Real-Time Data Mining for Event Streams

Conference Paper · October 2018

DOI: 10.1109/EDOC.2018.00025

4 authors, including:

Raphaël Khoury
University of Québec in Chicoutimi
30 PUBLICATIONS **109** CITATIONS
SEE PROFILE

Sylvain Hallé
University of Québec in Chicoutimi
125 PUBLICATIONS **632** CITATIONS
SEE PROFILE

Real-Time Data Mining for Event Streams

Massiva Roudjane, Djamal Rebaïne, Raphaël Khoury, Sylvain Hallé
Laboratoire d'informatique formelle
Université du Québec à Chicoutimi, Canada

***Abstract*—We show how trends of various kinds can be computed over event logs in real time, using a generic framework called the *trend distance workflow*. Many common computations on event streams turn out to be special cases of this workflow, depending on how a handful of workflow parameters are defined. This process has been implemented and tested in a real-world event stream processing tool, called BeepBeep. Experimental results show that deviations from a reference trend can be detected in realtime for streams producing up to thousands of events per second.**

I. Introduction

Information systems can produce logs of various kinds. For instance, workflow management systems, CRM systems and ERP platforms produce event logs in some common format based on XML. Financial transaction systems also keep a log of their operations in some standardized and documented format, as is the case for web servers such as Apache and Microsoft IIS. Network monitors also receive streams of packets whose various headers and fields can be analyzed. Recently, even the world of video games has seen an increasing trend towards the logging of players' realtime activities.

Analyzing the wealth of information contained in these logs can serve multiple purposes. Business process logs can be used to reconstruct a workflow based on a sample of its possible executions [45]; financial database logs can be audited for compliance to regulations [1]; suspicious or malicious activity can be detected by studying patterns in network or server logs [47]. This analysis can take multiple forms: comparing event sequences to a reference workflow, evaluate a set of rules or formal properties on the stream of events, or watching the occurrence of some predefined pattern or value. However, log analysis has recently started to involve elements of machine learning or data mining: users are increasingly interested in finding *patterns* emerging from datasets, and which could provide insight on the process being logged, and even determine the course of future actions.

This paper concentrates on a single particular problem related to event logs and data mining. More precisely, it focuses on the task of computing trends over a sequence of events, and detecting when a sequence veers away from a given reference trend. Section II shall give a few examples of what such "trends" can be, and what deviating from a trend means, depending on the context. An important aspect of all these scenarios is that the deviation should be computed in a *streaming* fashion: an arbitrary source generates events progressively, and an eventual deviation should be detected as soon as it happens.

This mode of operation is in contrast with more traditional data mining approaches working in batch mode on pre-recorded logs, and which generally compute a verdict after the fact. We shall see in Section III that few of the existing solutions provide the appropriate mix of realtime stream processing and data mining functionalities. This is why, in Section IV, we propose a new computing process called the *trend distance workflow*. This workflow defines a fixed sequence of computing steps to be performed on an event stream; however, the model is very generic, and abstracts away most of its functionalities through user-definable parameters. It turns out that many common event stream processing and mining tasks become particular cases of this workflow, depending on how these parameters are defined.

This workflow and some of its derivatives have been concretely implemented as an extension to an existing event stream processing library called BeepBeep. In Section V, we present this implementation, and evaluate its performance experimentally by measuring its runtime according to various combinations of parameters corresponding to common event processing models. These experiments reveal that realtime computation of trends and detection of trend deviations on event logs is feasible, up to throughputs in the range of thousands of events per second on commodity hardware.

II. Trends and Patterns in Event Logs

In this section, we describe a few simple use cases where it may become desirable to process event logs of various kinds and extract "trends" from the data they contain.

A. Business Process Logs

As a first example, consider a simple business process describing the handling of a request for compensation within an airline (Figure 1, taken from van der Aalst [44]). A log in such a scenario would typically consist of events belonging to multiple *instances* of the process in various stages of completion. Assuming that each event contains, among other things, an identifier for the process instance they belong to, it becomes possible to fan out that log into multiple "slices", each of which keeping only the events relevant to a single instance identifier.

One possible task that can be executed on each of these slices is to verify basic compliance to business process rules; one example of such a rule could be "a request must be examined thoroughly if the requested amount is greater than $100". These rules can be expressed in a variety of widely studied notations, and the process of verifying them is called compliance checking (e.g. [3], [16], [19], [31], [41]). However, it may also be interesting to compute and watch various other trends over these slices, and in particular, be notified when a log deviates from some reference trend given beforehand. For example:

Trend Query 1. Be notified when the duration of a request (from registration to decision) is more than 25% longer than some fixed bound B.

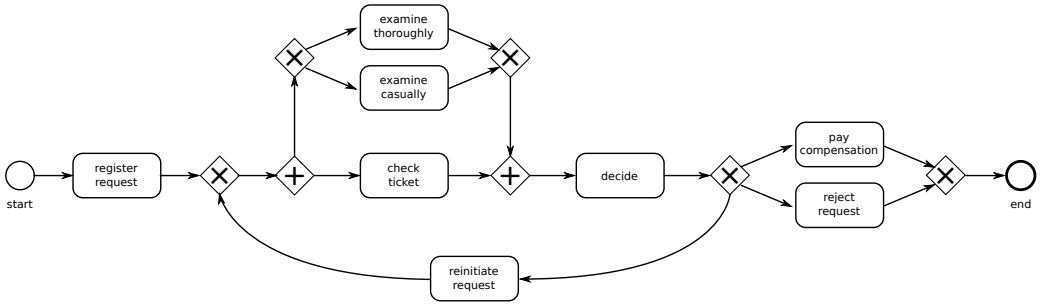


Figure 1: A simple representation of a request for compensation process using the BPMN notation (from [44]).

Contrary to regular compliance rules, such a notification would not necessarily indicate a strict "violation", but rather an indication that something out of the ordinary may be happening, and require further attention. For example, a special message could be sent to the customers, indicating a delay in the processing of their order. The border between strict compliance and a mere "notification" may be further blurred, by replacing a fixed reference with a trend computed from the past history of the process:

Trend Query 2. Be notified when the duration of a request is more than 25% longer than the average duration of the requests from the past month.

Notice how this time, the reference trend is not based on some fixed bound, but is rather computed from a set of previously recorded instances of the same process. Of course, similar computations could be done on aspects of the process other than duration: the number of times it is re-initiated, the amount given to the client, etc.

B. Electronic Medical Records

We now move to the field of medical record management, where events are messages expressed in a structured format called HL7 [42]. HL7 messages can be produced from various sources: medical equipment producing test results, patient management software where individual medical acts and procedures are recorded, drug databases, etc. For a given patient, the merging of all these various sources produces a long sequence of HL7 messages that can be likened to an event stream. The analysis of HL7 event traces produced by health information systems can be used, among other things, to detect significant unexpected changes in data values that could compromise patient safety [7]. In this context, a general rule, which can apply to any numerical field, identifies whenever a data value starts to deviate from its current trend:

Trend Query 3. Notify the user when an observed data field is three standard deviations above or below its mean.

Such queries can be made more complex, and correlate values found in multiple events, such as the following:

Trend Query 4. Notify the user when two out of three successive data points lie more than two standard deviations from the mean on the same side of the mean line.

Although our example query does not specify it, this aggregation can be computed over a "sliding window", such as the past 100 events, or events of the past hour. This time, note how the queries do not relate the log's trend to a fixed reference, or even to a trend computed over a set of past logs. Rather, we attempt to compare current values to a trend computed from a history of the log itself; we shall see later that this process is called *self-correlation*.

C. Systems Security

The computation of trends on event feeds can also be useful for security purposes, by processing logs coming from different sources. As a first example, one can consider a realtime capture of network packets as a stream of events, and monitor deviations of various metrics calculated over that stream [27]. A system can keep track of the TCP ports occurring in captured packets over a certain period of time, and aggregate these ports into a set. This set can then be compared against some reference, and an alarm can be triggered when the two sets differ enough. This can lead to a trend query such as:

Trend Query 5. Send a notification when the set of TCP ports seen over the last hour differs of at least 5% from some reference set of ports S.

Assuming that S is a sample taken from traffic verified as "normal", a difference may indicate that suspicious activity is occurring over the network. This time, note how the trend that is computed on the stream is not a numerical value, but a set of discrete elements (TCP ports). This is one more example where self-correlation can be used: instead of computing a static reference set S, one could use a set of ports observed at some point in the past (e.g. last hour, or same hour the day before).

Other system logs can be processed in a similar way. For example, Microsoft Active Directory maintains a log of all authenticated sessions across a network of Windows workstations [14]. Login/logout events from the same user or process ID can be aggregated into *sessions*, and trends over these sessions can be used to detect unusual or suspicious activity. For example, sessions can be regrouped into predefined categories according to their duration (e.g. less than one minute, 1–5 minutes, etc.). The relative frequency of sessions for each duration interval over a fixed period of time forms a discrete distribution, which can then be compared to some reference distribution:

Trend Query 6. Send a notification when the distribution

of session durations differs from at least k from a reference distribution.

This time, the computed trend is neither a scalar nor a set, but an associative map from categories to real numbers. Obviously, the "distance" between two such maps would need to be defined in an appropriate way according to the context; among others, one could cumulate the sum of the differences between map values for each category.

III. Related Work

The examples presented in the previous section have several points in common, even though they come from different fields. First, they involve the computation of some "aggregate value" over a sequence of events, which we called a *trend*. Second, they require a *realtime* feedback: a trend deviation should be detected as soon as possible; this excludes the possibility of some offline batch processing over a previously recorded log. Third, the "notification" they send does not necessarily represent an error, but rather an indication that something important *may* require further attention.

In this section, we survey the various tools, techniques and solutions that have been developed in the recent past for data mining, computation of trends, and event log processing. We classify these solution in three groups: log analysis tools, data mining tools, and process mining tools.

A. Log Analysis Tools

A first category of solutions regroups software and libraries that are specialized in the processing of event logs, either in realtime or on pre-recorded files. Some log analysis systems provide basic, Grep-like filtering capabilities, such as Snare[1], EventLog Analyzer[2], Splunk[3] and Lumberjack. More advanced tools fall into the broad term of Complex Event Processing (CEP) [34]. At its heart, CEP is the task of processing event feeds, possibly from multiple sources at the same time, in order to process their data and produce new event streams of a higher level of abstraction. Traditionally, CEP systems tend to be closer to database systems, and many of them borrow terms and syntax from database languages such as SQL.

For example, Cayuga [10] is a complex event monitoring system for high speed data streams. It provides a simple query language for composing stateful queries with a scalable query processing engine based on non-deterministic finite state automata with buffers. A Cayuga query has three parts; the `SELECT` clause chooses the attributes to include in the output events, the `FROM` clause describes a pattern of events that have to be matched, and the `PUBLISH` gives a name to the resulting output stream. Filtering, sums, averages are among the data manipulation functionalities supported by the engine.

In a similar way, Siddhi [43] is a query engine used in the WSO2 Complex Event Processor, an open source engine for real-time analytics of events. It supports classical event detection patterns like filters, windows, joins and event patterns and more advanced features like event partitions, mapping database values to events. In terms of query capabilities, Siddhi supports the computation of typical aggregation functions (sum, average) over windows, and can also express a form of sequential patterns on events. Other event processing and log analysis tools include Cordies [32], Flink [2], TESLA [11], and SPA [15]. They all provide functionalities similar in nature to that of one of the tools described above. Unfortunately, an in-depth comparison of all these solutions is out of the scope of this paper; the reader is referred to a recent technical report for more details on this topic [22].

Lesser known to the business process community is the existence of Runtime Verification (RV). In RV, a *monitor* is given a formal *specification* of some desirable property that a stream of events should fulfill. The monitor is then fed events, either directly from the execution of some instrumented system or by reading a pre-recorded file, and is responsible for providing a verdict, as to whether the trace satisfies or violates the property. Typically, a property expresses a correctness aspect of a trace, and is written in some formal notation such as temporal logic, μ-calculus, finite-state machines, or variants of these formalisms. Some runtime monitors developed in the past include MarQ [40], Mufin [12], RuleR [6], SpoX [25], and Tracematches [9].

Ties between log analysis and RV have been studied in a recent paper [21], where it was argued that, while CEP provides richer data processing and aggregation functions, RV could express more flexible properties related to the sequencing of events in a log. Apart from basic aggregation functions (such as average and sum), most runtime monitors and CEP tools provide no further data processing functions in their respective languages, and are hence unable to handle at least some of the use cases described in Section II.

B. Data Mining Tools

The first category of tools we have seen were specialized in the processing of event logs and streams, but did not necessarily provide advanced functionalities for computing trends. In contrast, a second broad category of solutions relates to tools specialized in statistics and data mining, but over generic datasets that are not necessarily sequential collections of events.

One of the oldest tools for statistical processing and data mining is Project-R, a GNU project written in the R programming language [28]. R is a powerful tool for statistical calculation and graphical programming, and allows the analysis of time series, classification, clustering and others. Another popular library is WEKA [20], which supports many standard data mining tasks, such as data preprocessing, classification, clustering, regression, visualization, and feature selection. WEKA's algorithms can be used directly by using its interface or by inserting them into Java code. The Apache Commons Math library[4] is another open source library providing basic functionalities for statistical manipulations, clustering and machine learning.

Among other tools specialized in data mining, let us mention Orange [13], Clementine[5], NLTK [8], as well as a

[1]https://www.intersectalliance.com/

[2]https://www.manageengine.com/products/eventlog

[3]https://splunk.com

[4]https://commons.apache.org/proper/commons-math/download_math.cgi

[5]http://www.spss.com/clementine

host of commercial data mining solutions described in a recent comparative study [39]. All these systems provide relatively advanced data mining and trend computation facilities; however, none is specialized in the processing of event logs. That is, the sequential aspect of event logs is not a defining characteristic of either of these systems. At best, a log can be represented as a list or a vector of elements, but otherwise, these libraries do not provide much in the way of stream processing, such as sliding windows. Perhaps more importantly, none of these systems can deal with *streams* –that is, sequences of events that are received one by one in realtime. They all assume that the collection of data over which mining must be done is static and completely known in advance.

C. Process Mining

A special category is reserved for tools and techniques from the field of process mining [44], which can be somehow seen as a crossover of the previous two types of solutions. In its original form, process mining is defined as the "automatic discovery of information from an event log" [46]. The best representative tool of this category is the ProM framework [18], which has been under active development for over a decade. ProM provides plugins of three types: *discovery* plugins compute outputs based only on the data contained in the event logs; *conformance* plugins compare log data to an external prescribed model, and *extension* plugins take advantage of both log data and models. RapidMiner [26] is a ProM extension that allows processing steps to be organized into a workflow —that is, a graph of processing steps, where the output of one step can be sent as the input of the next one.

Among other tools, iDHM [37] is an interactive data-aware process discovery tool that combines classification techniques and heuristics to discover data and control flow of the process simultaneously. MPE [36] is an interactive tool that integrates multi-perspective process mining. BupaR [29] is an open-source suite of R packages which can be used to explore and visualize event data and monitoring processes. Finally, Apromore [33] is an open-source business process analytics platform.

However, process mining, as its name implies, focuses on *processes*; many process mining operations either require a process model as part of their input, or attempt to deduce a process model from log data and work from this model afterwards. However, we have seen in Section II how the trends that are computed in the examples do not refer to a process model; for some of them, no such model could even reasonably be found (e.g. stream of TCP/IP packets or Active Directory sessions). Moreover, as with data mining software, we are not aware of process mining systems that can produce realtime feedback on an event log processed in streaming fashion.

IV. Detecting Trend Deviations on Event Streams

Based on the previous observations, we describe in this section various techniques for the computation of trends over a sequence of events taken from an arbitrary source, or from a set of such sequences. The various techniques we propose will be defined in terms of "workflows", in which streams of events are received, processed, and re-emitted to be consumed as the input of another processing step.

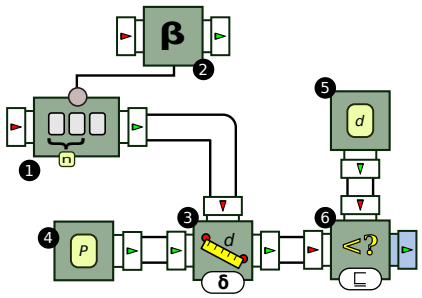


Figure 2: The static trend distance workflow

A. Static Trend Distance

We now introduce a generic technique for computing trends over a sequence of events taken from an arbitrary source, and for detecting whether the computed trend deviates from some "reference". This technique is best illustrated in the form of a workflow, as is shown in Figure 2. In this figure, each box represents an abstract unit of computation over one or more sequences of events. The pictogram on each of these boxes depicts the particular functionality of the corresponding box.

First, a stream of events is received and sent into a *sliding window* computation (box #1 in Figure 2). Input events come in from the left, and new output events are produced and come out from the right of the box. Such a computation requires two parameters: the width of the window (labelled n), and a computation β to perform on each window, represented by box #2. Concretely, the sliding window computation creates a window of the first n events received (e_0, e_1, …, e_{n-1}). It then feeds this window of events to computation β; the output returned by β is the first event to be output by box #1. The computation then "slides" one event forward, and creates a new window made of events e_1 to e_n. It feeds this window to β, whose return value is the second event to be output by box #1 —and so on.

The end result is that, from an input stream of events, box #1 creates a new stream, made of the application of β on successive windows of width n. Intuitively, β is intended to represent the calculation of a "trend" on a window that slides over the input stream. Since the goal of the process is to detect whether the input stream exhibits a "deviation" from some norm, the computed trends will be compared to some reference. This is the purpose of box #3, which evaluates what is called the *distance metric*. It takes two arguments; the first is the stream of trends computed by box #1; the second is a stream of reference patterns, provided by box #4. In the simplest scenario, the reference pattern does not change throughout the whole input stream, and box #4 simply returns the same reference pattern P over and over.

For each pair (P, p), where P is the reference pattern and p is a pattern computed by box #1, the distance metric $\delta(P, p)$ is evaluated, and its value d_p is returned as the output of box #3. Intuitively, δ is a function that estimates "how far" pattern p is from reference P. The workflow is expected to produce a notification when that distance becomes larger than a specific threshold. This is the task of box #6, which compares the computed distance d_p with a fixed threshold value d (provided by box #5). The function $\sqsubseteq$ is called the *comparison function*: $\sqsubseteq$

(d_p, d) returns $\top$ (true) when d_p is "greater" than the maximum threshold value d.

The net effect is that the workflow of Figure 2 receives a stream of arbitrary input events $e_0, e_1, \ldots$, and produces as its output a sequence of Boolean values $b_0, b_1, \ldots$ Under normal conditions, this workflow outputs the value $\bot$ (false) repeatedly. The occurrence of value $\top$ indicates an anomaly: technically, one can deduce that if $b_i = \top$, then:

$$\sqsubseteq (\delta(P, \beta(e_i, e_{i+1}, \ldots, e_{i+n-1})), d) = \top$$

In other words, the trend computed on the window of events e_i to e_{i+n-1} is at a distance greater than d from the reference P.

We call the workflow of Figure 2 the *static trend distance* workflow. As one can see, this basic workflow requires a number of parameters to represent a concrete computation. We can summarize them formally as follows. Let E be the set of events from which is made the original input stream, and let $\mathbb{B}$ be the set of Boolean values $\{\top, \bot\}$. Then:

- $n \in \mathbb{N}$ is a positive integer representing the *width* of the sliding window over which the trend is computed
- $\beta : E^n \to T$ is the *trend function*. From a window of n events in E, it computes a trend $t \in T$. The set T represents the set of possible trend values that can be returned by β.
- $P \in \mathbb{P}$ is an object called the *reference pattern*
- $\delta : \mathbb{P} \times T \to D$ is a function called the *distance metric*. It compares a reference pattern $P \in \mathbb{P}$ and a computed trend $t \in T$ and returns a distance $d \in D$, for some set of distances D.
- $d \in D$ is a distance value called the *maximum distance threshold*
- $\sqsubseteq : D^2 \to \mathbb{B}$ is the *distance comparison function*. In general, we expect that $\sqsubseteq$ induces an ordering relation $\leq$ on D, defined by $d \leq d' \Leftrightarrow \sqsubseteq (d, d') = \top$.

We call a *configuration* of the workflow a specific combination of definitions for each of these parameters.

B. Static Trend Distance Examples

A first advantage of the static trend distance workflow is that it is very generic. Depending on the way we define these seven parameters (the set E plus the six parameters above), the static trend distance workflow can represent many common computations over event streams. We give a few examples in the following.

1) Average: As a first example, we suppose that $E \triangleq \mathbb{R}$ is a stream of arbitrary numerical values. We define $\beta : \mathbb{R}^n \to \mathbb{R}$ as

$$\beta(e_0, \ldots, e_{n-1}) \triangleq \sum_{i=0}^{n-1} \frac{e_i}{n}$$

The function β computes the average of a sliding window of n events. Let us define $\mathbb{P} \triangleq \mathbb{R}$ and $\delta(x, y) : \mathbb{R}^2 \to \mathbb{R}^+$ such that $\delta(x, y) = |x - y|$. Finally, let $\sqsubseteq \triangleq \leq$ be the classical inequality over real numbers. For a given window width n, a reference $p \in \mathbb{R}$ and a distance threshold $d \in \mathbb{R}$, the static trend distance workflow detects whenever the average of the last n events diverges by more than d from the reference value p.

2) Vector of moments: The previous example can be generalized to arbitrary statistical moments. If X is a sequence of numerical values, the k-th raw sample moment (noted $E^k[X]$) can be defined as:

$$\frac{1}{|X|} \sum_{x \in X} x^k$$

As a reminder, the first two moments, $E^1[X]$ and $E^2[X]$, respectively represent the mean and the variance of the sample. One can define a function $\beta_m : \mathbb{R}^n \to \mathbb{R}^m$ such that $\beta_m(e_0, \ldots, e_{n-1}) = (E^1, E^2, \ldots, E^m)$, where E^k denotes the k-th sample moment of the set $\{e_0, \ldots, e_{n-1}\}$. The trend computed by function β_m is a *vector* of the first m statistical moments. Instead of a single number, the reference pattern $P \in \mathbb{P}$ also becomes a vector of real numbers $p = (p_1, \ldots, p_m)$, by letting $\mathbb{P} \triangleq \mathbb{R}^m$. It describes the expected statistical moments of the values contained in the input stream.

The distance function $\delta : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$ must now be defined over pairs of vectors $\overline{x} = (x_1, \ldots x_m)$ and $\overline{y} = (y_1, \ldots y_m)$. There exist a large number of distance metrics over an m-dimensional vector space, many of which can be described as particular cases of the Minkowski distance [38], which we note $\hat{\delta}_k$:

$$\hat{\delta}_k(\overline{x}, \overline{y}) \triangleq \left(\sum_{i=1}^{m} |x_i - y_i|^k \right)^{\frac{1}{k}}$$

When $k = 1$, $\hat{\delta}_1 = \sum |x_i - y_i|$ is called the *Manhattan distance*; when $k = 2$, we obtain the classical Euclidean distance. Finally, when $p = \infty$, $\hat{\delta}_\infty = \max(x_i - y_i)$ is the Chebychev distance. These various distance metrics may represent different concepts, depending on the context in which they are used. Example 1 above is the special case where $m = 1$ and $k = 1$.

3) Frequency distribution: Statistical computations are not the only trend computations that can generate a vector of numerical values. As an example, let E be a finite set of q arbitrary discrete symbols $\{\epsilon_1, \epsilon_2, \ldots, \epsilon_q\}$. Define $\beta : E^n \to [0, 1]^q$, such that $\beta(e_1, \ldots, e_n) \triangleq (c_1, \ldots, c_q)$ is such that c_i is the frequency of symbol ϵ_i (i.e. the number of occurrences of ϵ_i in $\{e_1, \ldots, e_n\}$ divided by n). Hence, $(c_1, \ldots, c_q)$ represents the *distribution* of symbols in a window of n events. Suppose that $q = 2$, and that $E = \{a, b\}$. The vector $(3/10, 7/10)$ would indicate a reference distribution where 30% of the symbols in a window of width n are a's, and 70% are b's.

Using Chebychev distance $\hat{\delta}_\infty$ as the distance metric, and $p = (p_1, \ldots, p_m)$ (where $p_i \in [0, 1]$ for every i) as the reference pattern, the static trend distance workflow will detect whenever any symbol ϵ_i occurs at a frequency that diverges by more than p_i from the reference distribution.

4) Multimodal reference patterns: The reference patterns we have seen so far are called *unimodal*: they consist of a single number, vector or distribution. In order not to raise an alarm, the event stream under consideration must remain close to that single reference pattern. However, there exist situations where the reference is made of multiple patterns: these are called *multimodal*.

As an example, suppose that the stream of discrete symbols in the previous example can follow one of *two* possible distributions: either 30% of a's and 70% of b's, or the other way around. This can be modelled by two reference patterns:

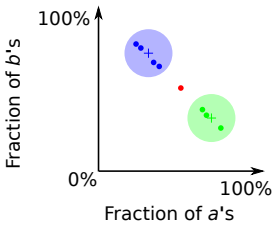


Figure 3: An example of a bimodal reference pattern

the vectors $(3/10, 7/10)$ and $(7/10, 3/10)$. It is not possible to know in advance to which of these two distributions the stream under study conforms, but one may want it to stay close to either of these two distributions.

This is a first example where the sets $\mathbb{P}$ and T differ. Here, $T \triangleq \mathbb{R}^q$, while $\mathbb{P} \triangleq 2^{\mathbb{R}^q}$. In other words, the computed trend is a distribution, while the reference pattern is a *set* of distributions. This, in turn, has an impact on the distance function δ. One possible way to define it is as follows: $\delta_{\delta'} : 2^{\mathbb{R}^q} \times \mathbb{R}^q \to \mathbb{R}$ is such that:

$$\delta_{\delta'}(\{\overline{p}_1, \ldots, \overline{p}_\ell\}, \overline{t}) = \min_{i=1}^{\ell} \{\delta'(\overline{p}_i, \overline{t})\}$$

This function takes as parameter another distance metric $\delta' : \mathbb{R}^q \times \mathbb{R}^q \to \mathbb{R}$; for example, it can be one of the vector space metrics we have introduced in the previous example. Intuitively, the distance between a computed trend vector $\overline{t}$ and a set of vectors $\{\overline{p}_1, \ldots, \overline{p}_\ell\}$ is defined as the smallest distance between $\overline{t}$ and one of the $\overline{p}_i$, according to the vector space distance metric δ'.

Going back to our example, let us use for the distance metric $\hat{\delta}_2$, the classical Euclidean distance; define as a threshold the value $d = 0.07$. As Figure 3 illustrates, each of the two reference vectors can be depicted as points in a two-dimensional space. With the specific distance metric and distance threshold used here, they represent the center of two discs with a radius of 0.07. Each window of 10 events can also be plotted as a point in this graph, corresponding to the frequency of a's and b's it contains. The windows that lie in the blue and the green discs are "close enough" to one of the reference vectors, while the window of events represented by the red dot is too far to either of them.

C. Self-Correlated Trend Distance

The static trend distance pattern, as its name implies, expects a fixed reference pattern $P \in \mathbb{P}$ given beforehand. So far, we have left out the question of how this reference pattern is obtained. In this section, we introduce a first method for computing such a reference pattern, by comparing the trend calculated on the current window to another trend computed on the same sequence of events, but further away in the past. In such a context, an alarm is triggered when the stream exhibits a trend that becomes too different from what was observed previously. Since there is no fixed reference pattern, and that the trend deviations we observe on a stream refer to the stream itself, we call this technique *self-correlated trend distance*.

Figure 4 shows the self-correlated trend distance workflow. Boxes #1–3 and #5–6 are similar to the static trend distance

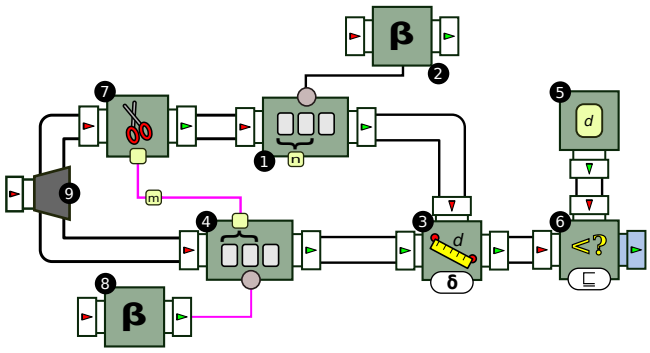


Figure 4: The self-correlated trend distance workflow

pattern. The difference lies upstream, in how the "current" and the "reference" trends are extracted from the input event sequence. That sequence is first split into two copies (box #9). The topmost copy is trimmed of its first m events, as is represented by box #7. This makes such that the streams entering boxes #1 and #4 are offset by m: while box #4 receives the stream of events $e_0, e_1, \ldots$, box #1 receives the stream $e_m, e_{m+1}, \ldots$ These two boxes then apply the same computation β on a sliding window: box #1 on a window of width n, and box #4 on a window of width m. The output of β on these two windows is then sent to the distance metric (box #3), and the rest of the process unfolds similarly to the static trend distance workflow we have introduced earlier.

As before, the distance metric (box #3) is fed with a sequence of pairs of trends (p, t), where $t \in T$ is the trend computed on the latest window of n events. The reference trend $p \in \mathbb{P}$, however, is now also computed from a window of events of the same stream. Let k be the number of events received from the input stream so far, with $k \geq m + n$. Due to the presence of the trimming box (#7), it can be observed that, when box #1 applies β on a window of the last n events $(\{e_{k-(n-1)}, e_{k-(n-2)}, \ldots, e_k\})$, box #4 applies β on a window of the m preceding events $(\{e_{k-n-(m-1)}, e_{k-n-(m-2)}, \ldots, e_{k-n}\})$. In other words, the distance metric compares the trend computed from the last n events to a reference computed from the m events before them.

The rest works in a similar way to the static trend distance workflow. The self-correlated workflow can be instantiated in various ways, depending on how one gives values to seven parameters:

- E is the set of input events, and is defined as before
- $n \in \mathbb{N}$ is the width of the window used to compute the "present" trend
- $m \in \mathbb{N}$ is the width of the window used to compute the "past" trend
- $\beta_m : E^m \to \mathbb{P}$ and $\beta_n : E^n \to T$ are the two trend computations to be applied on the "past" and "present" windows, respectively.
- $\delta : \mathbb{P} \times T \to D$ is the distance metric, defined as before
- $d \in D$ is the distance threshold
- $\sqsubseteq : D^2 \to \mathbb{B}$ is the distance comparison function

In many cases, one is interested in applying the same computation on both the past and the present windows; in such a case, $m = n$, $\beta_m = \beta_n$, and $\mathbb{P} = T$. Then, most of the examples shown

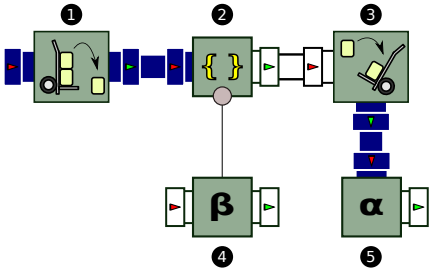


Figure 5: The trend extraction workflow

in the previous section can be converted into self-correlated workflows in a straightforward way. For instance, Example 1 will raise an alarm when the average of the last n numerical values has a difference greater than d with the average of the n values before them.

In the case of Example 2, let us use the Chebychev distance $\hat{\delta}_\infty$ as the distance metric, and a distance threshold of $d \in \mathbb{R}$. The self-correlated trend distance workflow will detect whenever any of the m sample moments computed on the last n events differs by more than d from the sample moments computed on the preceding n events. Using an alternate distance metric and comparison function, one can also specify a separate threshold for each moment. Let $D \triangleq \mathbb{R}^m$, and define $\delta : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^m$ such that $\delta((x_1, \ldots, x_m), (y_1, \ldots, y_m)) \triangleq (z_1, \ldots, z_m)$ if and only if $z_i = |x_i - y_i|$ for every $i \in [1, m]$. This distance metric computes the Manhattan distance of each pair of vector components. The distance threshold $d \in \mathbb{R}^m$ can now define a value for each component. Finally, the comparison function $\sqsubseteq : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{B}$ can be defined such that $\sqsubseteq ((x_1, \ldots, x_m), (y_1, \ldots, y_m)) = \top$ if and only if $y_i > x_i$ for some $i \in [1, m]$.

Example 3, which computes a frequency distribution of symbols, can be turned into a self-correlated workflow in a similar way. As a matter of fact, the only example that cannot readily be turned into self-correlation is Example 4, which involves a multimodal reference pattern. Indeed, since β_n computes a single trend on the "present" window, if we let $\beta_m = \beta_n$, a single trend will also be computed on the "past" window. This is in line with the (reasonable) hypothesis that a single event log follows one distribution or another, but not multiple such distributions at the same time.[6]

D. Trend Extraction

The case of multimodal reference patterns is much more likely to arise when collating trends computed from multiple logs. In this section, we introduce a second mechanism for obtaining a reference trend, this time by computing individual trends from a set of event sequences obtained beforehand. This yields to yet another workflow, called the *trend extraction workflow*.

This workflow is illustrated in Figure 5. The input to this workflow is a finite *set* of pre-recorded event logs. This set is "unpacked" by box #1, and each log is then fed to box #2, which replays it to the trend function β (box #4). Each event output by box #2 is the result of applying β on one complete event log; it corresponds to the trend extracted from that log. The trends for each log are re-packed into a set, which is then given to box #5. This box applies a function α to the set of trends; intuitively, α can be seen as an aggregation function that calculates a global trend from the individual trends obtained from each log. This global trend, once computed, can then be used as the reference pattern P in the original static trend distance workflow.

Like all the workflows seen so far, the trend extraction workflow can be instantiated in many different ways, this time by defining three parameters:

- E is the set of events in the input logs; let us denote by E^* the set of logs made of events from E
- $\beta : E^* \to T$ is the trend extraction function, which takes a log in E^* and computes a trend $t \in T$
- $\alpha : 2^T \to T'$ is the trend aggregation function; it takes multiple trends in T and computes a global trend $t' \in T'$

Note how the trend extraction workflow works as a preliminary step on a set of pre-collected logs, *before* attempting to detect deviations on some other event stream. Of all the workflows shown in this paper, this is the one closest to what could be considered as data mining. In the following, we shall give a few examples of trend and aggregation functions that can be used, depending on the context.

1) Average: We start with a simple example where $E \triangleq \mathbb{R}$, and event logs are made of scalar numerical values. We define $\beta : \mathbb{R}^* \to \mathbb{R}^2$ such that, for $\langle e_0, e_1, \ldots, e_n \rangle \in \mathbb{R}^*$, $\beta(\langle e_0, e_1, \ldots, e_n \rangle) = (x, y)$ if and only if $x = \sum_{i=0}^{n} \frac{e_i}{n}$ and $y = n$. In other words, for each log, β computes the average of the values in the log and the length of the log. We can then define an aggregation function $\alpha : 2^{\mathbb{R}^2} \to \mathbb{R}$, such that:

$$\alpha(\{(x_1, y_1), \ldots, (x_m, y_m)\}) \triangleq \frac{\sum_{i=1}^{m} x_i y_i}{\sum_{i=1}^{m} y_i}$$

The function α calculates the average of the average values in each log, weighted by the length of each log.

2) Clustering: A more interesting example involves multimodal reference patterns. Let $E \triangleq \{\epsilon_1, \epsilon_2, \ldots, \epsilon_q\}$ be a set of q discrete symbols, as in Example 4 of Section IV-B. Define $\beta : E^* \to [0, 1]^q$ as the function which, for each log, computes the vector of the relative frequency of each symbol in the log. This is one example where different logs can exhibit different distributions of symbols. In this case, a *clustering algorithm* can be used to find out what are the vectors that are most representative of the various distributions.

More formally, a clustering algorithm takes as input a set I of m-dimensional vectors, and returns as its output another set of O m-dimensional vectors. This set is such that the distance between each vector of I and the closest vector in O is minimized, according to some distance metric δ. If the vectors in I lie in a small number of relatively disjoint "clusters", a clustering algorithm attempts to find what are called the cluster *centroids*.

This is illustrated in Figure 3, where each of the points in the plot represents the relative frequency of a's and b's in a log. As one can see, there are many logs where the relative frequency of a's and b's is close to 30%/70% (blue dots), and many logs where the relative frequency of a's and b's is close

[6]However, we could consider the case where various *sections* of the same log follow different distributions. This will be discussed in the conclusion.

to 70%/30% (green dots). A clustering algorithm, given this set of points, would be able to find two clusters, and evaluate their centroids as the vectors $(3/10, 7/10)$ (blue "+" sign) and $(7/10, 3/10)$ (green "+" sign). A detailed description of clustering algorithms is out of the scope of this paper; such algorithms have been widely studied in the data mining community. Let us simply mention a few of the most well-known: K-means [35] and its many variants [5], [30], and DBSCAN [17].

This completes the loop for Example 4, by showing how a multimodal reference pattern can be obtained from a set of pre-existing logs. One could use as function $\alpha : 2^{\mathbb{R}^m} \rightarrow 2^{\mathbb{R}^m}$ any of the clustering algorithms mentioned above; the output of the pattern extraction workflow then becomes a set of m-dimensional reference vectors, which can be used as the basis for instantiating a static trend distance workflow. Note how this workflow is again very generic; it only depends on the definitions given to E, β and α. In the case where α is a clustering algorithm, the actual vectors extracted from the input logs can also be arbitrary; statistical moments and frequency distributions are but two examples of the wide range of numerical features one can compute over a sequence of events.

V. Implementation and Experiments

We now discuss how the concepts described in the previous section have been implemented into an actual event stream mining tool. To this end, a Java library called Pat The Miner (PTM for short)[7] has been developed. In this section, we first explain how the library has been implemented and how it can be used; we then proceed to describe various empirical measurements that have been taken on this library, and give an overview of its performance under various conditions.

A. A BeepBeep Palette for Stream Data Mining

Pat The Miner has been developed as an extension to an existing event stream processing engine called BeepBeep [24], which is freely available online through an open source license[8]. BeepBeep works according to a handful of fundamental concepts. We shall only very briefly describe them in this section; the reader is referred to a recent tutorial for detailed information [21].

The main computing unit in BeepBeep is called a *processor*; every processor can be seen as a "box" that takes event streams as its input, and produces other event streams as its output. BeepBeep's core is made of about a dozen processors for performing basic manipulations over event streams: filtering events, joining streams together, applying a function on each input event, or computing an aggregate function over a sliding window.

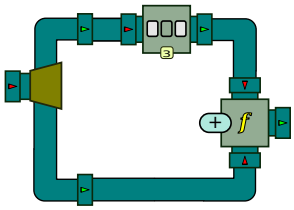
More complex computations can be achieved in two ways. The first is by creating new processors directly as Java objects that are programmed to perform a specific type of processing. Extensions to BeepBeep's core are called *palettes*; there exist palettes defining custom processors and functions for a variety of domain-specific use cases, such as processing XML documents, evaluating temporal logic formulæ, accessing databases,

[7]This is a mildly successful pun on the expression "pattern mining".

[8]https://liflab.github.io/beepbeep-3

```
Fork f = new Fork(2);
ApplyFunction sum = new ApplyFunction(Numbers.addition);
CountDecimate decimate = new CountDecimate(3);
Connector.connect(fork, BOTTOM, sum, BOTTOM)
  .connect(fork, TOP, decimate, INPUT)
  .connect(decimate, OUTPUT, sum, TOP);
```

(a)



(b)

Figure 6: Creating processor chains in BeepBeep. (a) A snippet of Java code that instantiates and connects processors. (b) The same chain represented graphically.

etc. The second is through *composition*, by directing the output of a processor to the input of another; this mechanism provides a flexible way to create chains of processors corresponding to various kinds of processing tasks.

BeepBeep provides multiple ways to create processor pipes and to fetch their results. One of them is programmatically, using BeepBeep as a library and Java as the glue code for creating the processors and connecting them. For example, Figure 6 shows a simple processor chain, both as a code snippet and as a graphical representation. As one can see, processor chains are exactly the BeepBeep equivalent of the "workflows" we introduced in Section IV, with each box represented by a `Processor` object.

Pat The Miner is a BeepBeep palette that defines new `Processor` and `Function` objects specific to the computation of trends on event logs. The library is freely available through an open source code repository[9]. In particular, it provides two new processors for the detection of trend deviations, called the `TrendDistance` and `SelfCorrelatedTrendDistance` processors. These two processors work exactly in the way described in the previous section. For example, the following is a Java code snippet that instantiates a trend distance pattern:

```
TrendDistance<Number,Number,Number> td = new TrendDistance(
  6, 200, new RunningAverage(),
  new FunctionTree(Numbers.absoluteValue,
      new FunctionTree(Numbers.subtraction,
          StreamVariable.X, StreamVariable.Y)),
  0.5, Numbers.isLessThan);
```

In this code fragment, one can recognize all the parameters that define a specific instance of the trend distance pattern. Here, `RunningAverage` is a processor object that computes the running average over a stream of numbers. The value `200` is the width of the sliding window over which this average is to be computed. The number `6` is the reference trend; here, we compute the deviation of the stream of numbers with respect to a reference average of 6. The `FunctionTree` constructor builds a function defining the distance metric: it is a binary function that receives two values, subtracts them and takes the

[9]https://github.com/liflab/PatTheMiner

absolute value of the difference (notice how this is the definition of Manhattan distance in dimension 1). The value `0.5` is the maximum distance threshold, and `Numbers.isLessThan` is a reference to the function used to compare the computed distance with that threshold.

From this point on, `td` is a `Processor` object that can be used like any other BeepBeep processor. Its input can be connected to any stream of numerical values, and its output produces a stream of Boolean values. Watching this output stream for the occurrence of value `false` can be used to detect a trend deviation in the input stream. Moreover, since all BeepBeep processors operate in a streaming fashion, this means that this deviation can be detected in realtime, as the processor progresses through the input stream.

In line with the observations we made in the previous section, we can see that the `TrendDistance` class is generic. The three `Number` class names appearing in the type declaration of `td` respectively refer to the type of the computed trend, the type of the events produced by the trend processor, and the type of value returned by the distance metric. This means that `TrendDistance` can be instantiated differently, by using other objects for the trend processor, the distance metric and the reference trend. For example, the following code snippet instantiates a trend distance pattern, using a vector of the first three statistical moments as the trend:

```
TrendDistance<DoublePoint,Number,Number> td = new TrendDistance(
  new DoublePoint(new double[]{0.3d, 0.1d, 0.6d}),
  200, new RunningMoments(3),
  new PointDistance(new EuclideanDistance()),
  2, Numbers.isLessThan);
```

This time, the reference pattern is a vector of numerical values (`DoublePoint`), and the distance metric is the Euclidean distance over these vectors.

The processor object for self-correlated trend distance can be created in a similar way; the arguments of its constructor match the parameters that this particular pattern takes.

B. Trend Distance Experiments

In order to assess the feasibility of the approach, we proceeded to various empirical measurements on the concrete implementation of Pat The Miner on the trend patterns discussed earlier. The experiments were implemented using the LabPal testing framework [23], which makes it possible to bundle all the necessary code, libraries and input data within a single self-contained executable file, such that anyone can download and independently reproduce the experiments. A downloadable lab instance containing all the experiments of this paper can be obtained from Zenodo, a research data sharing platform[10]. Overall, our empirical measurements involve 33 individual experiments, which together generated 81 distinct data points. All the experiments were run on a Intel CORE i5-7200U 2.5 GHz running Ubuntu 18.04, inside a Java 8 virtual machine with 1746 MB of memory.

In a first set of experiments, we measured the performance of the trend distance pattern on various trend computations. More specifically, we attempted to quantify the throughput one can expect when trying to detect a deviation for a fixed reference trend in realtime on a given event stream. The experiments were performed by generating on-the-fly an event stream of random numbers or discrete symbols, depending on the trend to be computed over the event stream. This stream was then fed to the trend distance pattern processor. The trend processors and distance metrics used in the experiments are summarized in Table I. These various workflow configurations can be seen as abstract versions of some of the queries introduced in Section II.

	Trend	**Metric**
C1	Running average	Manhattan distance of dimension 1
C2	Vector of the first 3 statistical moments	Euclidean distance
C3	Cumulative symbol distribution	Map distance
C4	Vector of symbol frequencies	Euclidean distance to closest cluster centroid

Table I: The various trend processors and associated distance metrics used in the experiments.

Since the purpose of the experiments is to measure the throughput that can be achieved by the processor, the actual distance threshold is irrelevant. Indeed, the trend distance processor returns a stream of Boolean values, and whether these values are true (the input stream is close enough to the reference trend) or false (the input stream is too far from the reference trend) has no impact on the computation. This, in turn, has no impact on the number of events per second that can be processed for a given trend processor. Therefore, in each of the experiments, the reference pattern and the distance threshold were set to arbitrary "dummy" values; the actual values being used have no practical impact on the measurements.

Figure 7 shows the cumulative computation time of the trend distance processor, when the pattern to be computed is a vector of the first three statistical moments (configuration C2 in Table I). The plot illustrates the computation time for different window widths, ranging from 50 to 200 events. It clearly shows a linear trend; this means that the computation does not "slow down" as the processor progresses through the event stream. As expected, a wider window entails a larger computation to be done for each window, and hence a lower throughput. In the case of the vector of moments, average throughput is approximately of 26200.0 Hz for a window of 50 events, and 7570.0 Hz for a window of 200 events. (As a reminder, a throughput of only 1200 Hz is enough to process one hundred million events per day.)

The other trend processors and distance metrics exhibit very similar behaviour, with only the throughput range differing between the experiments. For example, in configuration C3, throughput ranges from 34500.0 Hz for a window of 50 events to 8990.0 Hz for a window of 200. As a matter of fact, for a given window width, all the trend processors we studied behave linearly in the length of the event stream. This is due to the fact that the bulk of the computation in the trend distance pattern is done on a window of fixed size, which does not depend on how many events have been processed since the beginning of the stream. In other words, the amount of work to compute the first window is the same as the amount of work to compute the 1,000th.

We also measured the impact of window width on the trend

[10]DOI: 10.5281/zenodo.1252497

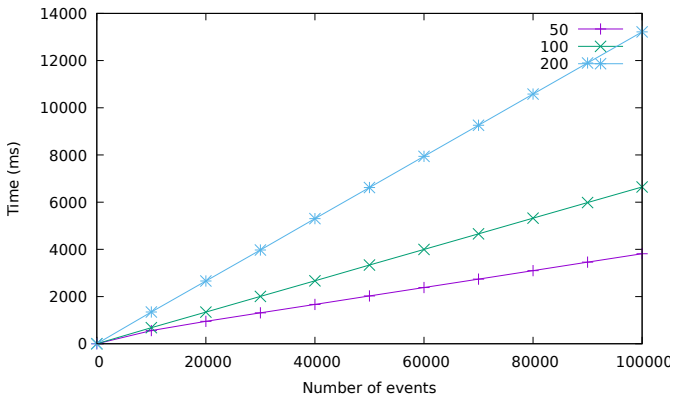


Figure 7: Cumulative computation time on an event stream, using the vector of moments as the trend computation.

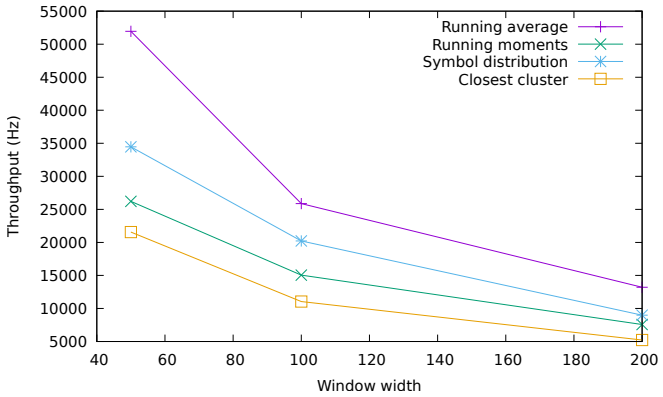


Figure 8: Impact of window width on throughput, for various trend distance computations.

computations. Figure 8 shows the average throughput of various trend processors, by varying the width of the window in each case. This figure confirms the intuition that increasing the size of the window has a negative impact on throughput. If $f(n)$ is a function that returns the computation time (in seconds) for a single window of width n, then $1/f(n)$ such windows can be processed during one second. As $f(n)$ increases with n, the ratio $1/f(n)$ decreases, yielding the shape of an inverse function. This is indeed what can be observed for all trend processors shown in Figure 8.

C. Self-Correlation Experiments

In a second batch of experiments, we measured the throughput of the self-correlated trend distance processor. As we have seen earlier, this pattern differs from a mere trend distance by the fact that the reference trend is not a fixed value, but is rather computed on-the-fly on a sliding window of past events. We therefore expect it to be heavier in terms of computational load. More precisely, let us assume that the computational cost of the sliding window dominates the overall computation of the trend distance pattern. Since the self-correlated trend distance pattern involves two such window computations, then its throughput should be reduced by half compared to the trend distance pattern.

Trend function	STD	SCTD
Average	13206.0	4059.0
Closest cluster	5227.0	2589.0
Running moments	7565.0	2792.0
Symbol distribution	8985.0	4199.0

Table II: Comparison of throughput (in Hz) for the static trend distance (STD) vs. the self-correlated trend distance workflow (SCTD), for each trend computation. The width of the window for all the experiments is 200.

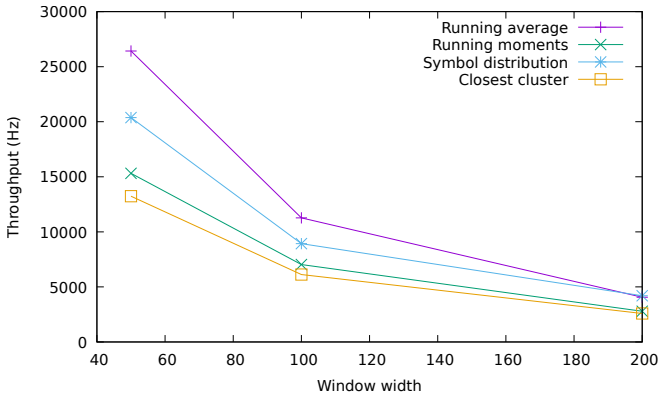


Figure 9: Impact of window width on throughput, for various self-correlated trend distance computations.

For all tested configurations, the running time for this workflow again behaves linearly in the total number of events processed; however, as expected the throughput is reduced compared to the use of a fixed pattern. This is summarized in Table II, which compares the throughput of the trend distance pattern vs. the self-correlated trend distance pattern, for the same window size. Globally over all trend metrics, we observed that the use of self-correlation incurs a slowdown of at most 49% for a window width of 50, and of at most 69% for a window width of 200. This is roughly consistent with the reduction of 50% of throughput predicted by our crude estimation.

Again, the global throughput is influenced negatively by the width of the windows themselves, as is illustrated by Figure 9. We observe the same trend as for the trend distance pattern, albeit with lower throughputs overall.

D. Trend Extraction Experiments

A last set of experiments involves the empirical evaluation of the trend extraction workflow. This time, for a given trend computation (β) and aggregation algorithm (α), the parameters that have an impact on computation time are the number of pre-recorded logs n_ℓ over which to compute a trend, as well as the length ℓ of these logs. The running time of β should be proportional to both n_ℓ and ℓ; however, since the aggregation algorithm is given one similar "trend" object for each log, its running time should only be dependent on n_ℓ.

Figure 10 shows how the number of logs impacts the running time of the trend extraction workflow, in the case where the trend computation is symbol distribution, and the aggregation computation is the k-means clustering algorithm

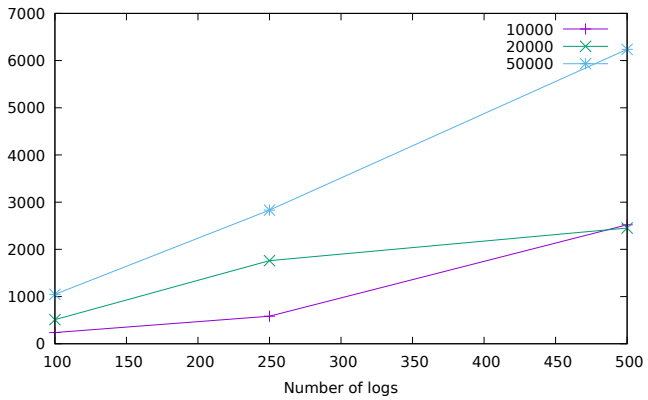


Figure 10: Impact of number of logs for the trend extraction workflow, using symbol distribution as the trend computation (β) and the k-means clustering algorithm as the aggregation computation (α). Each data series represents a different length given to each log in the set (from 10,000 events to 50,000 events per log).

mentioned earlier (provided by the Apache Commons Math library mentioned earlier). As one can see, both the number of logs used as the basis for computation and the length of each log contribute to the duration of the trend extraction process. In the worst case, 80.0 logs of 50,000 events could still be processed through this workflow at each second. Although not shown here, it shall be mentioned that the time taken for computation of a trend for each log largely dominates the time then taken by the aggregation function α. In other words, the bulk of the work is to extract a trend from a large set of logs, and not to run the clustering algorithm from these trends afterwards.

VI. Conclusion and Future Work

In this paper, we have studied the concept of *trends* over a log of events, and concentrated on the task of detecting trend *deviations* in a log in realtime. To this end, we introduced three generic processing workflows: the static and self-correlated trend distance workflows, and the trend extraction workflow. All three describe a general, step-by-step process for transforming event streams; specific types of computations are achieved by giving different definitions to the various parameters that these workflows accept.

We have seen through examples how various common log crunching tasks become particular cases of one of these workflows. These include many descriptive statistical measures, distributions, and even some data mining operations such as clustering. Thanks to the generic nature of these workflows, in principle any computation can be used for the trend computation. What is more, we have seen how an implementation of the workflows as an extension of the BeepBeep event stream engine makes it possible to compute trends and detect violations in a streaming fashion. Experimental results show that, for frequent types of trends, a processing speed in the order of thousands of events per second can be expected using standard hardware.

The ideas introduced in this paper open the way to multiple extensions, and also raise many interesting research questions. First, on the technical side, we are studying the possibility of integrating Pat The Miner as a plugin for the ProM platform; moreover, since the library is written in Java, it could easily incorporate data mining features of existing Java libraries, such as WEKA. Second, trend computations using less "traditional" functions than feature vectors should be explored. Thanks to the large number of existing plugins available for BeepBeep, one could imagine trend computations involving a mixture of statistics, formal methods, and signal processing. Third, as we have already mentioned, the trend computations presented in this paper can be seen as a distant relative of existing process mining tools and techniques. It would be very interesting to examine the implications of seeing a process model, reconstructed from a set of logs, as a reference trend over which distance metrics could be defined. Multiple variants of the three trend workflows themselves could also be proposed, for example by varying the relative placement of the windows with respect to the stream, by comparing the trends computed over two realtime streams against each other, or by enabling the computation of multimodal trends inside the self-correlation workflow.

The use of the trend workflows defined in this paper also raise a number of technical issues. First, we have seen that the impact of window width on the throughput of trend computations can be problematic for very large windows. A first possibility would be to study so-called "hopping" windows, which would slide across a stream at a larger step than every single event. Moreover, the computational cost of windows could also be mitigated by the use of parallelism. It turns out that the BeepBeep event stream processing engine allows certain types of computations to be performed in parallel when the host machine has a multi-thread processor; this process, however, requires much fine tuning to actually be beneficial. The insertion of multi-threading into trend distance processors, and the study of its impact on global throughput, is planned as future work.

Several questions are also left unanswered regarding the workflows themselves. For example, self-correlation may suffer from "tunnel vision", where a very progressive and long-running trend change may go unnoticed if the windows being compared are too small, and hence never be different enough to trigger an alarm. Moreover, the case where a log is separated into several sections, each following a different trend, is not properly accounted by either workflow model. Overall, the early results obtained using these workflows are nevertheless promising, and could prove useful in a wide range of use cases involving logs of various kinds.

References

[1] R. Agrawal, C. M. Johnson, J. Kiernan, and F. Leymann. Taming compliance with sarbanes-oxley internal controls using database technology. In L. Liu, A. Reuter, K. Whang, and J. Zhang, editors, *ICDE 2006*, page 92. IEEE Computer Society, 2006.

[2] A. Alexandrov, R. Bergmann, S. Ewen, J. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The stratosphere platform for big data analytics. *VLDB J.*, 23(6):939–964, 2014.

[3] A. Awad, G. Decker, and M. Weske. Efficient compliance checking using BPMN-Q and temporal logic. In M. Dumas, M. Reichert, and M.-C. Shan, editors, *BPM*, volume 5240 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2008.

[4] J. Bacon, P. R. Pietzuch, J. Sventek, and U. Çetintemel, editors. *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010*. ACM, 2010.

[5] S. Bandyapadhyay and K. R. Varadarajan. On variants of k-means clustering. *CoRR*, abs/1512.02985, 2015.

[6] H. Barringer, D. E. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *J. Log. Comput.*, 20(3):675–706, 2010.

[7] A. Berry and Z. Milosevic. Real-time analytics for legacy data streams in health: Monitoring health data quality. In D. Gasevic, M. Hatala, H. R. M. Nezhad, and M. Reichert, editors, *EDOC*, pages 91–100. IEEE, 2013.

[8] S. Bird, E. Klein, and E. Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. O'Reilly, 2009.

[9] E. Bodden, L. J. Hendren, P. Lam, O. Lhoták, and N. A. Naeem. Collaborative runtime verification with Tracematches. *J. Log. Comput.*, 20(3):707–723, 2010.

[10] L. Brenna, J. Gehrke, M. Hong, and D. Johansen. Distributed event stream processing with non-deterministic finite automata. In A. S. Gokhale and D. C. Schmidt, editors, *DEBS*. ACM, 2009.

[11] G. Cugola and A. Margara. TESLA: a formally defined event specification language. In Bacon et al. [4], pages 50–61.

[12] N. Decker, J. Harder, T. Scheffel, M. Schmitz, and D. Thoma. Runtime monitoring with union-find structures. In M. Chechik and J. Raskin, editors, *TACAS 2016*, volume 9636 of *Lecture Notes in Computer Science*, pages 868–884. Springer, 2016.

[13] J. Demšar, B. Zupan, G. Leban, and T. Curk. Orange: From experimental machine learning to interactive data mining. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 537–539. Springer, 2004.

[14] B. Desmond, J. Richards, R. Allen, and A. G. Lowe-Norris. *Active Directory: Designing, Deploying, and Running Active Directory*. O'Reilly, 2013.

[15] R. M. Dijkman, S. P. Peters, and A. M. ter Hofstede. A toolkit for streaming process data analysis. In S. Rinderle-Ma, F. Matthes, and J. Mendling, editors, *EDOC*, pages 304–312. IEEE, 2016.

[16] M. El Kharbili, A. K. A. de Medeiros, S. Stein, and W. M. P. van der Aalst. Business process compliance checking: Current state and future challenges. In P. Loos, M. Nüttgens, K. Turowski, and D. Werth, editors, *Modellierung betrieblicher Informationssysteme - Modellierung zwischen SOA und Compliance Management - 27.-28. November 2008 Saarbrücken, Germany*, volume 141 of *LNI*, pages 107–113. GI, 2008.

[17] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In E. Simoudis, J. Han, and U. M. Fayyad, editors, *KDD 1996*, pages 226–231. AAAI Press, 1996.

[18] B. F. van Dongen, A. Karla A. de Medeiros, H. M. W. Verbeek, A. Weijters, and W. M. P. Aalst. The ProM framework: A new era in process mining tool support. In *Lecture Notes in Computer Science*, volume 3536, pages 444–454, 06 2005.

[19] G. Governatori, Z. Milosevic, and S. W. Sadiq. Compliance checking between business processes and business contracts. In *EDOC*, pages 221–232. IEEE Computer Society, 2006.

[20] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[21] S. Hallé. When RV meets CEP. In Y. Falcone and C. Sánchez, editors, *RV 2016*, volume 10012 of *Lecture Notes in Computer Science*, pages 68–91. Springer, 2016.

[22] S. Hallé. From complex event processing to simple event processing. *CoRR*, abs/1702.08051, 2017.

[23] S. Hallé. LabPal: repeatable computer experiments made easy. In T. Bultan and K. Sen, editors, *ISSTA 2017*, pages 404–407. ACM, 2017.

[24] S. Hallé and S. Varvaressos. A formalization of complex event stream processing. In M. Reichert, S. Rinderle-Ma, and G. Grossmann, editors, *EDOC 2014*, pages 2–11. IEEE Computer Society, 2014.

[25] K. W. Hamlen and M. Jones. Aspect-oriented in-lined reference monitors. In Ú. Erlingsson and M. Pistoia, editors, *PLAS*, pages 11–20. ACM, 2008.

[26] M. Hofmann and R. Klinkenberg. *RapidMiner: Data mining use cases and business analytics applications*. CRC Press, 2013.

[27] J. Horcas, M. Pinto, L. Fuentes, W. Mallouli, and E. M. de Oca. An approach for deploying and monitoring dynamic security policies. *Computers & Security*, 58:20–38, 2016.

[28] R. Ihaka and R. Gentleman. R: a language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314, 1996.

[29] G. Janssenswillen and B. Depaire. Bupar: business process analysis in r. 2017.

[30] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(7):881–892, 2002.

[31] D. Knuplesch and M. Reichert. A visual language for modeling multiple perspectives of business process compliance rules. *Software and System Modeling*, 16(3):715–736, 2017.

[32] G. G. Koch, B. Koldehofe, and K. Rothermel. Cordies: expressive event correlation in distributed systems. In Bacon et al. [4], pages 26–37.

[33] M. La Rosa, H. A. Reijers, W. M. Van Der Aalst, R. M. Dijkman, J. Mendling, M. Dumas, and L. García-Bañuelos. APROMORE: An advanced process model repository. *Expert Systems with Applications*, 38(6):7029–7040, 2011.

[34] D. C. Luckham. *The power of events – An introduction to complex event processing in distributed enterprise systems*. ACM, 2005.

[35] D. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003.

[36] F. Mannhardt, M. De Leoni, and H. A. Reijers. The multi-perspective process explorer. *BPM (Demos)*, 1418:130–134, 2015.

[37] F. Mannhardt, M. de Leoni, and H. A. Reijers. Heuristic mining revamped: an interactive, data-aware, and conformance-aware miner. *BPM 2017 Demos*, 1920, 2017.

[38] M. Panda, S. Dehuri, and M. R. Patra. *Modern Approaches of Data Mining: Theory and Practice*. Alpha Science International, 2015.

[39] R. Raut and A. Nathe. Comparative study of commercial data mining tools. *International Journal of Electronics, Communication and Soft Computing Science & Engineering (IJECSCSE)*, page 128, 2015.

[40] G. Reger, H. C. Cruz, and D. E. Rydeheard. MarQ: Monitoring at runtime with QEA. In C. Baier and C. Tinelli, editors, *TACAS 2015*, volume 9035 of *Lecture Notes in Computer Science*, pages 596–610. Springer, 2015.

[41] S. Rinderle-Ma and S. Kabicher-Fuchs. An indexing technique for compliance checking and maintenance in large process and rule repositories. *Enterprise Modelling and Information Systems Architectures*, 11:2:1–2:24, 2016.

[42] J. Rodrigues. *Health Information Systems: Concepts, Methodologies, Tools, and Applications, Volume 1*. IGI Global, 2010.

[43] S. Suhothayan, K. Gajasinghe, I. L. Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara. Siddhi: a second look at complex event processing architectures. In R. Dooley, S. Fiore, M. L. Green, C. Kiddle, S. Marru, M. E. Pierce, M. Thomas, and N. Wilkins-Diehr, editors, *GCE 2011*, pages 43–50. ACM, 2011.

[44] W. M. P. van der Aalst. *Process Mining*. Springer, 2011.

[45] W. M. P. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Trans. Knowl. Data Eng.*, 16(9):1128–1142, 2004.

[46] H. M. W. Verbeek and R. P. J. C. Bose. ProM 6 tutorial. Technical report. http://www.promtools.org/prom6/downloads/prom-6.0-tutorial.pdf.

[47] D. Zong, G. Mao, and X. Wu. An intrusion detection model based on mining data streams. In R. Stahlbock, S. F. Crone, and S. Lessmann, editors, *DMIN 2008*, pages 398–403. CSREA Press, 2008.