general, we are not interested in including it, i.e. we are not able to trust the behavior, and thus draw any significant meaningful conclusions from it. Hence, even in the case of conformance-checking-oriented process mining studies, we aim to remove the untrustworthy behavior. In fact, we aim to omit any form of untrustworthy behavior. However, note that, even though behavior is untrustworthy, systematic errors may cause it to occur frequently. Therefore, in Fig. 11, we do distinguish between frequent and infrequent forms. We always aim to include behavior that is trustworthy, frequent and compliant, while when we perform process discovery, it is most likely that we aim to only include such frequent compliant behavior and leave out any other type of trustworthy behavior. When we apply conformance checking, it is more likely that all trustworthy behavior is required to be included. Observe that in Fig. 11, we explicitly highlight the types of behavior, i.e. infrequent behavior, that the presented filter is able to identify and remove.

It is important to note that, in principle, trustworthiness of behavioral data is often not known. Hence, when applying filtering techniques in practice, either online or offline, it is hard or even impossible to accurately detect untrustworthy behavior. In general, any frequency-based filter only allows us to detect infrequent behavior. Hence, in case non-compliant frequent behavior is present, this is not recognized as eligible to be filtered. Similarly, infrequent yet compliant behavior, is equally likely to be filtered. However, the impact of such filtering behavior is less severe, e.g., in the case of parallelism one does not need to observe all possible interleaving behavior of the parallel construct, to observe the parallel construct itself.

7. Conclusion

The existence of noise in event data typically leads to inaccurate results of process mining techniques. A fraction of noisy behavior in a single trace may already be enough to significantly reduce the accuracy of process mining artefacts such as an automatically discovered process model. While a range of techniques exist for filtering out or tolerating noise in offline settings, online process mining techniques working on event streams are still affected by this problem. As such, such techniques are not reliable in the context of event streams containing noise, which are common in reality. In this paper, we proposed an event stream based filter for online process mining. Our filter relies on an ensemble of probabilistic and non-deterministic automata which are updated dynamically as the event stream evolves. A state in one of these automata represents an abstract view on the recent history of process instances observed from the stream. The empirical probability distribution defined by the outgoing arcs of a state is used to classify new behavior as being spurious (i.e. noise) or not. The time measurements of the corresponding implementation indicate that our filter is suitable to work in real-time settings, i.e. with a response time within one second. Moreover, the experiments on accuracy show that, on a set of stable event streams, we achieve high filtering accuracy for different instantiations of the filter. Finally, we applied our filter to state-of-the-art online drift detection techniques and show that the filter significantly increases the accuracy of these techniques. The filter proposed in this paper allows us to employ a static delay prior to event filtering and construction of the output stream. The initial results with this delay are promising, yet, in practice, concept drift is not expected to occur on a static rate.The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.Hence, an investigation towards a dynamic filter delay which incorporates a concept drift detection

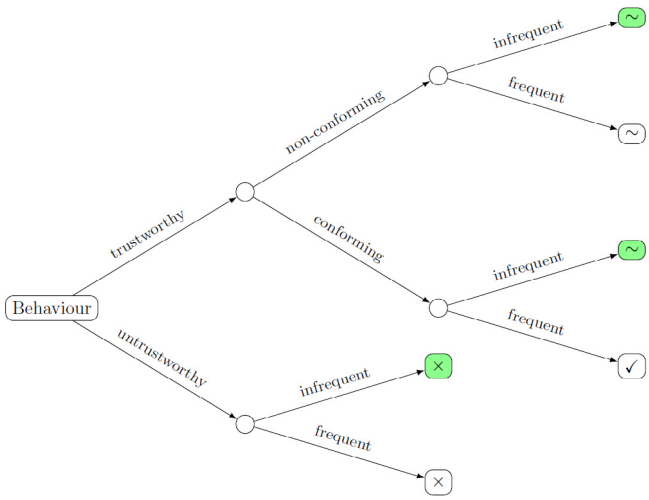


Fig. 11. Control-flow-oriented taxonomy of behavior. The types of behavior that, ideally, are used in process mining are marked with ✓. Behavior that is ideally removed is marked with ×. Behavior of which the type of process mining analysis determines inclusion, is marked with ∼. The types of behavior that are identified by the proposed filter, i.e. infrequent behavior, are highlighted in green. (For interpretation of the references to color in this figure legend, the reader is referred to the digital version of this article.).

component is of interest. In line with this, integrating the filter with an alternative underlying storing mechanism, e.g. dynamic sliding windows, could similarly achieve better results.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This research is funded by the Australian Research Council (grant DP180102839).

References

[1] W.M.P. van der Aalst, Process Mining - Data Science in Action, second ed., Springer, 2016, http://dx.doi.org/10.1007/978-3-662-49851-4.
[2] P. Gonella, M. Castellano, P. Riccardi, R. Carbone, Process Mining: A Database of Applications, two thousand and seventeenth ed., 2017.
[3] A. Ostovar, A. Maaradji, M. La Rosa, A.H.M. ter Hofstede, B.F. van Dongen, Detecting drift from event streams of unpredictable business processes, in: Conceptual Modeling - 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings, 2016, pp. 330–346, http://dx.doi.org/10.1007/978-3-319-46397-1_26.
[4] A. Ostovar, A. Maaradji, M. La Rosa, A.H.M. ter Hofstede, Characterizing drift from event streams of business processes, in: Advanced Information Systems Engineering - 29th International Conference, CAiSE 2017, Essen, Germany, June 12-16, 2017, Proceedings, 2017, pp. 210–228, http://dx.doi.org/10.1007/978-3-319-59536-8_14.
[5] A. Maaradji, M. Dumas, M. La Rosa, A. Ostovar, Detecting sudden and gradual drifts in business processes from execution traces, IEEE Trans. Knowl. Data Eng. 29 (10) (2017) 2140–2154, http://dx.doi.org/10.1109/TKDE.2017.2720601.
[6] M. Hassani, S. Siccha, F. Richter, T. Seidl, Efficient process discovery from event streams using sequential pattern mining, in: IEEE Symposium Series on Computational Intelligence, SSCI 2015, Cape Town, South Africa, December 7-10, 2015, 2015, pp. 1366–1373, http://dx.doi.org/10.1109/SSCI.2015.195.
[7] A. Burattin, A. Sperduti, W.M.P. van der Aalst, Control-flow discovery from event streams, in: Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2014, Beijing, China, July 6-11, 2014, 2014, pp. 2420–2427, http://dx.doi.org/10.1109/CEC.2014.6900341.
[8] S.J. van Zelst, B.F. van Dongen, W.M.P. van der Aalst, Event stream-based process discovery using abstract representations, Knowl. Inf. Syst. 54 (2) (2018) 407–435, http://dx.doi.org/10.1007/s10115-017-1060-2.
[9] A. Burattin, J. Carmona, A framework for online conformance checking, in: Business Process Management Workshops - BPM 2017 International Workshops, Barcelona, Spain, September 10-11, 2017, Revised Papers, 2017, pp. 165–177, http://dx.doi.org/10.1007/978-3-319-74030-0_12.
[10] S.J. van Zelst, A. Bolt, M. Hassani, B.F. van Dongen, W.M.P. van der Aalst, Online conformance checking: Relating event streams to process models using prefix-alignments, Int. J. Data Sci. Anal. (2017) http://dx.doi.org/10.1007/s41060-017-0078-6.
[11] A. Burattin, S.J. van Zelst, A. Armas-Cervantes, B.F. van Dongen, J. Carmona, Online conformance checking using behavioural patterns, in: Business Process Management - 16th International Conference, BPM 2018, Sydney, NSW, Australia, September 9-14, 2018, Proceedings, 2018, pp. 250–267, http://dx.doi.org/10.1007/978-3-319-98648-7_15.
[12] A.E. Márquez-Chamorro, M. Resinas, A. Ruiz-Cortés, Predictive monitoring of business processes: a survey, IEEE Trans. Serv. Comput. 11 (6) (2018) 962–977, http://dx.doi.org/10.1109/TSC.2017.2772256.
[13] S.J. van Zelst, M. Fani Sani, A. Ostovar, R. Conforti, M. La Rosa, Filtering spurious events from event streams of business processes, in: Advanced Information Systems Engineering - 30th International Conference, CAiSE 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings, 2018, pp. 35–52, http://dx.doi.org/10.1007/978-3-319-91563-0_3.
[14] V. Chandola, A. Banerjee, V. Kumar, Anomaly detection for discrete sequences: A survey, IEEE Trans. Knowl. Data Eng. 24 (5) (2012) 823–839, http://dx.doi.org/10.1109/TKDE.2010.235.
[15] M. Gupta, J. Gao, C.C. Aggarwal, J. Han, Outlier detection for temporal data: A survey, IEEE Trans. Knowl. Data Eng. 26 (9) (2014) 2250–2267, http://dx.doi.org/10.1109/TKDE.2013.184.
[16] A.J.M.M. Weijters, J.T.S. Ribeiro, Flexible Heuristics Miner (FHM), in: Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining, CIDM 2011, Part of the IEEE Symposium Series on Computational Intelligence 2011, April 11-15, 2011, Paris, France, 2011, pp. 310–317, http://dx.doi.org/10.1109/CIDM.2011.5949453.
[17] A. Burattin, M. Cimitile, F.M. Maggi, A. Sperduti, Online discovery of declarative process models from event streams, IEEE Trans. Serv. Comput. 8 (6) (2015) 833–846, http://dx.doi.org/10.1109/TSC.2015.2459703.
[18] A.K.A. de Medeiros, B.F. van Dongen, W.M.P. van der Aalst, A.J.M.M. Weijters, Process mining for ubiquitous mobile systems: An overview and a concrete algorithm, in: Ubiquitous Mobile Information and Collaboration Systems, Second CAiSE Workshop, UMICS 2004, Riga, Latvia, June 7-8, 2004, Revised Selected Papers, 2004, pp. 151–165, http://dx.doi.org/10.1007/978-3-540-30188-2_12.
[19] J. Wang, S. Song, X. Lin, X. Zhu, J. Pei, Cleaning structured event logs: A graph repair approach, in: 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015, 2015, pp. 30–41, http://dx.doi.org/10.1109/ICDE.2015.7113270.
[20] R. Conforti, M. La Rosa, A.H.M. ter Hofstede, Filtering out infrequent behavior from business process event logs, IEEE Trans. Knowl. Data Eng. 29 (2) (2017) 300–314, http://dx.doi.org/10.1109/TKDE.2016.2614680.
[21] M. Fani Sani, S.J. van Zelst, W.M.P. van der Aalst, Improving process discovery results by filtering outliers using conditional behavioural probabilities, in: Business Process Management Workshops - BPM 2017 International Workshops, Barcelona, Spain, September 10-11, 2017, Revised Papers, 2017, pp. 216–229, http://dx.doi.org/10.1007/978-3-319-74030-0_16.

[22] M. Fani Sani, S.J. van Zelst, W.M.P. van der Aalst, Repairing outlier behaviour in event logs, in: Business Information Systems - 21st International Conference, BIS 2018, Berlin, Germany, July 18-20, 2018, Proceedings, 2018, pp. 115–131, http://dx.doi.org/10.1007/978-3-319-93931-5_9.

[23] M. Fani Sani, S.J. van Zelst, W.M.P. van der Aalst, Applying sequence mining for outlier detection in process mining, in: On the Move to Meaningful Internet Systems. OTM 2018 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, October 22-26, 2018, Proceedings, Part II, 2018, pp. 98–116, http://dx.doi.org/10.1007/978-3-030-02671-4_6.

[24] W.M.P. van der Aalst, V.A. Rubin, H.M.W. Verbeek, B.F. van Dongen, E. Kindler, C.W. Günther, Process mining: A two-step approach to balance between underfitting and overfitting, Soft. Syst. Model. 9 (1) (2010) 87–111, http://dx.doi.org/10.1007/s10270-008-0106-z.

[25] B. Weber, M. Reichert, S. Rinderle-Ma, Change patterns and change support features - enhancing flexibility in process-aware information systems, Data Knowl. Eng. 66 (3) (2008) 438–466, http://dx.doi.org/10.1016/j.datak.2008.05.001.

[26] F. Mannhardt, Sepsis Cases - Event Log, Eindhoven University of Technology, 2016, http://dx.doi.org/10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460.