Politecnico di Torino

M.Sc in Computer Engineering

Final thesis

Towards automatic extraction of rules for complex event processing over event-based systems.

Supervisor:

prof. Elena Baralis

External supervisor:

prof. Yanlei Diao

(Ecole Polytechnique)

Candidate:

Francesco Pierri

April 2018

Abstract

In the recent past, large scale event-based systems have gained adoption in many applications domains like finance, health-care and supply chain management.
Such architectures are characterized by flows of information where events consist of messages, new items or "changes of state", that generate asynchronous data characterized by high volumes, high velocity and high variety.

These "Big Data" streams are potentially rich of information and raise the demand for applications that allow for more accurate and faster decision making. However they need to be properly processed by means of operations such as filtering, correlating and aggregating in order to extract useful insights.

Complex Event Processing (CEP) is a novel methodology for analyzing data streams from event-based distributed databases: associating a precise semantics to the information item being processed, this paradigm allows to identify interesting composite events using a set specific rules, built on relational predicates and temporal constrains, which are matched against the primitive events in the input.
Though, manually writing these "complex" rules for performing pattern detection is usually hard and tedious and domain experts are often called in to specify these patterns.
Such a drawback clearly constitutes a limiting factor for the diffusion of CEP applications, which unlock the possibility of more proactive, reactive and predictive applications in the context of distributed architectures.

The aim of this project is, thus, to pave the way for a framework that enables to compose interesting CEP rules in a completely automatic fashion.
In an evident manner, this ambitious goal presents many open challenges such as defining a concept of interestingness for temporal pattern analysis, consistently and efficiently expressing multiple correlations between data as well as achieving a scalable solution for massive data streams.

In this work, the problem of learning automatically Complex Event Processing rules over a stream of events is tackled with a brand new approach w.r.t existing works in the literature: streams of events are first partitioned and modeled as multivariate time series; segmentation techniques are then combined with temporal pattern mining as to detect situations of interests and eventually derive potential complex rules.

The final contributions of our work consist of (1) a first analysis on the in-feasibility of a search-based brute force approach and (2) a pipeline for a batch processing of event-based data streams; the latter also comprises the development of two new kinds of time series segmentation based, respectively, on the concepts of "trend" and "level" and a simple temporal pattern mining algorithm for interval-based sequence databases.

Contents

List of Figures

Abbreviations

CEP Complex Event Processing

EPL Event processing language

FPM Frequent pattern mining

FPOP Function pruning and optimal partitioning

MK Mann Kendall test

NFA Non-deterministic finite automata

PAA Piecewise aggregate approximation

PAA Piecewise linear representation

PELT Pruned exact linear time

RFID Radio frequency identification

SASE Stream-based And Shared Event processing

SAX Symbolic aggregate approximation

SWAB Sliding window and bottom-up

TPM Temporal pattern mining

TSKR Time series knowledge representation

WBS Wild binary segmentation

Chapter 1

Introduction and background

1.1 Motivation

In recent times large scale event-based systems have become increasingly popular in several domains such as system and cluster monitoring, network monitoring, supply chain management, finance and healthcare [Wu et al., 2006].
A high variety of different applications in these domains incessantly produces large volumes of data that potentially contain useful information ready to be exploited.

Complex event processing (CEP) is a stream processing paradigm, which constitutes nowadays a crucial component in the industry world, where data are processed as streams of continuously arriving events that are matched against complex patterns as to detect interesting composite events [Agrawal et al., 2008].
The detections of events in these "Big data" streams, produced by a wide range of applications, provides the opportunity to implement reactive and proactive measures in active databases [Giatrakos et al., 2017].

These unendingly flowing data streams from heterogeneous sources (often geographically distributed) are characterized by large volumes, high rates and intricate relationships between different objects: these also constitute the main challenges for CEP applications which aim to extract high-level meaningful and actionable information in a scalable way [Zhang et al., 2014].

Some examples [Wu et al., 2006, Agrawal et al., 2008, Zhang et al., 2014] where this methodology can be applied are:

- **Real-time cluster monitoring**: there is an increasing demand to correlate performances measurements to identify unbalanced workloads or task stragglers (i.e. those that perform poorly and keep the others waiting for them).
- **Finance**: a brokerage customer might need to analyze a sequence of stock trading events which potentially constitute a new market trend.
- **Logistics**: in applications based on Radio Frequency IDentifier for tracking and monitoring one may want to detect anomalies in supply chains and track valid shipment paths.

CEP systems effectively allow to express a large set of monitoring queries using declarative languages which free the user from manual programming, which is usually complex and time-consuming.

Nonetheless, any complex event detection takes place according to user-defined rules, which describe the several predicates to satisfy in order to detect a situation of interest. Many researchers [Margara et al., 2014a, George et al., 2016a] have recently claimed that the complexity of writing such rules constitutes a limiting factor for the diffusion of CEP, as domain experts have to specify these particular triggering situations. This is particularly due to the possibility of aggregating and correlating different attributes from different event types (as we will clarify later on).

The intention of this work is to pave the way for a system which is able to extract potential interesting patterns in a stream of events, operating without any user intervention, with the ultimate goal of deriving complex event rules that can be expressed using any Event Processing Language. This objective is addressed by first partitioning and modeling event streams as time series and, consequently, combining together several data mining techniques such as segmentation, trend analysis and temporal pattern mining. The data used for the experimental phase consists of a set of different Hadoop workloads activity logs and OS metrics.

The complete software of our project can be found at:
https://github.com/Piru93/learning-cep-rules-from-multivariate-time-series

The outline of this document is the following:

- Chapter 1 contains a brief motivation along with some topics which are essential for the understanding of future sections; we describe some CEP systems, we present time series and pattern mining paradigms and eventually we describe trend analysis in time series.
- In Chapter 2 we carry out a literature review of the most pertinent works that we will refer to throughout our work and that guided our analysis.
- In Chapter 3 we show a proof for the in-feasibility of a brute force search approach which constitutes the starting point of our contributions.
- Chapter 4 contains the statement of our problem and the details of the architecture of our proposed solution.
- Chapter 5 contains the conclusions and the remarks for possible future applications of our work. An appendix is also included in the end, containing the pseudocode of the algorithms implemented in the application.

1.2 Complex Event Processing

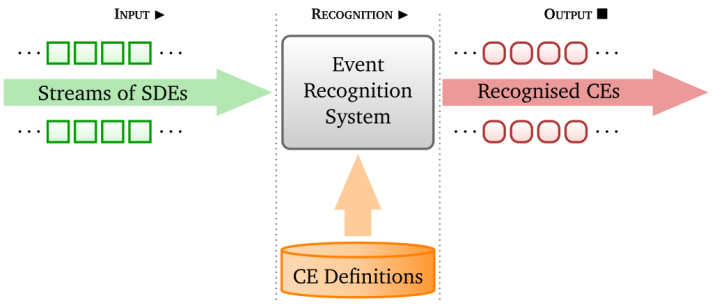


Figure 1.1: A simple model for a Complex Event Processing architecture.

As already highlighted, Complex Event Processing (or Recognition) refers to the detection of composite events in Big Data streams that present different challenging features

such as volumes, speed, uncertainties, etc [Giatrakos et al., 2017].
Both academic and industrial world offer many Event Processing Languages and CEP systems specifically built to allow queries for filtering and correlating simple events as to detect complex events which may be used as new input, each with its own architecture and processing mechanisms [Giatrakos et al., 2017].

Nonetheless, they all usually refer to the same event model where the input is an infinite sequence of events, also known as an event stream, which are instantaneous and atomic occurrences of interest at different points in time [Wu et al., 2006].
As in database systems and programming languages, where instances and types are distinguished, this model includes event types that describe a set of attributes contained by a class of event.
Moreover, each event is assigned a timestamp which reflects their true order of occurrences, which is assumed to be a natural total order (an assumption which is not always true in practice).
Defining composite rules such as predicates on attributes and temporal constraints enables therefore a CEP application to detect situations of interest by matching the events that satisfy the conditions.

The following application example from [Gyllstrom et al., 2006] might help to understand the potentialities of this technique: suppose to have a retail store where four RFID[1] readers are placed in the following locations: the store exit, two shelves, and check-out counter (see Fig. 1.2).
We are then allowed to detect the following events:

- **Misplaced inventory query**: the engine monitors for an event where a shelf item is placed on the wrong shelf by checking the attribute of the item that contains its true position.
- **Shoplifting query**: the event is triggered when an item is registered at the exit of the retail store without passing the check-out counter.

[1]Radio-Frequency IDentification is a technology based on electromagnetic waves that allows a hand-free massive identification of moving objects.

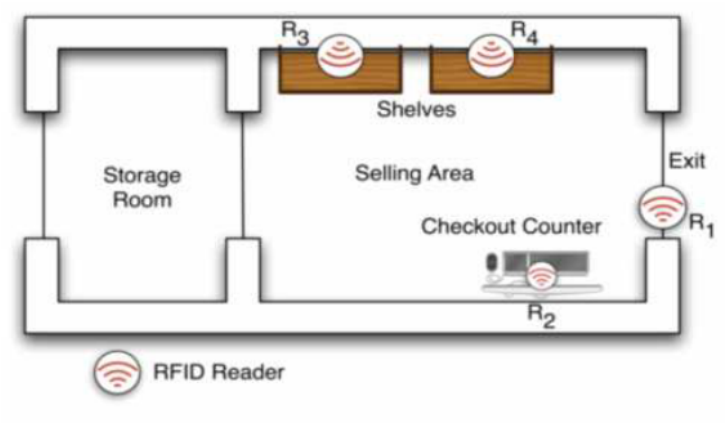


Figure 1.2: A simple retail store management setup for Complex Event Processing applications.

The different formal models that underlie existing techniques place the different systems in three main categories: **automata-based**, **tree-based** and **logic-based** models [Giatrakos et al., 2017]. A brief overview of them is provided in the next section.
However, we mainly considered the SASE (Stream-based And Shared Event processing) framework [Wu et al., 2006] as starting point for our research due to its potentialities which will be highlighted in the last section.

1.2.1 An overview of CEP systems

As previously stated, there are plenty of different Complex Event Processing (or Recognition) tools available in the literature which share common requirements such as: handling instantaneous and "durative" events as well as relational events, concurrency constraints, atemporal reasoning and event hierarchies [Giatrakos et al., 2017].

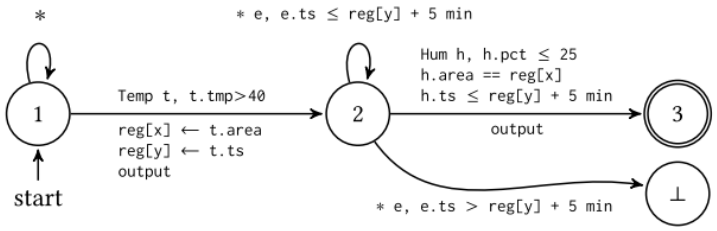


Figure 1.3: An example of NFA model for a CEP query.

Automata-based models include systems like SASE [Wu et al., 2006] and Cayuga [Demers et al., 2007], which build automata for both semantics and recognition, and

TESLA [Cugola and Margara, 2010], which uses automata only for pattern recognition.
In these frameworks, a crucial assumptions is that timestamps are non-decreasing with the arrival order of events (timestamp is treated as an additional attribute); out-of-order events would need buffering or re-ordering outside the automaton in order to be processed.
These automata are much powerful than simple regular expression matching ones: they are (1) symbolic, as edges are characterized by formulas; they are (2) register as they need some data storage for previously observed events and (3) they are also transducers, as they produce finite output rather than simple matching.
Though, a certain degree of non-determinism is intrinsically present in such systems: they, in fact, need to store at runtime all the possible candidate runs (whose size can easily become polynomial and even exponential), a problem which can be faced by using compressed forms, maximal runs mechanisms or lazy approaches.

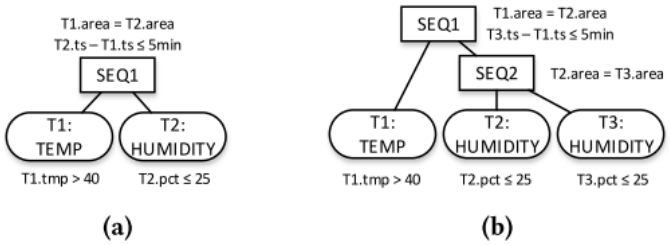


Figure 1.4: Two examples of tree-based models for the same CEP query.

Tree-based systems employ trees for both detection and recognition: Zstream [Mei and Madden, 2009] for example can handle all event operators like sequencing, negation, Kleene closure, etc, as well as predicates and constraints.
Practically speaking, they nest hierarchically event operators assigning buffers to each node, thus decoupling pattern order and recognition order (which are a major drawbacks of automata approaches) and performing recognition in a bottom-up fashion, saving intermediate results in non-leaf nodes.

Last but not least, **logic-based** models are characterized by logics used to express formal semantics (sometimes they translate rules into automata, like in T-Rex [Cugola and Margara, 2012]) and use inference to detect complex events.

```
Rule R
define  WoodFire(area: string, temp: double)
from    Humidity(percentage < 25 and area = $a) and
        last Temp(value > 40 and area = $a)
        within 5 min from Humidity
where   area = Temp.area, temp = Temp.value
```

Figure 1.5: An example of logic-based CEP query.

There are two main approaches in this field:

- Chronicle recognition, which relies on temporal logic and events are encoded using logic predicates that define occurrence and content of each event. First-order temporal logic formulas are then used to indicate conditions that are met for a particular composite event and selection and consumption policies can be customized.
- Event calculus, which is instead based on fluents, i.e. are properties that hold different values at different points in time. Recognition can be performed even when new information arrives that updates "old" one, revising the recognized events.

Some systems use Prolog to recognize events by means of logic inference mechanisms, whereas others use temporal constraint networks or automata. Recognition is usually performed at query time and windowing mechanisms exist to limit the scope and improve efficiency and scalability.

The figures in this section (respectively Fig.1.3, Fig.1.4, Fig.1.5) all refer to the following scenario [Giatrakos et al., 2017]: assume that we want to monitor the temperature and humidity in some areas to detect wood fires. Sensors report temperature and humidity values, with each event containing the area in which the sensor is located and a measurement value. As a condition for detecting a hazard, we say that wood fires are likely to occur in certain areas when humidity drops below 25% while temperature is higher than 40 degrees.

1.2.2 Stream-based And Shared Event processing

SASE is a SQL-like rich declarative language for event processing language based on automata. It was first developed in 2006 [Wu et al., 2006] and then progressively extended [Agrawal et al., 2008, Zhang et al., 2014] throughout recent past.

(a) Query 1:

```
PATTERN SEQ(Shelf a, ~(Register b), Exit c)
WHERE   skip_till_next_match(a, b, c) {
          a.tag_id = b.tag_id
    and   a.tag_id = c.tag_id
          /* equivalently, [tag_id] */   }
WITHIN  12 hours
```

(b) Query 2:

```
PATTERN SEQ(Alert a, Shipment+ b[ ])
WHERE   skip_till_any_match(a, b[ ]) {
          a.type = 'contaminated'
    and   b[1].from = a.site
    and   b[i].from = b[i-1].to }
WITHIN  3 hours
```

(c) Query 3:

```
PATTERN SEQ(Stock+ a[ ], Stock b)
WHERE   skip_till_next_match(a[ ], b) {
          [symbol]
    and   a[1].volume > 1000
    and   a[i].price > avg(a[..i-1].price)
    and   b.volume < 80%*a[a.LEN].volume }
WITHIN  1 hour
```

Figure 1.6: Three SASE queries for applications in store management, logistics and finance.

The main challenges for pattern matching over event streams addressed by SASE are mainly: **(1)** intrinsically high volume streams, **(2)** large sliding windows which can produce a lot of intermediate results, **(3)** performance requirements that put a constraint on time critical actions and **(4)** handling timestamps uncertainties due to granularity mismatches or spurious readings.

The overall structure of the language is shown in Fig.1.6, where Query 1 detects shoplifting, Query 2 contamination in a food supply chain and Query 3 a complex market stock trend.

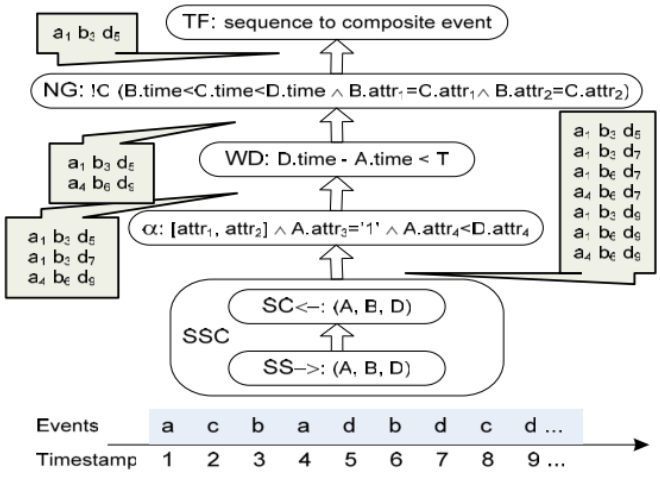


Figure 1.7: An execution plan for a SASE query.

Adopting the usual event model depicted in the previous section, events are processed with a query-plan based approach that exploits a pipeline of native operators like sequencing, negation, parameterized predicates (i.e. those who compare the attributes of different events) and sliding windows, which are then formally translated in algebraic

query expressions (see Fig. 1.7). These primitive operations are eventually translated by means of a Non-deterministic Finite Automata model (which uses a buffer as to optimize query processing), which is shown in Fig. 1.8.

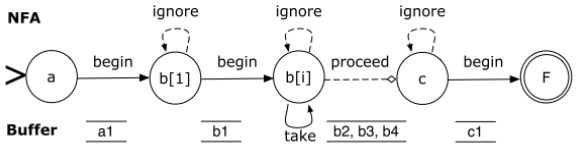


Figure 1.8: An NFA^b automaton for a SASE query.

The particularity of this language, though, is the ability to express Kleene[2] closure patterns, which make CEP much more challenging than simple regular expression matching, as they can be used to extract a "finite yet unbounded number of events of interest" [Agrawal et al., 2008]; moreover, different possible strategies (as in Fig.1.6) for selecting relevant events and termination criteria add to the richness of such a language.

Nonetheless Kleene+ [3] queries show a performance bottleneck, as the "all answers" decision problem reveals an exponential worst-case upper bound in terms of pattern matching and result construction.
On the other hand, as different features (such as Kleene+, aggregation and selection strategies) are successively included, a descriptive complexity analysis shows that the language can be cleanly mapped into a hierarchy of low level complexity classes ranging from AC0 to NSPACE[log$n$].

Finally, an imprecise temporal model [Zhangn et al., 2013], which assumes a possible set of worlds over time interval with a corresponding confidence value for matches, enables the possibility of handling of CEP scenarios with time uncertainty.

[2] A Kleene closure (or Kleene star) basically means "zero or more" in the context of regular expression (https://en.wikipedia.org/wiki/Kleene_star).
[3] The star operator is actually expressed as a "+" in the SASE language.

1.3 Pattern mining

Frequent pattern mining [Lee et al., 2014, Han et al., 2011] is one of the major problems of data mining: usually it is performed as a preliminary phase for additional tasks such as clustering, classification, association rules and outlier detection.
The traditional setting is: given a transactional database (as in Fig.1.9 D of transactions $T_1, T_2, ...T_n$, determine all patterns P with minimum support (i.e. such that they are present at least in s transactions). This support can be either a relative or an absolute value.

tid	Items
2	a,b,c,d,f,h
3	a,f,g
4	b,e,f,g
5	a,b,c,d,e,h

Figure 1.9: A transactional database.

Originally, this problem was formulated in the context of market basket analysis, where the overall task was to find association rules for customer behaviours (the "mythic" rule beers $\rightarrow$ diapers for American married men with children); the first step would consist of finding all minsup itemsets and then build rules accordingly, the former being much more challenging than the latter.

The following sections will briefly introduce the basic approach to FP along with some common variations. In the last section we discuss the core algorithms of such domain, notably *APriori* and *FP-Growth*.

1.3.1 Frequent itemsets

When focusing on mining frequent itemsets, a crucial assumption for a certain class of pattern mining algorithms is the so called *APriori* principle which enables the design of a bottom-up approach to exploring the space of frequent patterns in several passes of the database:

Every subset of a frequent itemset is a frequent itemset.

In other words, a $(k+1)$-pattern may not be frequent when any of its subsets is not frequent.

There are many ways to state the "equivalence" of all the frequent pattern mining algorithms: we will observe that they can be all seen as modifications of the following algorithm:

> given a transactional database D and a support s, build all length-one frequent patterns, then repeat the following step until all frequent patterns are found: generate a candidate pattern and count its support in the database, deciding whether it is frequent or not.

They are thus all virtually equivalent, differing only in the way they explore the space of candidate itemsets, which satisfies a downward closure or anti-monotonic property (i.e. the *Apriori* principle) and whose size is exponential w.r.t the number of transactions: *Apriori* algorithm, for instance is a join level-wise (i.e. it joins k-length frequent itemsets with themselves to build $(k+1)$-length candidate itemsets) method that explores the tree breadth-first (in a lexicographic order), while *FP-Growth* performs suffix-based recursive exploration and employs the notion of projected database to reuse the work done for counting.

Notably the key points to characterize an algorithm are the pruning technique, the efficiency at counting and the data structures employed.

Some special cases of itemsets which can be mined are:

- *closed (frequent) itemsets*, which have no frequent supersets with the same support;
- *maximal (frequent) itemsets*, which have no frequent supersets (they are indeed also *closed*).

The main difference between these two definitions is that mining maximal itemsets we lose information about the support of underlying itemsets, while closed itemsets "are a condensed representation of frequent itemsets that is lossless" [Lee et al., 2014].

An additional particular case are long patterns, where the number of sub-patterns can be very large.

Nonetheless, we can sometimes argue that the traditional support-confidence framework is unable to express certain kinds of patterns, leading to adopt other interestingness measures (such as entropy or tiles) and constraints.
To go further, a closely related issue is finding negative association rules: knowing the relationship between the absence of an item and the presence of another can be very important in some applications; however, counting absences of items is prohibitive if the number of possible items is very large, which is typically the case.

1.3.2 Sequential pattern mining

The first extension of frequent itemsets mining is sequential pattern mining
[Lee et al., 2014, Han et al., 2011] where frequent sequences are mined in a database of ordered sequences (e.g. customer transactions, DNA sequences, web log data).
We can express a database with two formats: horizontal (`seq_id` | `sequence_of_itemsets`) and vertical (`item` | `list of seq_id`) which give rise to different fashions of algorithms (see Fig.1.10).

Sequence_id	Sequence
1	$\langle a(abc)(ac)d(cf)\rangle$
2	$\langle (ad)c(bc)(ae)\rangle$
3	$\langle (ef)(ab)(df)cb\rangle$
4	$\langle eg(af)cbc\rangle$

Item	tidlist
a	1, 2, 3, 5
b	1, 2, 4, 5
c	1, 2, 5
d	1, 2, 5
e	1, 4, 5
f	2, 3, 4
g	3, 4
h	2, 5

Figure 1.10: A sequence database with horizontal and vertical notation.

This problem can be stated as:

> Given a set of sequences, where each sequence consists of a list of elements and each element consists of a set of items, and given a user-specified min-support threshold, sequential pattern mining aims to find all frequent subsequences, i.e., the subsequences whose occurrence frequency in the set of sequences is no less than min-support

Sequential frequent patterns satisfy the anti-monotony property as well and algorithms

can be generalized to two major classes: *Apriori*-based and *pattern growth*-based, whose implementations, pros and cons are derived from similar approaches for traditional support-confidence frequent itemsets mining.

1.3.3 Data streams

Data object	Pattern
Item or Itemset	Item
Itemset	Subitemset
Sequence	Subsequence
Itemset	Sequence of items spanning a sequence of itemsets
Tree	Subtree
Graph	Subgraph or subtree

Figure 1.11: Possible patterns in a data stream.

When pattern mining is applied to data streams [Lee et al., 2014] the main limitation is that only a single pass is allowed on data.
The main issues that need to be particularly taken into account w.r.t traditional techniques are especially:

1. the space of candidate patterns has an exponential blowup while streaming approaches should be linear to keep up with newly arriving data,
2. any algorithm should be memory efficient as more memory is required with a large answer set
3. a trade-off between accuracy and efficiency must be reached.

Let a streaming dataset T be a sequence of indefinite length of transacted patterns $T = T_1, T_2, ..., T_k$, where a pattern P is a sequence or set of items. At a time j we can define a finite data window $T_{i,j} = \{T_i, T_{i+1}, ..., T_j\}$ which can can contain numerous patterns.

For a given support threshold $0 < \theta < 1$ the general task is thus to find all θ-frequent patterns P, $s(P) \geq \theta$, contained with a data window $T_{i,j}$; a variant exists that seeks Top-K frequent patterns regardless of the threshold.

The support of a pattern P is defined as:

$$s(P) = \frac{\text{count}(P, \text{Patt}(T))}{|T|}$$

where the numerator is the number of times P appears in the multiset $\text{Patt}(T)$ of all patterns contained in T.
Check Fig.1.11 for a variety of sample patterns usually mined in a stream.

In conclusion, several windows can be used depending on how much consideration we want to give to all the data:

- a landmark window contains all data from the beginning to the current time t, say $T_{s,t}$
- a sliding window of width w contains data in the window $[t - w + 1, t]$
- a damped window gives more weight to most recent observations using a decay rate $0 < \delta \leq 1$ which is multiplied to the support of previously recorded patterns (e.g. after k steps a pattern P has a weight d^k) and the total support of a pattern is the sum of all time-decayed counts
- time-tilted windows are windows of varying width where frequent itemsets are mined.

1.3.4 Apriori, TreeProjections and FP-growth

Apriori method [Agrawal and Srikant, 1994] is one of the earliest algorithms [Lee et al., 2014, Han et al., 2011] for frequent pattern mining in a transactional database: it generates the candidate tree in a top-down fashion by using join operations to obtain $(k + 1)$-length possible frequent itemsets from k-length frequent itemsets.

The algorithm is straightforward: obtain F_1 and F_2, the set of 1-length and 2-length frequent itemsets (which can be mined quickly with specialized techniques), then for $k = 2$ generate F_{k+1} until $F_k! = \emptyset$, i.e. join C_k candidate itemsets (duplicates are avoided by sorting them in lexicographic order and joining only itemsets with $(k - 1)$ items in

commons), prune them by exploiting the anti-monotonic property and count their support in order to generate F_k.

Some optimizations for counting and pruning were proposed in the past:

- *Apriori*-TID (and Hybrid) consists of replacing transactions with shorter/null transactions by looking at candidate
- Direct Hashing and Pruning trims transactions and performs partial counting to prune ulteriorly candidates at each step by using an hash table
- triangular arrays and hash table are usually used together with textitApriori principle in order to efficiently count 2-length frequent itemsets
- an Hypercube representation for counting support of multiple itemsets at a time
- a lower bound on support can be used to avoid counting support: given two k-itemsets A and B with $(k-1)$ items in common we have $s(A \cap B) \leq s(A) + s(B) - S(A \cup B)$, so if the right side is greater than the threshold the computation is not performed; a corollary of this is that whenever we have $s(X) = s(X \cup Y)$, then for any $X' \supseteq X$ we also have $s(X') = s(X' \cup Y)$

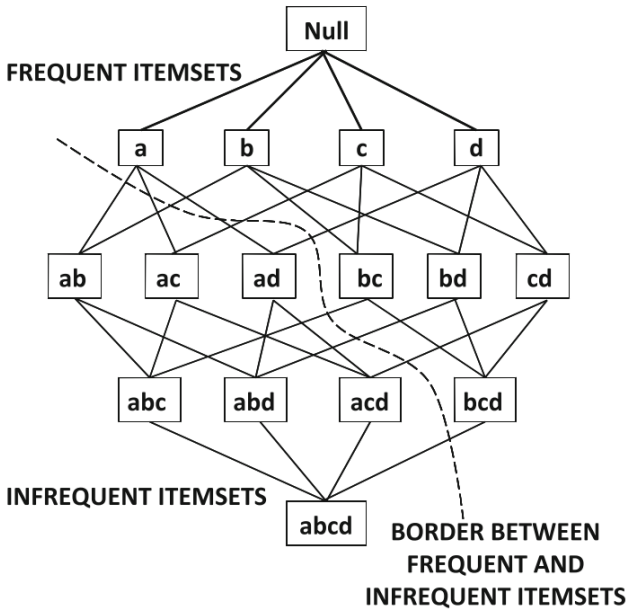


Figure 1.12: A possible lattice of itemsets.

As a matter of fact this algorithm can be seen as an enumeration-tree method which explores the lattice of candidates (Fig.1.12) in a breadth-first way, which allows for ef-

ficient pruning; depth-first methods for instance are more memory efficient and usually used for long patterns and maximal itemsets. Visualizing the enumeration tree allows for understanding what kind of gains are obtained in terms of counting, pruning and memory saving.

The tree-based algorithm is a paradigm based on set-enumeration concepts: a lexicographic ordering is assumed for items in database as to build a lexicographic (or enumeration) tree where the root is the NULL itemset and each node $I = \{i_1, i_2, ...i_{k+1}\}$ is the child of itemset $\{i_1, i_2, ...i_k\}$; we define the set of frequent extensions, i.e. 1-extensions of an itemset such that the last item is the contributor to the extension, of a node P as $E(P)$ while the prospective branches are $F(P)$, i.e. the items in $E(Q)$ where Q is the immediate ancestor of itemset P, and correspond to possible frequent lexicographic extensions of P (see Fig.1.13.
We can observe that *Apriori* can be explained in terms of the enumeration tree as candidates are be generated by joining two frequent siblings.

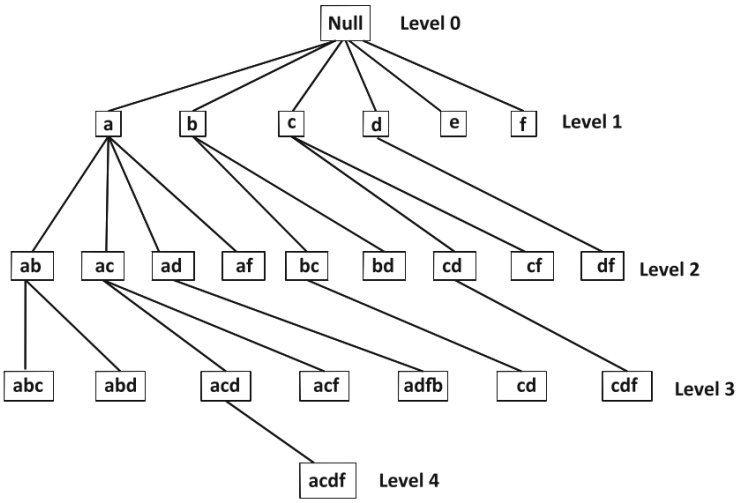


Figure 1.13: The lexicographic (or enumeration) tree.

Other methods [Lee et al., 2014, Han et al., 2011] achieve better counting strategies by reusing work done for k-candidates to $(k+1)$-candidates; the primitive AIS algorithm simply builds the enumeration tree, though not presenting it as such, in level-wise fashion eventually counting supports and with no pruning nor any optimizations techniques.
Different exploration strategies are proposed, instead, together with recursive projections

in the TreeProjection algorithms, each of which with different advantages depending on the task.

The main difference with FP-growth is the way the projected databases are internally represented, but still the important shared observation is that any non-relevant transaction for a node is discarded when considering descendants as well.

A simple depth-first version of *TreeProjection* [Agarwal et al., 2001] consists of recursively extending frequent prefixes and maintaining only transaction database relevant to the prefix (a naive optimization would be to project only when the conditional database changes size), never repeating the counting work done at the higher levels: given a database D, a minimum support s and a current pattern prefix P (the function is recursive and first run with the root which is NULL), count frequent 1-items, remove infrequent items and for each frequent item i build a conditional database for the extended pattern $(i \cup P)$ also removing lexicographically lower items, then recur. A breadth-first version would have the overhead of considering all the projected databases at once.

The projected databases in *TreeProjection* are usually implemented by means of triangular arrays or bitstrings; *FP-growth*, instead, is a suffix-based recursive pattern growth method [Han et al., 2000]

The algorithm consists of two steps: first build the FP-tree, than recursively call *FP-growth* function on each item.
In order to build the tree, a first scan is performed to remove infrequent items from the transactions and store frequent 1-itemsets (items in each transaction are also ordered decreasingly by support); then, a Trie is built by reading each transaction and incrementing a counter when a node is traversed or allocating new nodes with initialized counter equal to 1 when reaching beyond a prefix; eventually pointers for each item are "chased" and a header table is built: this contains on each row an item, its support count and a link to left-most node in the Trie; items are also ordered decreasingly by support. A sample of FP-tree construction and final result are in Fig. 1.14.

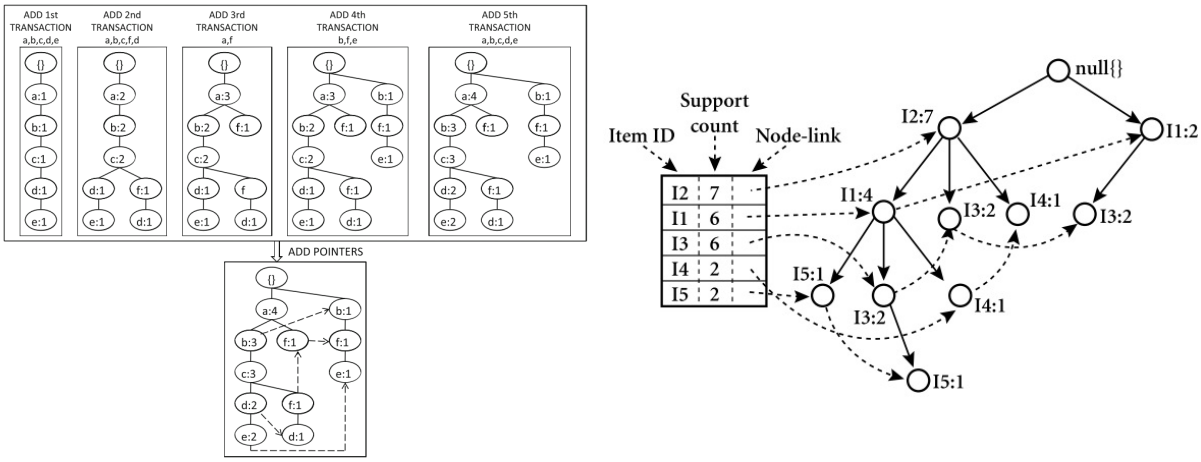


Figure 1.14: A FP-tree construction and a resulting structure.

The "tricky" part is calling recursively the FP-growth function on each item of the header table, starting from the bottom: for a given suffix i a new FP-tree can be built (called conditional FPT_i) by extracting transactions containing i and counting the support, as to remove infrequent items in the projected database which usually consists of a different representation w.r.t the original FP-tree (the primary bottleneck is locating instances of items and counting supports for each in order to built the Trie). An example is given in Fig.1.15.

In the case of very large databases, partitions must be created because the Trie may not fit into memory.

A performance comparison between *FP-growth* and *Apriori* is not straightforward: it is true that for certain transaction data sets, the former outperforms the standard *Apriori* algorithm by several orders of magnitude.

But generally, if the resulting conditional FP-trees are very bushy (in the worst case, a full prefix tree), then the performances of the *FP-growth* algorithm degrade significantly because it has to generate a large number of subproblems and merge the results returned by each subproblem.

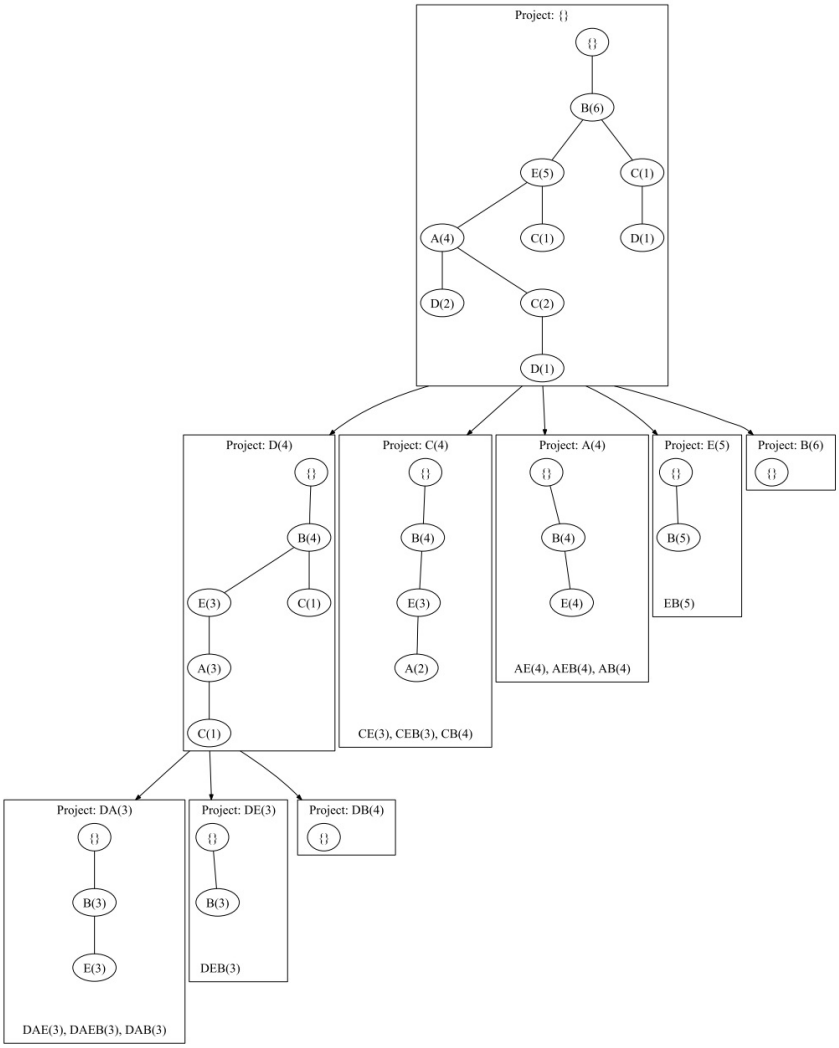


Figure 1.15: An execution of the algorithm.

1.4 Time series

Amongst the different classes of temporal data, time series have risen an increasing interest in the scientific community in the recent past [Yang et al., 2006].
A time series is an ordered sequence of points, a collection of measurements performed over time as the result of an underlying process.

A common assumption is that these observations are collected at time instants which are uniformly spaced according to a given sampling rate (which does not hold always in practical world).
Moreover, we usually distinguish between univariate time series which regard a single

variable and multi-variate time series where, instead, we have multiple univariate time-series defined on the same time range [Esling and Agon, 2012a][Fu, 2011].

They currently have several fields of applications, from financial markets to climate change, and they present many challenges due to their continuous nature, noise and dimensionality (N.B. the length rather than the space size of the observations).
Time series are usually shown by interpolating points (i.e. connecting one point to another) as to obtain the result of Fig.1.16.

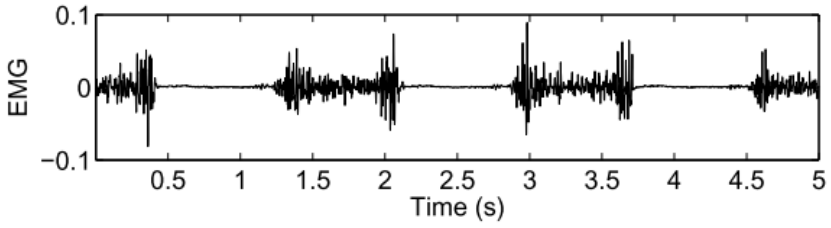


Figure 1.16: A time series of an Electromyography (EMG).

Mining time series is currently considered one of the most challenging problems in data mining [Yang et al., 2006]. Common data mining techniques [Esling and Agon, 2012b] include for instance:

- Indexing: given a query and some similarity/dissimilarity measure, find the most similar time series in database.
- Clustering: find appropriate groupings of the time series in a database using some similarity/dissimilarity measure.
- Classification: given an unlabeled time series, assign it to one or more predefined classes.

In Section.2 we will discuss a few time series representations (SAX and PLR) and introduce changepoint analysis[4].
We will not mention time series decomposition models as they mainly deal with forecasting (we refer the reader to [Hyndman and Athanasopoulos, 2017] for an introduction)

[4]These two topics actually intersect segmentation, dimensionality reduction and pattern discovery.

which does not relate to our project.

For what concerns time series representation, we will just provide the classical taxonomy [Moerchen, 2006] (see Fig.1.17) which is the following:

- *non-data adaptive*, which use always the same parameters for the transformation, regardless of the data (e.g. wavelet transformations such as HAAR, DCT and DFT); they have a fixed size
- *data adaptive*, whose parameters for the transformation depend on the data (e.g. SVD, SAX, shapelets); they also have a fixed size
- *model*, which assume an underlying model for the data and try to fit them to estimate parameters; models can capture the structure of the data generating process.

Sometimes we find an additional class which consists of *Data dictated* representations, which do not allow the user to decide the level of the detail as the size of the representation is automatically determined (as in "clipping" or "grid-based" methods).
Again, we refer the interested reader to [Fu, 2011] for an overview of existing techniques.

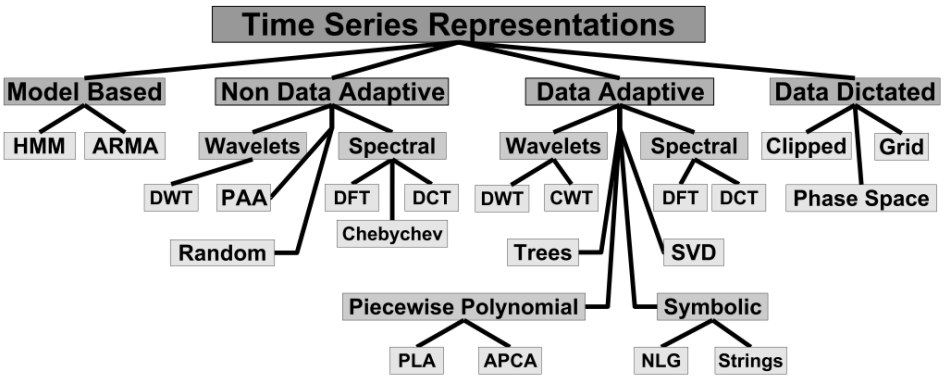


Figure 1.17: Categorization of time series representations.

1.5 Trend analysis for time series

When examining a series of observations over time a frequent practice is a trend analysis, i.e. checking whether the values of a random variable consistently increase (or decrease)

over some period of time in statistical terms [Helsel and Hirsch, 2002]. A general definition of trend for time series is:

A significant change of a variable of interest over a specific time interval

The tools usually employed for accomplish such goal are statistical tests such as Mann-Kendall test or Student's t-test.
However, while a statistical trend analysis can help to identify trends and estimate their rate of change, it will not provide much information on the particular cause causing the trend, which usually requires much domain knowledge [Yue et al., 2002].

The traditional testing scenario consists then of the null hypothesis $H_0 = \{\text{no monotonic trend}\}$ against the alternate hypothesis is $H_a = \{\text{there exists a monotonic upward or downward trend}\}$. Usually a significance value α is specified: this correspond to the Type I error or the probability of getting a false positive, i.e. the probability of falsely accepting the alternate hypothesis.
The power of a test is defined as $(1 - \beta)$ where β is the Type II error or the probability of getting a false negative, i.e. falsely rejecting the null hypothesis (see Fig.1.18 for a summary).

	True Situation	
Decision	No trend. H_0 true.	Trend exists. H_0 false.
Fail to reject H_0. "No trend"	Probability = $1-\alpha$	(Type II error) β
Reject H_0. "Trend"	(Type I error) significance level α	(Power) $1-\beta$

Figure 1.18: Errors and probabilities for a statistical test.

The tests previously mentioned actually differ in nature: t-test is a parametric method (i.e. it makes certain assumptions on the distribution of the data) while Mann-Kendall is a non-parametric one (i.e. "distribution free").

Parametric tests for trend analysis basically assume that the residuals (i.e. the differences between actual and predicted values of the dependent variable) are normally distributed and homoscedastic (homogeneous variance); the t-test, in particular, is used to check the

existence of a linear trend.

Non-parametric tests like Mann-Kendall, instead, make no assumptions on the distribution and thus they work better on skewed (i.e. a distribution of values which is asymmetric w.r.t the mean value) and non-normal distributions; this test has also been proved to be equivalent to Spearman's one and it is usually less affected by outliers [Meals et al., 2011].

Another remark [Helsel and Hirsch, 2002] is that if only a test for trend is of interest, then monotonic transformations of the data are of no consequence when a nonparametric test is used; such tests are in fact invariant to monotonic power transformations (for example the logarithm or square root).

The decision to transform data is, however, highly important for computing and expressing slope estimates. Trends which are nonlinear (say exponential or quadratic) will be poorly described by a linear slope coefficient, whether from regression or a nonparametric method.

In the end, a time series trend analysis should take into account different aspects of data in order to adjust the technique employed: type of distribution (normal, skewed, symmetric, etc), presence of outliers, presence of seasonality, correlations and missing values [Onoz Bayazit, M., 2003].

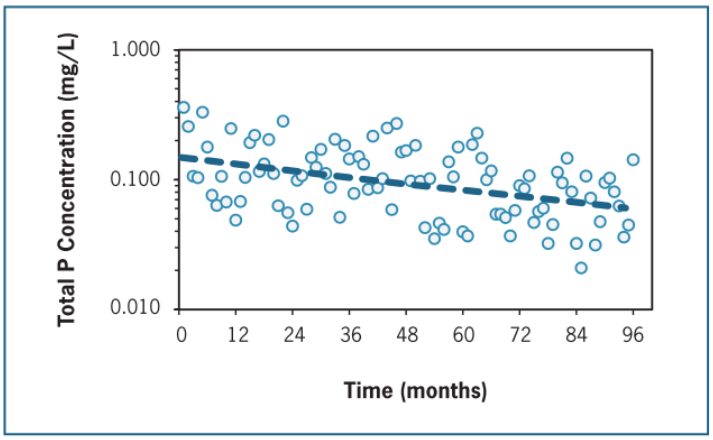


Figure 1.19: Linear regression on a sample of data (Eight years of monthly total phosphorus concentration data from Samsonville Brook, a stream draining a Vermont agricultural watershed[Meals et al., 2011])

For instance, if the assumptions for linear regression are met, notably "Y is linearly related to t, residuals[5] are normally distributed, residuals are independent, and variance of residuals is constant", then data can be interpreted according to the model:

$$Y = \beta_0 + \beta_1 X + \epsilon$$

and the null hypothesis H_0 is that β equals zero; a (Student's) t-test is then performed to check significance[6]. See Fig.1.19 for an example of data where this conditions are met and the linear regression works quite fine.
Otherwise, a non-parametric test may be more suitable for the analysis.

During our work we specifically focused on the non-parametric Mann-Kendall test which will thus be described with a bit of detail in the next section.

1.5.1 Mann-Kendall test

The purpose of this test is to statistically assess if there is a monotonic upward (or downward) trend of the variable of interest over time, i.e. the variable consistently increases (decreases) through time, but the trend may or may not be linear.
The MK test can be used in place of a parametric linear regression analysis, which tests if the slope of the estimated linear regression line is different from zero.

The regression analysis requires that the residuals from the fitted regression line be normally distributed; an assumption not required by the MK test, that means, once again, that this is a non-parametric (distribution-free) test.
It does actually have two weak requirements:

- a value can always be declared less than, greater than, or equal to another value
- observations are independent

The MK test works as follows: given n input points x_i, ordered according to their occur-

[5] https://en.wikipedia.org/wiki/Errors_and_residuals
[6] https://en.wikipedia.org/wiki/Student%27s_t-test#Slope_of_a_regression_line

rence timestamp, it first computes a sum S:

$$S = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} sgn(x_i - x_j)$$

which intuitively tells that, if positive, observations obtained later in time tend to be larger than observations made earlier; if S is a negative number, instead, observations made later in time tend to be smaller than observations made earlier.

Using S we can compute the variance (which also takes into account tied groups t_p, i.e. groups of points with same value):

$$\mathrm{VAR}(S) = \frac{1}{18}\Big[n(n-1)(2n+5) - \sum_{p-1}^{g} t_p(t_p-1)(2t_p+5)\Big]$$

and a statistic test:

$$Z_{MK} = \begin{cases} \frac{S-1}{\sqrt{\mathrm{VAR}(S)}} & \text{if } S > 0 \\ 0 & \text{if } S = 0 \\ \frac{S+1}{\sqrt{\mathrm{VAR}(S)}} & \text{if } S < 0 \end{cases}$$

which we can use to test the different alternative hypothesis (e.g. at the Type I error rate α, where $0 < \alpha < 0.5$):

- Upward monotonic trend: it is accepted if $Z_{MK} \geq Z_{1-\alpha}$
- Downward monotonic trend: it is accepted if $Z_{MK} \leq -Z_{1-\alpha}$
- Upward or downward monotonic trend (two-tailed test): it is accepted if $|Z_{MK}| \geq Z_{1-\alpha/2}$

where $Z_{1-\alpha}$ is the $100(1-\alpha)^{th}$ percentile[7] of the standard normal distribution.

[7]The value p that holds $p\%$ of the values below it

A final and interesting remark is that S is equivalent to Kendall's tau or coefficient:

$$\tau = n_c - n_d$$

where n_c and n_d are respectively the number of concordant and discordant pairs in the sequence[8].

This allows for optimizations in computing the test whose naive implementation has a quadratic complexity (simply looping on all pairwise differences).

We refer the interested reader to the following works: a specific better approach to compute it in $O(n \log(n))$ exists[9] in [Knight, 1966] while a $O(n \sqrt[2]{\log(n)})$ algorithm can be derived from [Chan and Pătraşcu, 2010] (where the problem is actually to find the number of inversions in an array, which can be adapted to compute Kendall's rank coefficient).

[8] https://en.wikipedia.org/wiki/Concordant_pair

[9] See also http://adereth.github.io/blog/2013/10/30/efficiently-computing-kendalls-tau/.

Chapter 2

Related work

2.1 Mixed-initiative approaches to exploring massive databases

In this chapter we provide a literature review of some interesting works which address the automatic exploration (by means of visualizations or anomaly detections) of large databases with little to none intervention of the human user.
This strictly relate to our very goal of performing an automatic understanding of a stream dataset as to extract processing rules.

2.1.1 SeeDB: Efficient Data-Driven Visualization Recommendations to Support Visual Analytics

Nowadays, visualizations have become more and more important as a starting point for data analysts to carry out their data exploration; though, they still need a tedious hit-and-trial approach in order to find interesting views.

SeeDB [Vartak et al., 2015] is a "mixed-initiative" middleware interface conceived to provide the user with interesting visualizations, given a subset of the data to explore (e.g. a query), adopting a deviation-based metric to indicate trends and interesting views: *a*

visualization is likely to be interesting if it shows a large deviation from some reference. Nevertheless, some other factors can sometimes affect judgment, like aesthetic, particular attributes or even lacks of deviation. The problem is stated as follows:

> given a snowflake schema (i.e. facts and dimensions), a set of aggregate functions and a query (in the selection-join-projection paradigm), find the top-k aggregate views scored by an utility metric (Earth's mover by default).

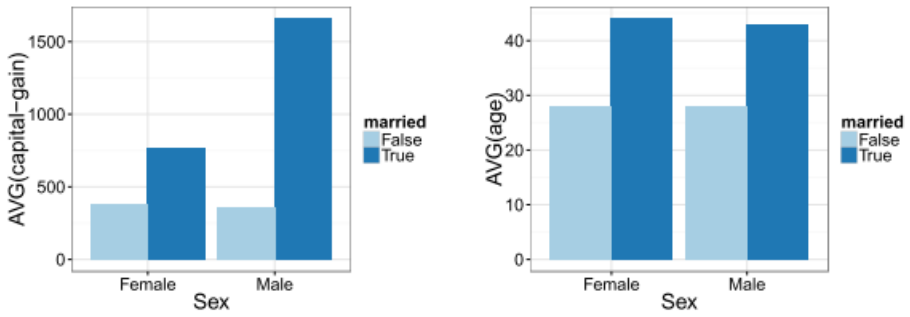


Figure 2.1: An interesting and a uninteresting visualization in the SeeDB framework.

As they deal with major issues such as the high number of candidate visualizations, the repeated computations and the performance constraints for interactivity, the framework is developed in a phase-based way, i.e. each phase operates on a "i-th" portion of the whole dataset, updating partials results and pruning intermediate ones when possible.

In order to prune aggregate views on the fly two interesting techniques are proposed: a confidence interval-based one (using Hoeffding-Serfling inequality) and a varied multi-armed bandit approach (with Successive Accept and Rejects algorithm); these approaches originally present particular assumptions, which are guaranteed by a weak consistency property on the utility metric (i.e. low utility views are pruned with high probability).

In conclusion, they develop a framework which constitutes an important first step in the exploration of visualization recommendation tools, which pave the way towards rapid visual data analysis.

2.1.2 Extracting Top-K Insights from Multi-dimensional Data

OLAP tools indeed allow companies to make better and faster decisions, but they still require the user to manually specify attributes and dimensions. Given the concept of insight, which captures interesting observations from multiple aggregation steps, authors of [Tang et al., 2017] build a system that allows also non-expert users to find useful information in massive data, starting from interesting views provided in an automatic fashion.

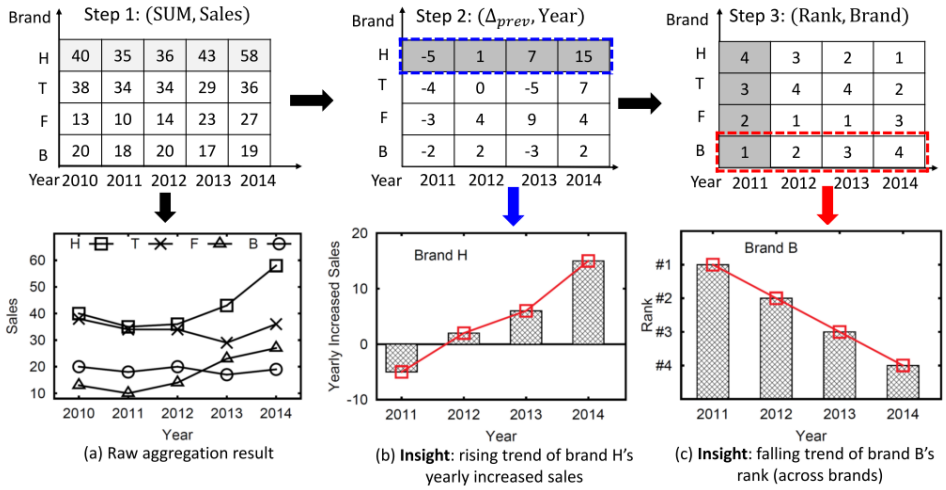


Figure 2.2: Examples of insights in Top-K.

The main challenges in such a context are, consequently, a huge search space, expensive computations and the non-monotonicity of the insight score (that prevents from using existing aggregation computation methods).

Providing a formal model on top of a multi-dimensional dataset (subspaces, sibling groups, extractors and composite extractors) exceptional facts and unexpected trends are defined via insight types (such as point or shape types).

The problem is then stated as follows:

> find top-k insights with the highest scores among all possible combinations of sibling groups, composite extractors and insight types (given a dataset and a depth), while achieving effectiveness and efficiency in terms of results and computations.

They propose a scoring function built as a combination of an impact and a significance measure (based on the p-value definition) and they allow for extensibility of aggregate

functions and extractors for expert users, as well as customized scoring functions, insight types and constraints on the search space; eventually, they provide support for any kind of dataset in the OLAP system.

Ultimately, this application is a first attempt to extract insights hidden in the data, with practical usages in business intelligence applications such as providing informative summaries to non-expert users and guiding directions for data exploration by means of customizable insights for expert users.

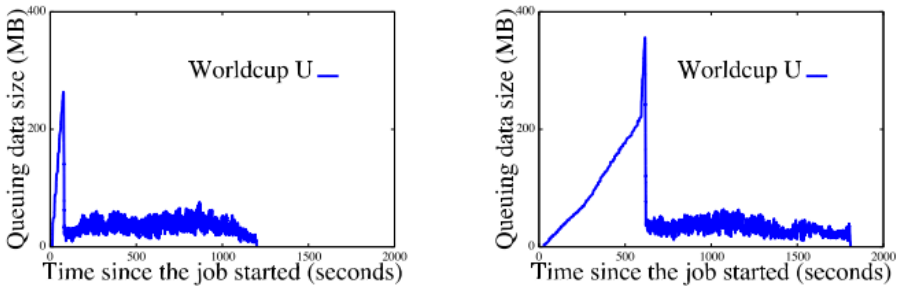
2.1.3 EXstream: Explaining Anomalies in Event Stream Monitoring

The aim of this work [Zhang et al., 2017] is to provide "high-quality explanations" to anomalies encountered in CEP-based monitored results, which are manually annotated by a user.
Declaring a formal set of requirements for an explanation (conciseness, consistency and predictive power) and the definition itself of "optimal explanation", the problem of generating and selecting features is tackled along with an evaluation on two real-world cases (Hadoop cluster monitoring and manufacturer's supply chain).

The motivation behind this is that CEP technology is nowadays at the core of real-time monitoring in several areas of application including IoT, financial market analysis and cluster monitoring; though, only in a passive fashion where the user explicitly defines patterns of interest.
One step further would be to automatically detect interesting patterns (e.g. anomalous behaviours) providing, at the same time, with an explanation that could allow the users to intervene immediately in what is called a proactive monitoring.

The idea is to let the user manually annotate the abnormal values requesting the platform to search for an explanation from the archived raw data.
Existing techniques (like regression or decision trees) are designed for prediction rather

than consistency or conciseness and sometimes analyze only the data explicitly involved in the analysis (i.e. the tuples selected by a query) and do not look for other causes in the full dataset.



(a) Data queuing size of a normal Hadoop job

(b) Data queuing size of an abnormal Hadoop job

Figure 2.3: Hadoop cluster monitoring example in EXStream.

The EXstream system is actually built on the SASE framework (1.2.2) but the results can be extended to other CEP languages.
Going back to the problem, they claim that an expert user should check all the dataset in order to identify the causes of such anomalies (e.g. the Hadoop cluster monitoring logs). The system provides a new heuristic method to tackle the problems:

- it searches and provides a sufficient feature space (raw data do not necessarily carry all necessary features)
- it proposes an entropy-based single-feature reward function (i.e. the larger the distance, the more reward produced)
- it subsequently identifies a cut-off to reject features with low rewards and it uses a correlation-based filtering to eliminate overlapping features (emulating the submodularity property).

In order to obtain a reasonable feature space, the CEP system records all the events (and the matching ones) with all their attributes in order to produce different raw features (i.e. time-series for each attribute); these are put together using aggregation functions with sliding windows in order to obtain higher-level features which can show more general trends and smooth outliers.

The performances evaluation on real-world cases shows that this kind of approach outperforms other techniques such as logistic regression, decision trees, data fusion and majority voting in terms of conciseness and consistency, while showing a highly competitive prediction power.

2.2 Time series data mining techniques

As we will see in 4.1 an intermediate goal will be to find a good representation for time series in order to apply frequent pattern mining (which is usually performed on a transactional database) and extract interesting correlations between different event types and attributes. The approach of detecting trends into a time series, for instance, is a well known research problem strictly related to pattern and motif discovery as well as segmentation and discretization. In the following sections we'll discuss some existing techniques that guided our analysis, notably the SAX and PLR representations and the changepoint analysis.

2.2.1 Symbolic Aggregate approXimation

According to [Lin et al., 2003], many symbolic representation of time series have several disadvantages:

1. they do not scale with dimensionality
2. distance measures defined on these representations do not correlate well with original time series
3. they need to have access to all the data before creating the symbols (this last features seriously affects a streaming context).

A critical achievement in time series representation is that any ultimate data mining technique applied as a second-stage should perform identical results on a symbolic time series as on the original one, which is exactly what authors of this work claim, along with a lower bound on corresponding distance measures when developing SAX.
Their approach thus reduces dimensionality in a streaming fashion while obtaining lower

bounding.

We define the **lower bounding** property as:

> the guarantee that a defined distance measurement on the new representation of data is less than or equal to the true distance measured on the raw data.

According to [Faloutsos et al., 1994]: "it is this lower bounding property that allows using representations to index the data with a guarantee of no false negatives".

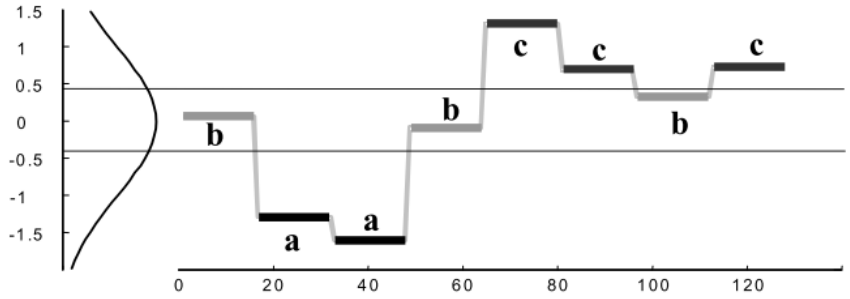


Figure 2.4: PAA coefficients of a time series discretization are mapped into SAX symbols.

This technique works by firstly normalizing each time series to have a mean of zero and standard deviation of one, then transforming it with Piecewise Aggregate Approximation (the lower bound proof derives in fact from PAA definition), which is basically a segmentation based on moving average calculation[1], and eventually symbolizing each representation with a discrete string. Breakpoints for the last step are obtained by looking them up in a table of Gaussian quantiles.
They correspond to a sorted list of numbers such that the area under a $N(0,1)$ from one to another is $1/a$, as we want to produce a equal-sized areas under a Gaussian curve.

2.2.2 Piecewise Linear Representation

In [Keogh et al., 2001] we find an exhaustive discussion existing techniques for Piecewise Linear Representation, one of the most used time series representations usually employed as a first stage for several data mining techniques like classification, clustering or indexing.

[1] A time series is divided into K segments, each of which is mapped to the average value of the original time series in that interval.

The key point is the dimensionality reduction that can be achieved with such technique.

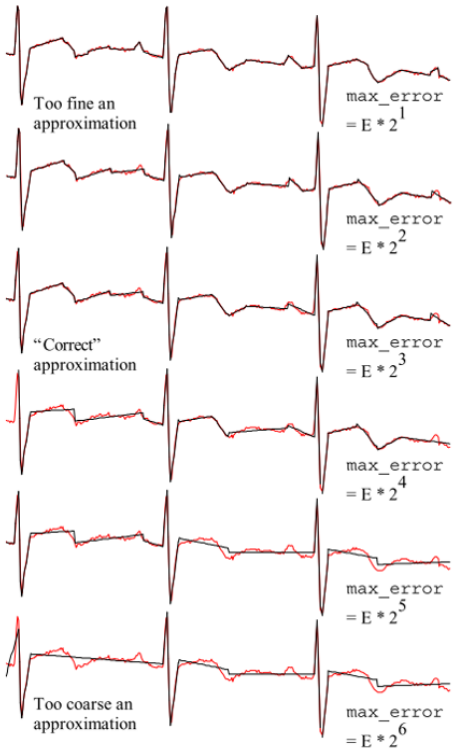


Figure 2.5: PLR with different maximum error thresholds.

PLR basically consists of performing Piecewise Linear Approximation to segment time series (see Fig.2.5 for an example); the problem is to find the best way of representing a time series by means of straight lines and additional constraints such as:

- using only K segments (where K is chosen by the user)
- using a threshold for the maximum allowed error on a single segment
- using a threshold for the maximum combined error of a segmentation

Usually interpolation or regression are the methods used to find these segments, while common error measures are sum of squared residuals or absolute norm (i.e. the difference between the approximating segment and the farthest point in the vertical direction).

The three main categories of algorithms identified in such work (and that inspired some our segmentation algorithms) are the following:

- Sliding windows (online): a segment is grown until a condition is met (e.g. an error

bound is exceeded); the process repeats with the next data point not included in the newly approximated segment.

- Top-Down: the time series is recursively partitioned until some stopping criteria is met.
- Bottom-Up: starting from the finest possible approximation, segments are merged until some stopping criteria is met.

Results show that the first approach holds the worst results while the last one is the most promising one, also scaling linearly w.r.t the size of the input.

Ultimately, authors propose a new method called SWAB (Sliding Window And Bottom-up) which combines the best of both worlds, notably the on-line flavor and the efficiency/accuracy: they use a sliding window with a size big enough to contain (ideally) 5-6 segments and they apply bottom-up to retrieve the leftmost segment.

2.3 Changepoint detection

Changepoint detection is the problem of estimating the point(s) at which the statistical properties (usually the mean or the variance) of an ordered sequence of data change, with applications in many areas as climatology, bioinformatics or finance. An example is shown in Fig.2.6

Finding multiple changepoints, which is the natural generalization of finding a single changepoint, is actually one approach to deal with *segmentation*, as they split the data into a set of segments, each of which can be summarized by a set of parameters.

An interesting case for our context is that in which the number m of changepoints is not known apriori; given a sequence y and a time τ, one formulation of such problem, known

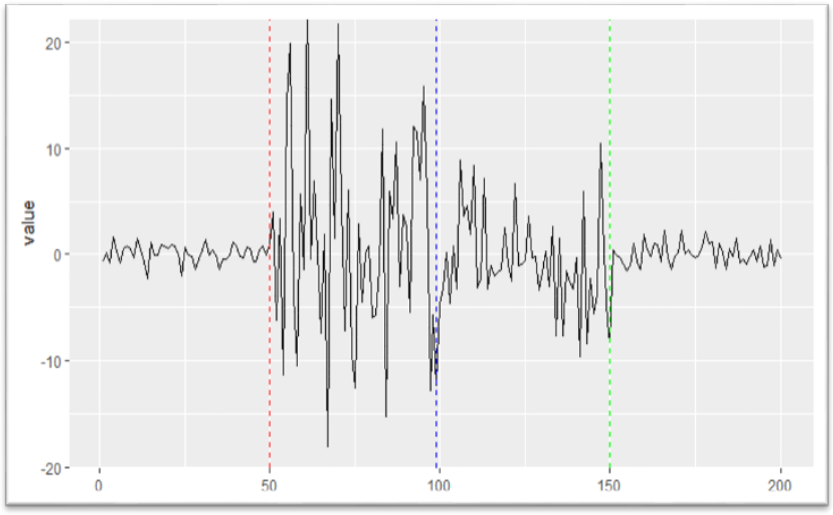


Figure 2.6: An example of changepoint analysis with three different changepoints highlighted with dashed lines of different colours.

as *penalized optimization*, consists in minimizing the following function:

$$\min_{m} \sum_{i=1}^{m+1} [C(y_{(}\tau_{i-1}+1) : \tau_i] + \beta f(m)$$

where C is a cost function for a segment, e.g. log-likelihood, and β a penalty function (such as Schwarz Information Criterion).

A first naive approach would be to enumerate all the possible combinations, holding an exponential cost w.r.t the input size.
In the following, we propose some state-of-the-art existing approaches for changepoint analysis in time series.

2.3.1 (Wild) Binary segmentation

Binary segmentation [Scott and Knott, 1974] is indeed the most used technique for changepoint analysis.
It consists of a greedy procedure that works as follows:

- Apply a single changepoint test statistic to the entire data, if a changepoint is identified the data is split into two at the changepoint location.

- The single changepoint procedure is repeated on the two new subsets, before and after the change. If changepoints are identified in either of the new subsets, they are split further.
- This process continues until no more changepoints are found in any of the subsets.

It is a greedy and approximate algorithm but it is computationally fast as it only considers a subset of the $2^{(n-1)}$ possible solutions, with a computational complexity T(N)= $O(n \log(n))$.

An improved version of such algorithm was proposed in [Fryzlewicz, 2014], notably Wild Binary segmentation.
It is a a variation that counters the drawbacks of the "greedy" flavor of Binary Segmentation, as searching for a single change-point in each stage makes the method unsuitable for some functions containing multiple change-points in particular configurations.
In fact, they show that a minimum spacing is needed to ensure consistency of BS results, as side results of their paper.
Therefore, they propose an algorithm that work as it follows:

- Randomly draw (hence the term "Wild") a number of subsamples of variable length and compute a test statistic on each subsample.
- Choose the most significant changepoint candidate (according to the CUSUM[2] statistic) and test it against a stopping condition.
- If it is considered to be significant, the dataset is split and the same procedure is then repeated recursively to the left and to the right of the changepoint.

The advantages of such approach are shown in Fig.2.7, where the input sequence is represented as:

$$X_t = f_t + \epsilon, \quad t = 1, ..., T$$

where the residuals are normally distributed with zero mean and unit variance and is a deterministic, one-dimensional, piecewise-constant signal.

[2] https://en.wikipedia.org/wiki/CUSUM

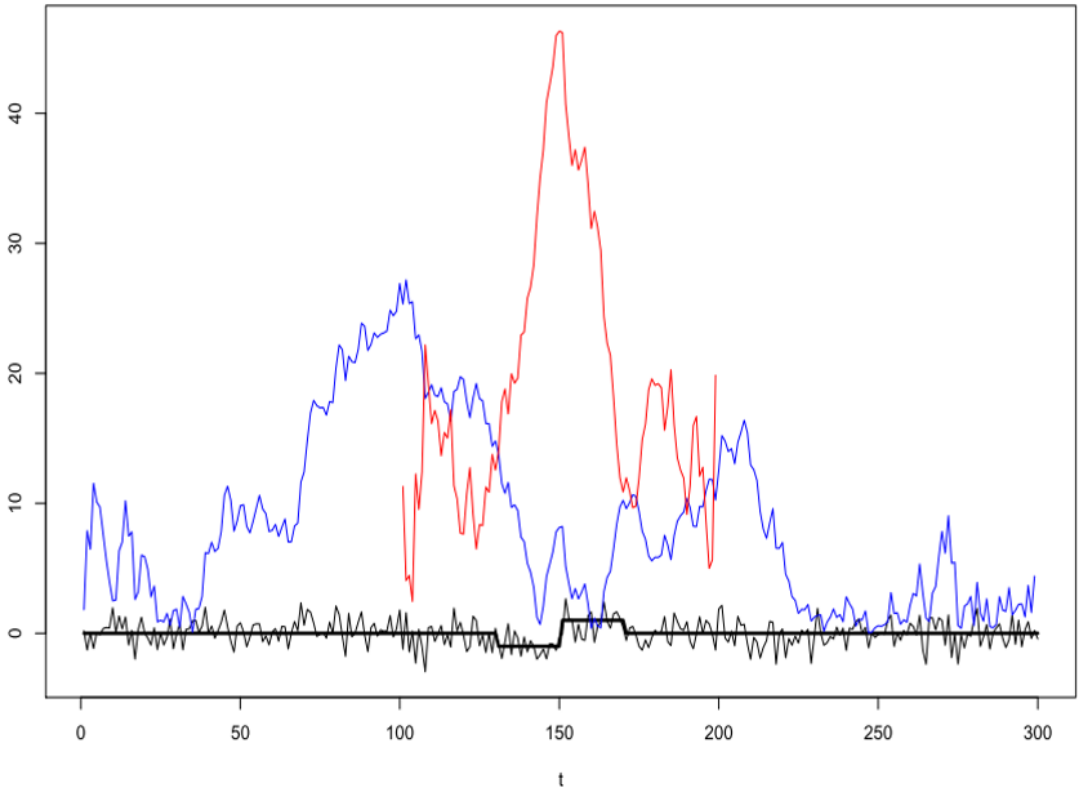


Figure 2.7: A sample scenario for Wild Binary Segmentation: there are shown the true function (thick black) and observed one (thin black), CUSUM value for each point for the entire sequence (blue) and for a sub-sample (red).

2.3.2 Pruning exact methods (PELT and OPFP)

Another approach for changepoint analysis is to minimise a cost function over possible numbers and locations of changepoints.

In the following we will examine two different methods that are exact, i.e. they find the minimum of such cost functions (and thus the optimal location of these changepoints), and that exploit pruning to avoid repeated computations.

A key difference between these methods and binary segmentation is that when a new changepoint is added in the exact methods, the location of previous changepoints may change to accommodate the result, while in the Binary Segmentation method(s) previously identified changepoints are fixed, as the result of a greedy and approximated approach.

Authors of [Killick et al., 2012] introduce an exact method called **P**runed **E**xact **L**inear

Time that under certain mild assumptions[3] performs in a linear time w.r.t data points. It is important to notice that in such context accuracy often comes at expenses of performances (as for binary segmentation).

What PELT does is to apply an optimal partitioning method that begins by first conditioning on the last point of change and then calculates the optimal segmentation of the data up to that changepoint.

In this way the calculation of the global optimal segmentation is performed using optimal segmentations on subsets of the data,i.e. a recursive form to the method is given as the optimal segmentation for data $y_{1:\tau^*}$ is identified and then used to "inform" the optimal segmentation for data $y_{1:(\tau^*+1)}$.

The pruning intuition simply consists in removing those values that can never be minima in further computations; this happens if the cost function is such that when adding a changepoint the cost of the sequence reduces.

The same authors further propose a non-parametric version of such approach in [Haynes et al., 2017], that uses the empirical distribution with a given number of quantiles (they suggest $4\log(n)$ where n is the input size) and an algorithm to select the best number of changepoints for a range of manual penalties, notably CROPS, in [Haynes et al., 2014]: it is an iterative procedures that efficiently computes the cost function reusing previous computations.

This last contribution is particularly interesting as the main issue of any changepoint analysis is selecting the parameters, which includes choosing a penalty (ranging from SIC to log-likelihood to manual values).

As they suggest one can eventually use the "elbow"[4] method to choose the correct number of changepoints as shown in Fig.2.8.

Authors of [Maidstone et al., 2017] provide an alternative technique **F**unction **P**runing and **O**ptimal **P**artitioning that performs better than PELT in certain scenarios, with a computational cost which is competitive w.r.t binary segmentation.

This is due to the fact that this method prunes "more" than PELT, thus storing less candidates and being more efficient; nonetheless their assumptions for functional pruning are quite strong (we refer the interested reader to [Maidstone et al., 2017]).

[3] We refer the reader to [Killick et al., 2012] for a clear analysis.

[4] https://en.wikipedia.org/wiki/Elbow_method_(clustering)

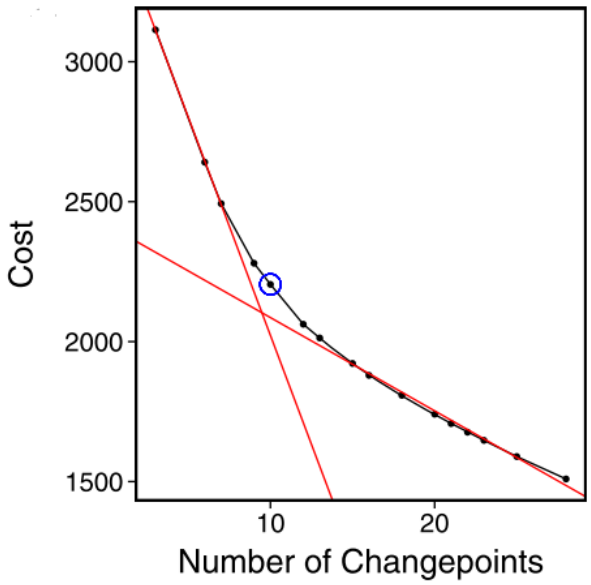


Figure 2.8: A "cost" vs "number of changepoints" plot. The red lines and the blue circle highlight what is retained as the centre of the elbow.

2.3.3 Event detection in time series data

In [Guralnik and Srivastava, 1999], authors tackle the problem of detecting changepoints, which they refer to as "events" and "episodes", in time series data, i.e. points where the "behaviour" of a dynamic phenomenon, modeled as a time series, qualitatively changes (e.g. the change of highway traffic from light to heavy to congested).

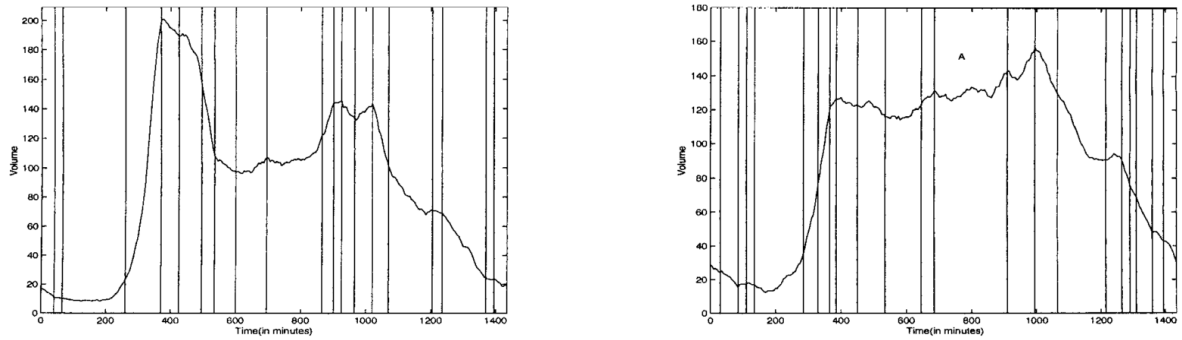


Figure 2.9: Changepoints analysis on data taken from highway traffic sensors, called loop detectors, in Minneapolis-St.Paul.

Overall, they determine the number of such points and the model to fit the data in each "segment" by means of an iterative algorithm and a likelihood criterion, addressing both a batch and an incremental (i.e. "online") scenario.

Formally, their problem is to find a piecewise segmented model composed by different functions that fit the data in different pieces (whose endpoints are changepoints): no assumption is made on the nature of these functions, but basis classes of functions are considered accordingly to the universal approximation theorem[5].

As the number of changepoints is not given, the algorithm proceeds iteratively with a stopping criterion given by the stability of the likelihood function (see Fig.2.9 for some results on real world cases).
Eventually, they ask four human experts for a visual changepoint detection and they show that their results are more robust, as human observers often disagree (see Fig.2.10 and have predilection for piecewise straight lines.

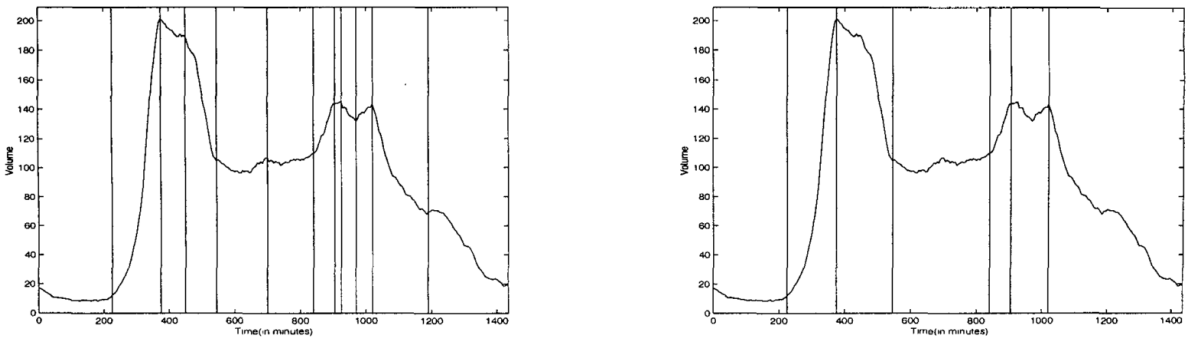


Figure 2.10: Comparison of two human responses on the same figure regarding Minneapolis traffic sensors.

2.4 Temporal pattern analysis

Temporal mining is a relatively new field of research which "mines" data that involve a temporal aspect; it does not exist yet a widely accepted taxonomy and thus we will refer to several different authors notations.

[Moerchen, 2006] proposes a nice survey on temporal pattern mining: sequential pattern mining is also included in the temporal mining context (in the market basket scenario, frequent itemset subsequences of items are searched across different transactions of customers as to give buying recommendations) as well as many other application fields like

[5]https://en.wikipedia.org/wiki/Universal_approximation_theorem

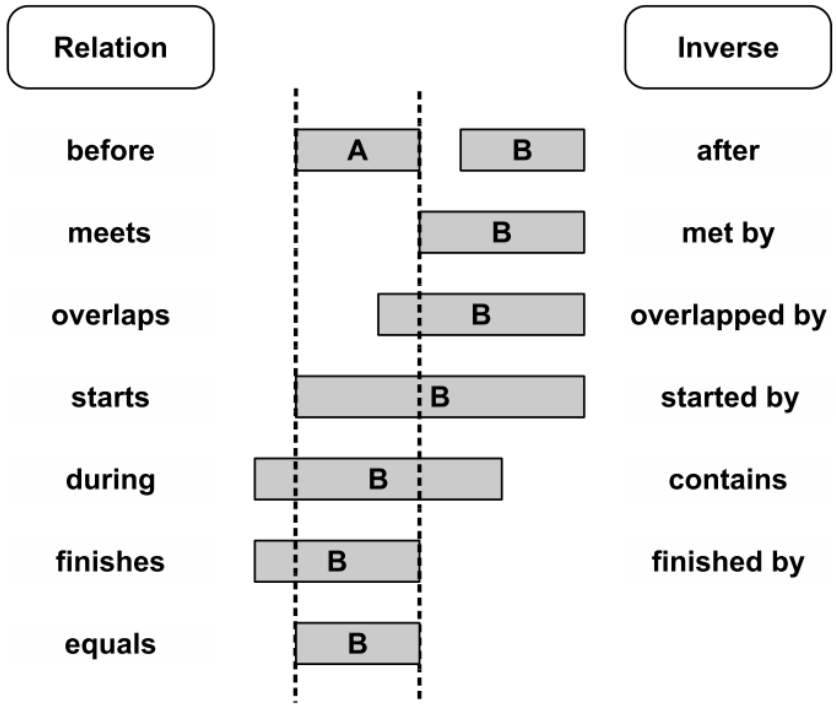


Figure 2.11: Examples of Allen's 13 interval relations.

temporal association rules, data streams, web usage mining, pattern evolution, temporal reasoning and databases which, however, do not relate to our task.

After presenting time series data mining techniques, the author moves to unsupervised mining of temporal patterns in time series, which is truly connected to our open problem as we'll see in 4.1.

Leaving aside time point rules (i.e. finding frequent sub series of symbols occurring sequentially in an univariate time series) we focus instead on time interval rules. Over the existing techniques, mostly of which can be applied on multivariate data, we can identify a precise set of different paradigms:

- Allen's relations (1983) (see Fig.2.11)
- Freksa's semi-interval operators (1992), a set of 11 operators which are combined to derive six operators whose advantage is to be represented by simpler formulas;
- Roddick's Midpoint Interval Operators, a set of 49 operators derived from Allen;
- Unification-based Temporal Grammar (UTG), which is an approximated version of Allen's relations,
- Temporal logic operators.

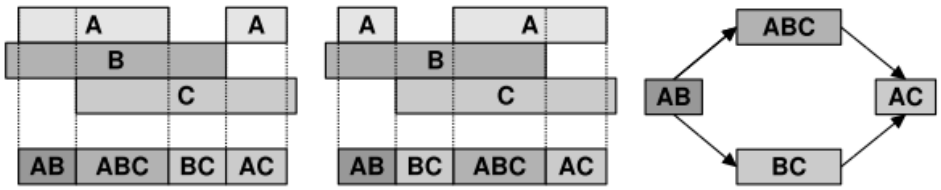


Figure 2.12: Some set of Chords (which include Tones) and a Phrase in TSKR.

Stating that all representations which are based on Allen's relations have severe disadvantages for knowledge discovery, the author proposes a hierarchical language for interval patterns known as Time Series Knowledge Representation (in the following we will refer to another work of the same author for sake of concision [Mörchen, 2007]).
The aim of such language is to extend UTG in order to achieve robustness and comprehensibility of patterns, as well as avoiding ambiguity.
A pre-processing stage is performed to transform time series into a symbolic representation (a custom one proposed by the author in [Moerchen, 2006]) to produce symbolic interval series.
The basic primitives (see Fig.2.12) are:

- Tones: they consist of a label, a symbol and an interval series
- Chords: time interval where $k > 0$ tones coincide
- Phrases: a partial order of $k > 1$ chords

The overall procedure is to find margin-closed (We refer the reader to [Moerchen, 2006][Mörchen, 2007] for an exhaustive definition) Chords and then margin-closed Phrases. You can check Fig.2.13 for an idea of the process.

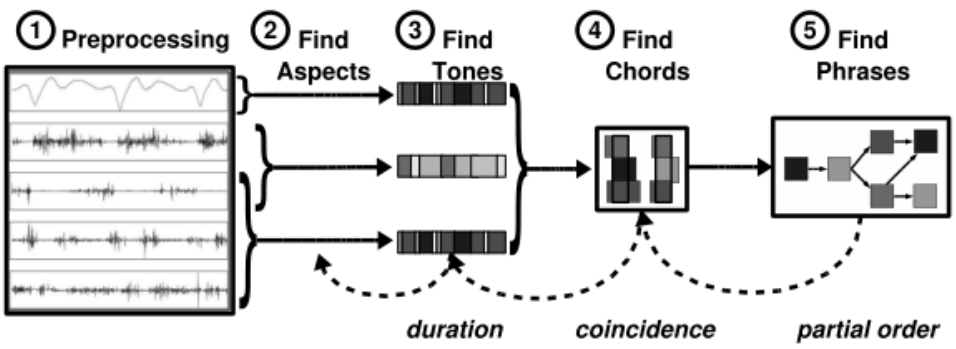


Figure 2.13: The five steps for TSKR pattern discovery.

Figure 2.14: A set of intervals and a temporal pattern consisting of their temporal relations (*co-occurs* and *before*).

Authors in [Batal et al., 2012, Batal et al., 2016] also propose a pattern mining approach to learn event detection models from multivariate data: they convert time series into time interval sequences of temporal abstractions and look backwards in time to find recent time interval patterns (they work in a supervised context).
Basically, given that certain events are associated to specific time points in a interval, they assume that relevant patterns to predict these events in the future have to be detected into a local context and giving more relevance to more recent instances (what they name a Recent Temporal Pattern).
They convert multivariate data by means of two temporal abstractions:

- Trend, they segment a time series like we aim to do (with local trends, i.e. `up`,`down`,`flat`) with a sliding window.
- Value, they segments values by using percentiles (and assigning labels such as "Very Low", "Medium", "High", etc).

They employ Hoppner's paradigm[6], which consists of two simple relations: *co-occurs* and *before* (rather than Allen's 13 relations) and they accordingly give a matrix representation for a pattern as in Fig. 2.14.
An interesting remark: also "instantaneous" events, where `start_time`=`end_time`, are correctly taken into account.
The core of their approach (see Fig. 2.15) is to build backwards these pattern by using several parameters to "tune" the gap between different components of a pattern which close to the end (i.e. the event occurrence) and have a limited duration (again they try to focus on more recent observations).
They build incrementally patterns in an efficient way: they only consider compatible

[6]As in the work we mentioned before, they claim that Allen's relations may lead to ambiguity and pattern fragmentation, i.e. several different patterns which are really similar.

relations when trying to find the $k+1$ pattern from a k pattern (where k is the number of components) and they thus prune the search space.
Eventually they focus on predictive patterns by means of Bayesian inference: these are, in fact, the most potentially useful amongst all the frequent RTP to learn event prediction rules.

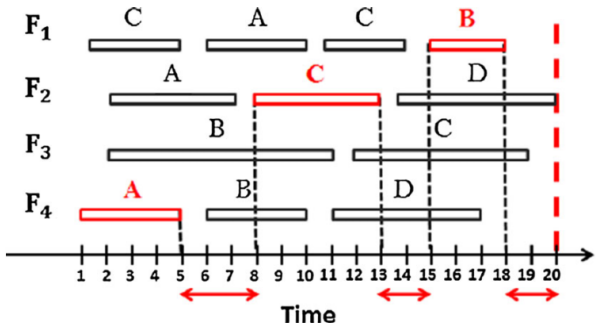


Figure 2.15: A highlight of a Recent Temporal Pattern (the triggered event occurs at t = 20).

Authors in [Moskovitch and Shahar, 2015a, Moskovitch and Shahar, 2015b] concentrate instead on time series classification: they apply temporal abstractions to raw multivariate time series, they mine frequent Time Interval Related Patterns and they exploit these patterns to build a classifier.
They compare SAX (see Section 2.2) and Equal Width Discretization, the former performing better, and they employ Allen's relations to build patterns: eventually they derive three supersets (*before*, *contains*, *overlap*) and extend them with an ϵ version which allows for flexible pattern mining (see Fig. 2.16).
Their algorithm KarmaLego firsts finds 2-sized TIRPs and then extend them recursively to build a tree (similarly to [Batal et al., 2016] they only consider valid relations when trying to extend patterns).
Eventually, as duration is not taken into account when building patterns, they propose to cluster either time intervals or TIRPs for a better knowledge discovery (with significant effects on the computations).

[Chen et al., 2015] proposes two interesting different representations ("endpoint" and "endtime" as in Fig.2.17) to deal with the intrinsic complexity of describing relations

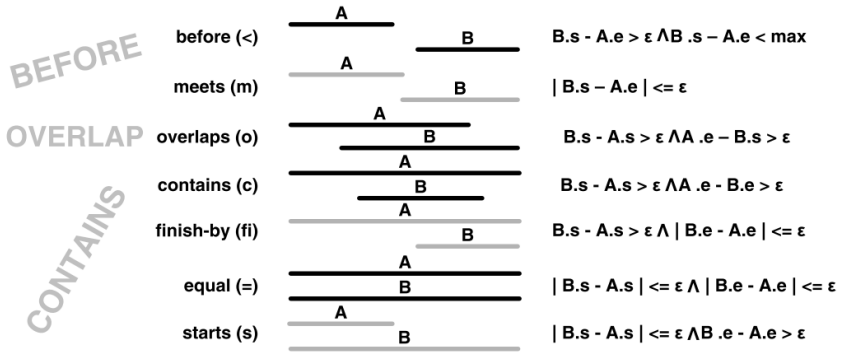


Figure 2.16: Allen's seven relations with epsilon flexible extension and the three supersets.

between intervals. They basically reduce Allen's relations into three relationships: before, equal and after.

They also propose three different kinds of interval based patterns:

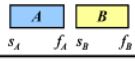
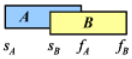
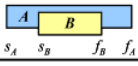
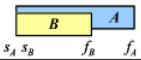
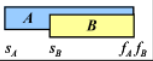
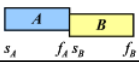
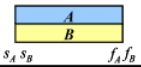
temporal relation	inversed relation	pictorial example (*s*: starting time, *f*: finishing time)	endpoint representation	endtime representation
A before B	*B after A*		$A^+ A^- B^+ B^-$	$\begin{pmatrix} A^+ & A^- & B^+ & B^- \\ s_A & f_A & s_B & f_B \end{pmatrix}$
A overlaps B	*B overlapped-by A*		$A^+ B^+ A^- B^-$	$\begin{pmatrix} A^+ & B^+ & A^- & B^- \\ s_A & s_B & f_A & f_B \end{pmatrix}$
A contains B	*B during A*		$A^+ B^+ B^- A^-$	$\begin{pmatrix} A^+ & B^+ & A^- & B^- \\ s_A & s_B & f_A & f_B \end{pmatrix}$
A starts B	*B started-by A*		$(A^+ B^+) B^- A^-$	$\begin{pmatrix} (A^+ & B^+) & B^- & A^- \\ s_A & s_B & f_B & f_A \end{pmatrix}$
A finished-by B	*B finishes A*		$A^+ B^+ (A^- B^-)$	$\begin{pmatrix} A^+ & B^+ & (A^- & B^-) \\ s_A & s_B & f_A & f_B \end{pmatrix}$
A meets B	*B met-by A*		$A^+ (A^- B^+) B^-$	$\begin{pmatrix} A^+ & (A^- & B^+) & B^- \\ s_A & f_A & s_B & f_B \end{pmatrix}$
A equal B	*B equal A*		$(A^+ B^+) (A^- B^-)$	$\begin{pmatrix} (A^+ & B^+) & (A^- & B^-) \\ s_A & s_B & f_A & f_B \end{pmatrix}$

Figure 2.17: Proposed equivalent representations for Allen's relations.

1. temporal, which can reveal the correlation among intervals;
2. occurrence-probabilistic, which provide with insight on the occurrence probability, enabling prediction;
3. duration-probabilistic, which expresses the correlation between intervals and the distribution of the different interval lengths.

They assume a temporal database of transactions (as in Fig.2.18, which may be adapted to our specific case.

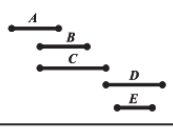
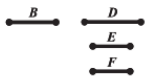
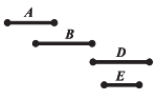
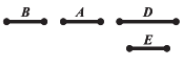
SID	event symbol	start time	finish time	event sequence	$\binom{\text{endpoint sequence}}{\text{endtime sequence}}$
1	A	2	7		$\binom{A^+\ (B^+\ C^+)\ A^-\ B^-\ C^-\ D^+\ E^+\ E^-\ D^-}{2\ 5\ 5\ 7\ 10\ 12\ 16\ 18\ 20\ 22}$
1	B	5	10		
1	C	5	12		
1	D	16	22		
1	E	18	20		
2	B	1	5		$\binom{B^+\ B^-\ D^+\ (E^+\ F^+)\ (E^-\ F^-)\ D^-}{0\ 5\ 8\ 10\ 10\ 13\ 13\ 14}$
2	D	8	14		
2	E	10	13		
2	F	10	13		
3	A	6	12		$\binom{A^+\ B^+\ A^-\ (B^-\ D^+)\ E^+\ E^-\ D^-}{6\ 7\ 12\ 14\ 14\ 17\ 19\ 20}$
3	B	7	14		
3	D	14	20		
3	E	17	19		
4	B	8	16		$\binom{B^+\ B^-\ A^+\ A^-\ D^+\ E^+\ E^-\ D^-}{8\ 10\ 13\ 16\ 20\ 21\ 22\ 23}$
4	A	18	21		
4	D	24	28		
4	E	25	27		

Figure 2.18: A temporal database and the proposed representation.

Eventually, [Rawassizadeh et al., 2016] aims to develop a set of scalable algorithms to find interesting patterns in human behaviors by looking at consecutive daily routines (see Fig.2.19).

They define groups and profiles to detect these patterns, and they apply "sliding" (rather a subdivision of a week in groups of consecutive two days) windows in order to find frequent sets of similar activities; an interesting remark is how they convert point-wise occurrences to intervals by means of rounding it to a reference granularity.

Although they use Allen's relations they clearly state that their approach should not be categorized as a time series one (yet they process multivariate data).

Moreover, their approach has a scope limitation: authors claim to be able to identify daily consecutive behaviors but not all routine behaviors (e.g. going to the cinema every two or three months or going to a campaign once a year) which would be some interesting abstractions that we aim to find in an event stream.

2.5 Learning automatically rules for CEP

In the recent literature we found a few different approaches, not truly remarkable, that tried to first tackle the general issue of an automatic formulation of complex rules for event processing, spanning from machine learning to time series data mining techniques.

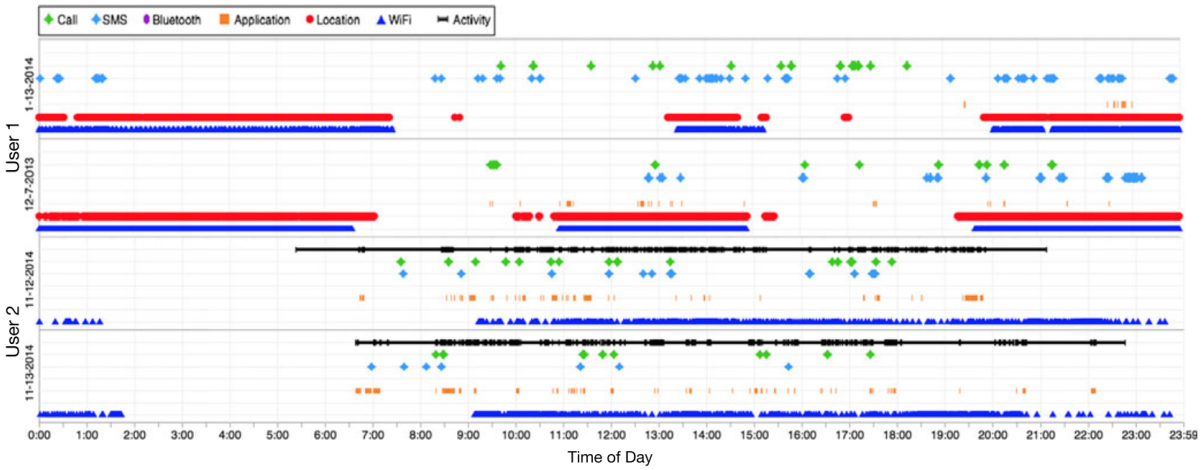


Figure 2.19: Two days visualization for the data of two different users in [Rawassizadeh et al., 2016].

2.5.1 iCEP, autoCEP and other machine learning techniques

A couple of works fit to some extent our problem as they make similar assumptions on the complexity of building these complex patterns and they try to build them using Event Processing Language semantics.

[Margara et al., 2014b] presents a modular solution to learns causality between events starting from historical traces, which they distinguish into positive and negative ones matching a pattern to a dataset.
They exploit to infer complex rules decomposing the problem into several ones: determine the time window, identify types and attributes and eventually discover selection, parameter, aggregate and negation constraints.
Everything is performed by applying these operations on traces (i.e. inferring all the existing correlations in between the events), which constitute thus a training set; in the end, an evaluation is carried out on a public transportation dataset assuming only one existing rule and the goal of "reconstructing" it in the most efficient way.
Although the interesting submodular approach and the premises are very promising, the goal of iCEP of "reconstructing" one single rule departs a bit from our context; nonetheless the idea of decomposing the problem of finding the different components of a pattern is a good hint.

[Mousheimish et al., 2016] tries instead to integrates time series data mining techniques

(in particular Early Classification) to deliver an automatic rule extraction; precisely, they employ "shapelets" (a new primitive for time series data mining) to "annotate" multidimensional time series and find correlations across several attributes to "translate" into CEP rules (using Esper[7] framework, an industrial CEP tool).

The following works only focus on the Human Activity Recognition domain, where events are three-axis measures (forward, upward, downward and horizontal movement of the legs) and patterns are situations to detect (e.g. walking or running):

- [Petersen et al., 2016] exploits SVM along with TESLA query model. The technique proposed is an online version of SVM algorithm that can learn and generate rules on the fly, as these can change dynamically in time; overall, only 6 amongst the 43 features of the dataset are considered (average, deviation and others) and neither these variables neither the functioning of the rule extraction are really clear.
- [Mehdiyev et al., 2015] proposes a rule-based model insisting on the "updating" need of patterns and their readability; a particular attention is also paid on outliers and missing data, although it is not so related to our case, that are handled with an R package. In the end they exploit 18 input variables (the three-axis measures along with 5 lagged values for each of them) and simply compare different rule-based classifiers performances leading to different rankings according to different measures.
- [Mehdiyev et al., 2016], conceived as a "future" direction of the previous work, proposes a fuzzy algorithm combined with a feature subset selection (where "fuzziness" is justified by outliers and uncertain data). The evolutionary approach employed is the most competitive wrapper one, i.e. it considers interdependencies among the variables, for feature subset selection; roughly speaking it is the solution of a multiple objective optimization problem which is not clearly explained. The fuzzy algorithm (FURIA) is a rule induction one (essentially a modification of RIPPER, which is also present in the previous work) whose main disadvantage is the complexity in interpretation. Features are eventually reduced to 11 from 18, and a similar evaluation is performed (no clear patterns, just classification) as to substantiate a

[7] http://www.espertech.com/esper/

possible "primate" of fuzzy rules in the CEP world.

In the end, the works aforementioned provide very little contributes to the cause and they lack of many technical details.

2.5.2 IL-Miner

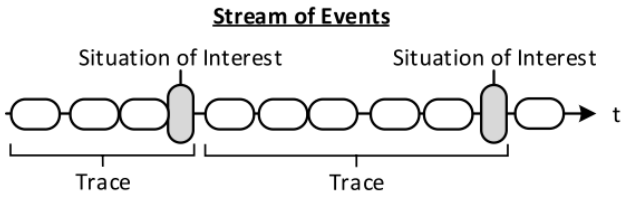


Figure 2.20: Partitioning an event stream (or a prefix thereof) in traces.

As discussed afterwards in 3.1 correlating events to define complex patterns has notably an exponential blow-up in terms of time windows, primitive operators and value predicates; authors of [George et al., 2016b] claim that using labeled historic event data (aka *traces*), instead, allows to learn these patterns automatically (see Fig.2.20.

Traditionally, CEP applications need domain experts to define situations of interest, with a very recent trend of shifting towards pro-active processing where patterns should anticipate these scenarios.

Using labeled data, one can split event streams in sequences knows as traces which can be used for a learning process (linking this task to the field of frequent sequence mining), where existing techniques discover sequential patterns on type-level, i.e. they partition events into types and identify order over these types, or partitioning on attribute values for global correlation conditions; nonetheless, an automatic approach would need to explore a very large space of possible patterns, thus becoming intractable in practice.

The work adopts a relational model of event streams (like SASE) and assumes that all events have the same schema; moreover it defines formally an event pattern as a triplet of event variable, a set of constraints and a time window.

A trace is then defined as a finite sequence of events such that the situation of interest occurs at the end of the timespan covered by the trace; these can be obtained using a maximal duration or partitioning the stream by situation of interest.

IL-MINER learns event abstractions and correlation conditions automatically by exploiting these traces:

1. it employs FSM (frequent sequence mining) to find candidate sequences of event abstractions;
2. it links them to events in the trace to find relevant event abstractions and correlation conditions;
3. it filters interesting patterns among a large result set.

This procedure is eventually shown to be formally correct, complete and minimal.
In the end, evaluation is performed on four different datasets and several existing patterns are used to build up traces. Comparing it to [Margara et al., 2014b], the latter is outperformed and fails to discover patterns for many scenarios.

Despite being the most promising methodology across the works analyzed so far, the concept of trace does not properly fit completely our task (our main premise is a completely unsupervised setting) although the clustering procedure to filter out patterns could be interesting (nonetheless there is some kind of information which resides in the syntax of the pattern itself, that needs to be evaluated by a human expert).
Moreover, the assumption of all the events with a same schema is not so practical.
Although they mention classical CEP operators, they do not produce any readable rules but they focus on the same problem as [Margara et al., 2014b] of "reconstructing" patterns from their traces.

Chapter 3

An in-feasibility proof sketch for a search-based approach

3.1 Problem definition

As discussed so far, Complex Event Processing is a powerful technique for data analytics which is, nonetheless, guided by manually defined rules. As such, implementing a CEP application is not straightforward as domain knowledge must be exploited in order to define rules which may be useful on a certain dataset.
As we will indicate, there is an unnumberable amount of different correlations that can be expressed using an Event Processing Language (which makes it really powerful to process streaming datasets) that makes any data-driven brute force approach infeasible for composing such patterns.

In the following sections we will first provide the sketch of a proof of in-feasibility for brute-force searching all the existing rules that would match on a given dataset: a simplistic analysis shows that even considering only sequences of events (without any `JOIN` conditions or aggregate functions such as `SUM`, `AVG`, etc.) holds an exponential blow-up in terms of results.

The simplified problem we will (implicitly) refer to in our analysis is the following: given a stream of events S of fixed size, find all possible matching queries (using SASE lan-

guage).
Eventually, some additional processing on such results could give the most interesting rules.
The basic form of a SASE query that we employ is expressed in the form:

```
FROM S
PATTERN p
WHERE skip_till_any_match(){}
WITHIN W
```

where S is our stream, p is a pattern defined using SASE operators (such as `ANY`, `+`, `SEQ`) and W is the time window which needs to be specified.
For the moment we do not specify any `WHERE` predicates but only the strategy employed. We refer the reader to [Zhang et al., 2014] for a deeper understanding of how the language works.

A first remark is that all matches of a pattern containing `ANY` and `+` operators in SASE (1.2.2) can be obtained also by using patterns that contain only the SEQ operator, i.e. the result set of the former is the union of the results obtained with several of the latter. Moreover, considering such strategy allows us to analyze the worst-case scenario.

We can then formulate the problem in two equivalent forms, which both have an exponential blow-up in terms of results: find distinct subsequences of a given sequence and find all the distinct matching queries over an input stream.

3.2 Find distinct subsequences

Given a finite input sequence S defined as follows:

$$S = \langle e_i \rangle \quad i \in N, i < \infty$$

where $e_i \in \Sigma \equiv \{\text{Types of event}\}$.
We define the set of distinct subsequences P as:

$$P = \{p \in P \mid p \equiv \langle e_j \rangle, j < \infty, j \in N, \forall p_1, p_2 \in P : p_1 \neq p_2\}$$

where

$$p_1 \neq p_2 \equiv \exists k : p_{1,k} \neq p_{2,k}$$

Here we use the notation $p_{1,k} \equiv (k, p_{1,k})$ to denote the "k-th" element of p_1.
The problem is then to find P (or $|P|$).

A well-known result for a sequence of k distinct symbols is that the total number of non-empty distinct subsequences is $2^k - 1$.
The result is bounded by $2^{|S|} - 1$ subsequences if symbols are repeated: in fact, we see that in the general result we could have sequences included in the former result (duplicated sequences) and sequences not included (e.g. in "ABA" we have "BA" which does not belong to the power set of "AB"[1]).
Thus, we can argue that:

$$2^k - 1 \leq |P| \leq 2^{|S|} - 1$$

3.3 Find distinct matching queries

Given a finite input S defined as follows:

$$S = \{e_i \in S \mid e_i(t) = \left\{ \begin{array}{ll} TRUE & \text{if } t = t_i \\ FALSE & \text{otherwise} \end{array} \right\}, \forall i < j : t_i < t_j\}$$

where $0 < t < \infty$ and $e_i \in \Sigma \equiv \{\text{Types of event}\}$, where e_i denotes just the symbol.
We define P as the set of distinct matching patterns (or queries in the SASE framework) on S:

$$P = \{p \in P \mid p \subseteq S, \forall p_1, p_2 \in P : p_1 \neq p_2, \forall p(i) : t_i < t_{i+1}, \exists t_1 < t_2 < ... < t_n : \bigwedge_1^n e_i(t_i) \in p = TRUE\}$$

[1] "A","B","AB".

where

$$p \subseteq S \equiv \forall e_i \in p, e_i \in S$$

and

$$p_1 \neq p_2 \equiv \exists k : p_1(k) \neq p_2(k)$$

and the last two conditions mean that the symbols are ordered according to the "triggering time" (i.e. when they evaluate to TRUE) and the pattern matches the input.
We further use the notation $p(i)$ to identify the "i-th" symbol in the pattern and t_i its "triggering time".
The problem is then to find P (or $|P|$) and we easily see that it is equivalent to the previous problem (i.e. we can always build a sequence of symbols ordered by their "triggering time" and obtain such a formulation).
Thus:

$$2^k - 1 \leq |P| \leq 2^{|S|} - 1$$

3.4 Adding predicates

So far we have considered only simple patterns in the form:

```
FROM S
PATTERN p
WHERE skip_till_any_match(){}
WITHIN W
```

What if we add predicates in the `WHERE` clause?
We can say that we have up to C boolean conditions which are `TRUE` for the input S: this parameter would depend on $|A_i| \equiv$ {No. attributes for event type "i"} and $|V_a| \equiv$ {No. values for attribute "a"}; we could have different types of conditions (e.g. $c_{eq}, c_{attr}, c_{aggr}$) that would add differently to C.
Given m conditions we have thus $(2^m - 1)$ non empty subsets of conditions, each of which evaluates to `TRUE` when using only `AND` connectors between two literals.
Therefore we have $(2^C - 1)$ possible `WHERE` clause that match the input, holding for a W

size input a worst-case result:

$$|P| \leq (2^W - 1)(2^C - 1)$$

as the previous "truth" assumption on conditions may not hold for each pattern (e.g. c_1 might be `TRUE` only for pattern p_3 and so on).

3.5 NEG operator

What if we also consider the NEG operator?
Our previous results tell us that:

$$|P| = 2^k - 1 \quad \text{with } |S| = k \text{ distinct symbols}$$

$$|P| = 2^k - 1 + N \quad \text{with } |S| \neq k \text{ symbols, } k \text{ unique}$$

Where N is the number of distinct subsequences which are not included in the first term.
We can reformulate the last line in:

$$|P| = 2^k - 1 + \sum_{1}^{|S|} N_i$$

where N_i is the number of distinct subsequences of length i that are not included in the first term.
We also observe that, given a sequence and a matching pattern, we can put the `NOT` operator in any position of the pattern, using a symbol which is not present in the overall sequence, such that the pattern still matches the input:

ABCBDB $\rightarrow$ SEQ(!E, A, B), SEQ(A, !E, B), SEQ(A, B, !E)

for a total number of $(i+1)$ possibilities, where i is the length of the pattern.
In the end, for k' non-present symbols[2] we obtain:

$$|P| = \sum_{1}^{k} \binom{k}{i} k' \cdot (i+1) + \sum_{1}^{|S|} k' \cdot (i+1) N_i = k' \cdot (2^k - 1) + k' \cdot 2^{(k-1)} k + \sum_{1}^{|S|} k' \cdot (i+1) N_i$$

Though, we did not consider any `NEG` operator using symbols present in the input (e.g. "ABCBD" is matched by `SEQ`(A, C, !D, B)).

[2] $k' = |\Sigma| - k$

Chapter 4

Towards automatic extraction of complex rules

4.1 Problem statement and proposed solution

The main challenge to face in order to learn CEP rules automatically is the intrinsic complexity that derives from the very richness of Event Processing Languages, which are capable of expressing innumerable different correlations between primitive events.
As we pointed out, applying a brute-force search-based approach to generate all possible interesting rules that apply on a given dataset is not feasible.
Instead, we propose a brand new approach to tackle the problem of automatic extraction of CEP rules from streams of events, which consists of the following procedure:

1. Transform a finite stream of events in a dataset of multivariate time series;
2. identify potential situations where to build basic CEP operators: i.e. temporal windows where time series show an interesting behavior (whether isolated or in conjunction);
3. build a set of matching patterns according to these windows;
4. identify the most pertinent ones according to some notion of *interestingness*;
5. evaluate the overall procedure by means of ground truth or human experts.

Notice that the pipeline operates in a "batch" mode, i.e. a finite stream of events is analyzed off-line; however, we developed our approach keeping in mind that a streaming fashion should be achieved for performing real-time CEP rules learning.
We give here one formulation for our specific problem:

> Given a finite stream of events S and finite alphabet of event types T, build a multivariate time series for each event type T_i; then represent each time series by means of a proper abstraction and apply temporal mining techniques as to extract interesting patterns and build rules according to a given Event Processing Language.

The pipeline of such "plan of attack" is depicted in Fig.4.1.
In the following subsections we first describe the real world case dataset we used for our analysis and therefore we go into the details of each block of our architecture.

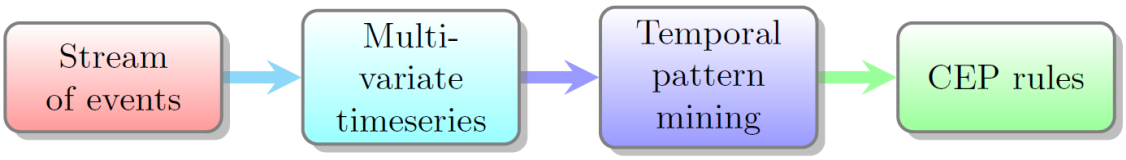


Figure 4.1: A diagram that describes our overall system.

4.2 Real world dataset: monitoring Hadoop activities

In this section we provide with a bit of context regarding our dataset from a real world case study on Hadoop.

The data we used for our analysis refer to the monitoring of Hadoop activities carried on in the period 28th March-12th April 2015; these were produced using logs of OS metrics and Hadoop itself, by means of Ganglia monitoring daemon `gmond`[1].

[1] http://ganglia.sourceforge.net/.

These logs correspond to three different workloads, as in [Zhang et al., 2017]: count statistics for tri-grams for tweets, compute frequent users from click logs in the 1998 FIFA Worldcup website and divide user clicks into session. Anomalies such "High Memory Consumption" or "High IO activity" were manually "injected" in order to experiment some supervised anomaly explanation.

The Hadoop architecture used in such applications is depicted in Fig.4.2: it corresponds to the (older) version 1.x. of MapReduce framework and it is slightly different from the actual[2] version 2.x. which is shown in Fig.4.3.

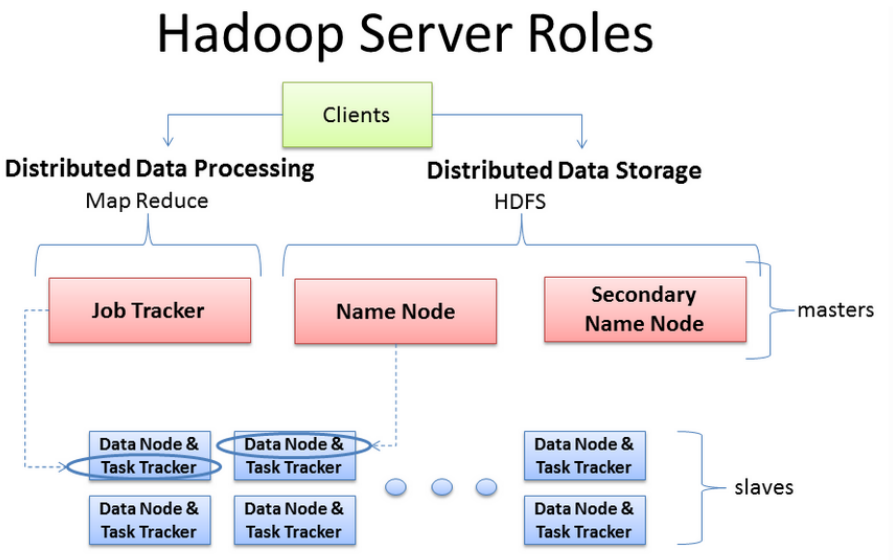


Figure 4.2: Hadoop 1.x architecture ovierview.

As explained in [White, 2009], the Name Node coordinates the data storage function (HDFS), while the Job Tracker coordinates the parallel processing of data using Map Reduce.

Slave Nodes make up the vast majority of machines and do all the "dirty" work of storing the data and running the computations.

Each slave runs both a Data Node and Task Tracker daemon that communicate with and receive instructions from their master nodes. The Task Tracker daemon is a slave to the

[2] https://www.cloudera.com/documentation/enterprise/5-3-x/topics/cdh_ig_mapreduce_to_yarn_migrate.html.

Job Tracker, the Data Node daemon a slave to the Name Node.

Client machines have Hadoop installed with all the cluster settings, but are neither a Master or a Slave. Instead, the role of the Client machine is to load data into the cluster, submit Map Reduce jobs describing how that data should be processed, and then retrieve or view the results of the job when its finished.

In smaller clusters (∼40 nodes) you may have a single physical server playing multiple roles, such as both Job Tracker and Name Node. With medium to large clusters you will often have each role operating on a single server machine.

The Secondary Name Node occasionally connects to the Name Node (by default, ever hour) and grabs a copy of the Name Node's in-memory metadata and files used to store metadata (both of which may be out of sync). The Secondary Name Node combines this information in a fresh set of files and delivers them back to the Name Node, while keeping a copy for itself.

Should the Name Node die, the files retained by the Secondary Name Node can be used to recover the Name Node.

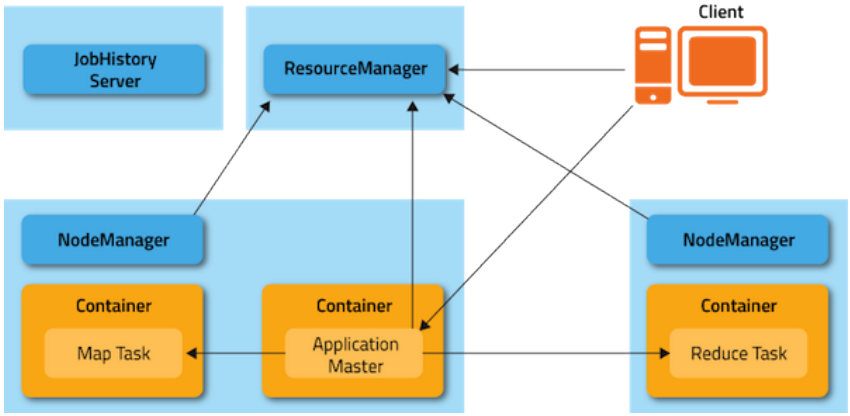


Figure 4.3: Hadoop 2.x architecture ovierview.

Apache Hadoop's jobtracker, namenode, secondary namenode, datanode, and tasktracker all generate logs [White, 2009].

In our specific case, input data are actually the results of a parsing operation (carried out by `gmond`) on Job Statistics logs: these are created by the jobtracker that writes runtime statistics from jobs to these files. Those statistics include task attempts, time spent shuffling, input splits given to task attempts, start times of tasks attempts and other information.

We have the following set of events for what concerns Hadoop statistics (some of which are "self-explaining"):

1. HadoopDataActivity (cf. DataIO in [Zhang et al., 2017]) records the activities of generation (positive values) / consumption (negative values) of intermediate data; it has the `nodeNumber` of the JobTracker node.
2. JobStart and JobFinish, which have the `nodeNumber` of the JobTracker node.
3. MapStart, MapFinish, ReduceStart and ReduceFinish, that have the `nodeNumer` of the node they run on.
4. MapPeriod and ReducePeriod, show the duration of the tasks (with same `nodeNumber` as the tasks themselves).
5. MergeStart, MergeFinish correspond to the final output of all reducers and they have the `nodeNumber` of the JobTracker, as they occur at the end of the job.
6. PullStart, PullPeriod and PullFinish correspond to the input splitting before map start.
7. RequestStart and RequestFinish correspond to the request of input by the mappers.

And for what concerns OS metrics an exhaustive list is found in the Appendix[3]. We will just say that they correspond to specific performance metrics for each machine: they thus have a `nodeNumber` that represents the corresponding Node of the cluster and a `value` that represents the value of the corresponding metrics.

We eventually derived the following hierarchy, where each single line is a (left-to-right) one-to-many relationship with the only exception of the the many-to-many relationship between `nodeNumber` and `jobId`):

$$\underset{nodeNumber}{\bigcirc} = \underset{jobId}{\bigcirc} - \underset{taskId}{\bigcirc} - \underset{attemptId}{\bigcirc}$$

In fact:

[3]You can also check https://github.com/ganglia/monitor-core/blob/master/ganglia.pod

- a job can be run on several machines and one machine can run several different jobs (`jobId` are 1-based, i.e. the first job has `jobId`=1)[4]
- for each job we have different tasks (0-based, i.e. the first task has `taskId`=0)
- for each task we have different attempts (0 based as well)

After an exploratory analysis of the data we manually selected the following *dimensions*:

- `jobId`
- `nodeNumber`

with the only consistent *measure* being the `value` attribute. In this way we can extract the partitioning tuples and build partitions accordingly.
In order to handle some (event) types that do not have the `jobId` attribute (OS metrics from Ganglia) we include them in the partitions if they just match the `nodeNumber` of the partition.
An example of two events from the same partition is shown in Fig.4.4.

4.3 From event streams to multivariate time series

As said in 1.4, a multivariate time series is a collection of univariate time series which refer to same underlying process and, thus, to the same time span. You can build such model whenever you have a sequence of multi-dimensional points (or variables): each dimension would constitute one univariate time series and the overall result would be a d-multivariate time series (where d is the number of dimensions). An example with 3 dimensions is shown in Fig.4.5

The main intuition behind our work is that given an event stream that follows the event model (see 1.2), one can always build a dataset of multivariate time series, one for each event type, by using the sequences of values of each attribute. The next goal is then to compare different multivariate time series and extract potential correlations that can give

[4]Actually the jobId is a concatenation of the date and a counter which starts from 1, e.g. 2015031422290008

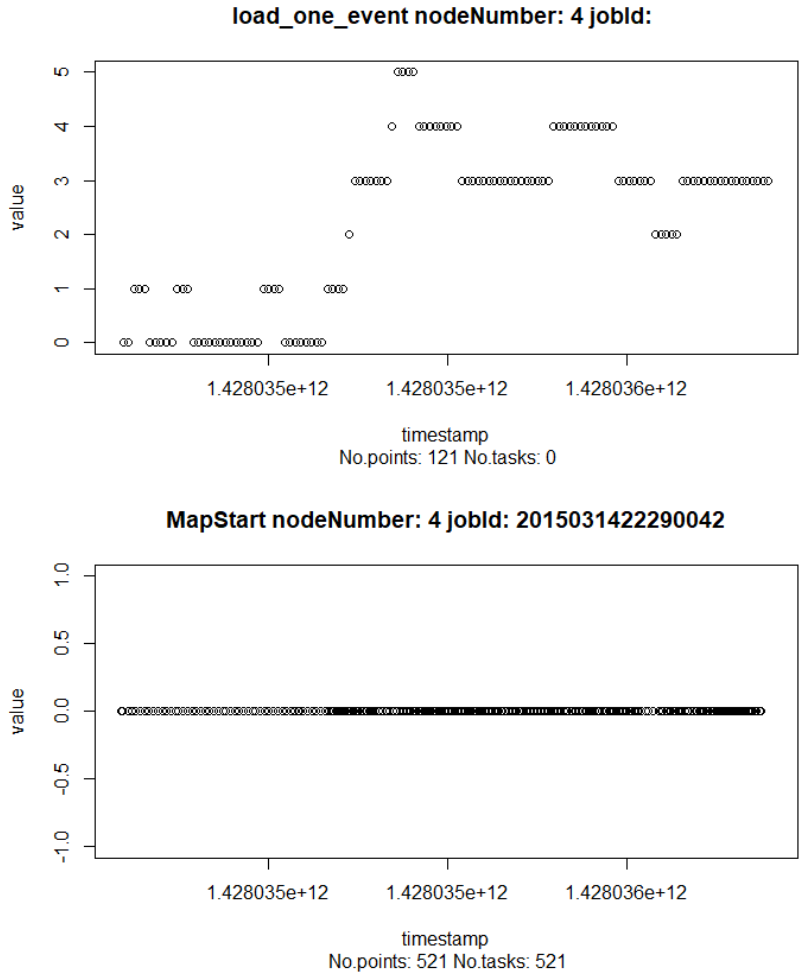


Figure 4.4: Two time series corresponding to two different event types (respectively from Hadoop logs and OS metrics logs) in the same partition (nodeNumber=4, jobId=2015031422290042).

insights and, more importantly, build CEP rules.

It is often the case where several attributes of an event type are used only for "joining" different event types (say `jobId` or `transactionId`) whereas other attributes carry values that provide "more continuous" information (such as `price`, `volume`, etc).
Thus, our realistic assumption is that these streaming data are generated by architectures that can be modeled using a Dimensional Fact Model[5], such as logistics or healthcare data warehouses, naturally providing the means to partition data along some dimensions and building time series according to some measures.

[5] https://en.wikipedia.org/wiki/Dimensional_fact_model

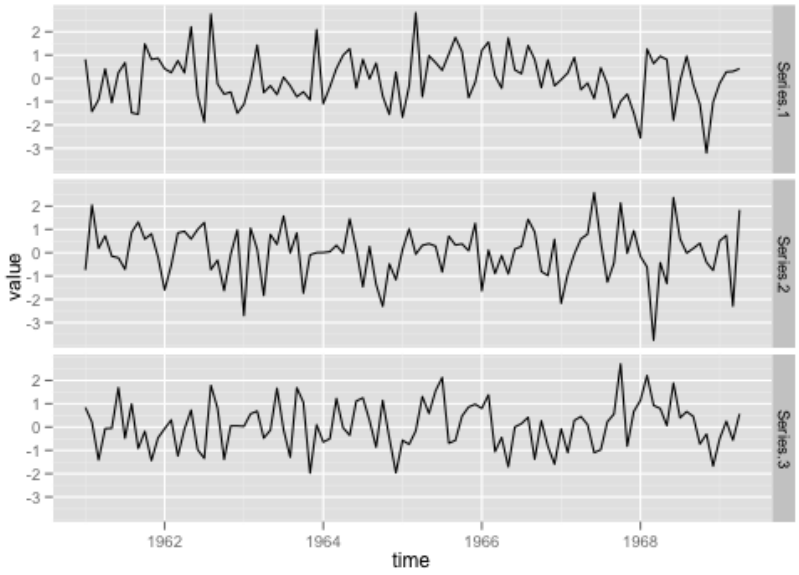


Figure 4.5: A plot of a 3-multivariate time series

Our initial step, therefore, consists of partitioning the stream by `type` first and then according to a set of dimensions. We assume the user provides both a list of event types and the attribute taxonomy.

We provide here some formal definitions to understand the scenario.

Definition 1. *Event stream.* A finite event stream $S = \langle e_i \rangle_{i \in \mathbb{N}_0}$ is a set of events e_i occurring in a non-decreasing order (w.r.t `timestamp`[6]), where each $e_i \models E^j$ i.e. e_i is of event type $E^j \in \Sigma, E^j \equiv \langle a_1, a_2, ..., a_n \rangle$; we can also say that $E^j \equiv \sigma_E^j(S)$, i.e. it represents a selection over the stream of all the events of type E^j (our first partitioning step).

Definition 2. *Attributes taxonomy.* Given an event type $E^j = \langle a_1, a_2, ..., a_n \rangle$ and a set of labels $L = \langle l_1, l_2, ..., l_n \rangle$, we build a set of *dimensions* D_E^j and *measures* M_E^j according to labels: for each attribute a_i we put it in one set by checking the corresponding label l_i, D_E^j if l_i is a *dimension*, M_E^j otherwise. We assume that the `timestamp` is a special attribute in the sense that it is not considered in such procedure and never dropped; it will be used eventually to build time series.

Definition 3. *Partition.* Given an event type E^j and a partition p_i identified uniquely identified by a list of values (i.e. instances of the dimension attributes) $p_i = (v_1, ..., v_n)$, a sub stream of E^j belonging to the partition is defined "recursively", i.e. a `SELECTION`

[6]We will consider it as a "special" attribute: nor dimension nor measure.

on each *dimension* followed by a `PROJECTION` over the other attributes, as:

$$p_i(E^j) = \pi_M(\sigma_{d_n==v_n}(\pi_{d_{n-1},M}(\sigma_{d_{n-1}==v_{n-1}}(\pi_{d_{n-2},d_{n-1},M}(\ ...\ \pi_{d_2,...,d_n,M}(\sigma_{d_1==v_1}(E^j))$$

or more compactly

$$p_i(E^j) = \pi_M(\sigma_{d_n==v_n}(\pi_{\neg d_{n-1}}(\sigma_{d_{n-1}==v_{n-1}}(\pi_{\neg d_{n-2}}(\ ...\ \pi_{\neg d_1}(\sigma_{d_1==v_1}(E^j))$$

where M is the set of *measures*. In the end what we obtain is a "table" composed of only *measures* and a `timestamp` column which will be used to build time series.

Definition 4. *Time series.* Given a partition $p_i(E^j)$ of an event type E^j, its corresponding multivariate time series $\mathrm{TS}_{p_i(E^j)}$ is $(\langle m_{1,j}\rangle, \langle m_{2,j}\rangle, ..., \langle m_{n,j}\rangle)$ where $\langle m_{1,j}\rangle$ is a sequence of points whose subscript indexes the corresponding `timestamp` (equivalently $(m_{1,j}; \mathtt{t}_j)$); we recall that each m_i is a *measure* and each sequence of points belongs to a different partition.

Given a finite event stream S, an alphabet of event types Σ and a set of labels L we can build for each event type E^j a set of multivariate time series (one for each partition p_i) $\mathrm{TS}_{p_i(E^j)}$ by applying the following procedure:

1. Read the event stream S;
2. partition the stream by event type according to Σ;
3. for each event type label attributes according to L;
4. build the set P of partitions by looking at the set of dimensions $D \subset L$;
5. for each sub stream (corresponding to a specific *continuous* event type) in each partition build a multivariate time series by using *measures* attributes and the `timestamp`.

A simplistic assumption we made is that all the event types carry a set of *dimensions* which allow to build consistent partitions.

We expect to have some event types that do not carry *continuous* information but still their occurrences can be used to detect temporal patterns. In this case, we assume that the user provides labels for each event type, as answering to the question "Should we

treat this sequence of points as a time series?" is not straightforward and one can not simply argue on the frequency or the variance of the values of a sequence of observations.

Using this rough taxonomy of *time series* and *pointwise* event types, one can then select those that should be further processed by means of temporal abstractions (in our case, time series segmentations techniques) and, as a ultimate stage, apply any pattern mining techniques (which work on both point-based and interval-based symbols).

4.4 Temporal abstractions for time series

As seen in Section 2.4, an interval-based representation is compulsory in order to apply temporal pattern mining techniques that take into account the concept of duration for a given item.
This means that representing time series with a consistent abstraction is a crucial pre-processing phase for our final goal.
In our analysis we considered three different sets of representations:

1. *trend-based* segmentation, where each segment is represented either by a monotonic trend or a flat behavior;
2. *level-based* segmentation, where each segment contains point belonging to a certain range of values (e.g. quantiles);
3. **S**ymbolic **A**ggregate appro**X**imation (SAX), which transforms a time series into a set of symbols by applying Z-normalization and Piecewise Aggregate Approximation.

We will describe the first two in the following sections: we actually implemented our own algorithms as no existing work focused on this kind of approach (we enhanced the intuitions of [Batal et al., 2016] where these representations are used in an approximate way).
All the pseudocode is available in the Appendix.
For what concerns SAX representation, we refer the reader again to Sec.2.2.

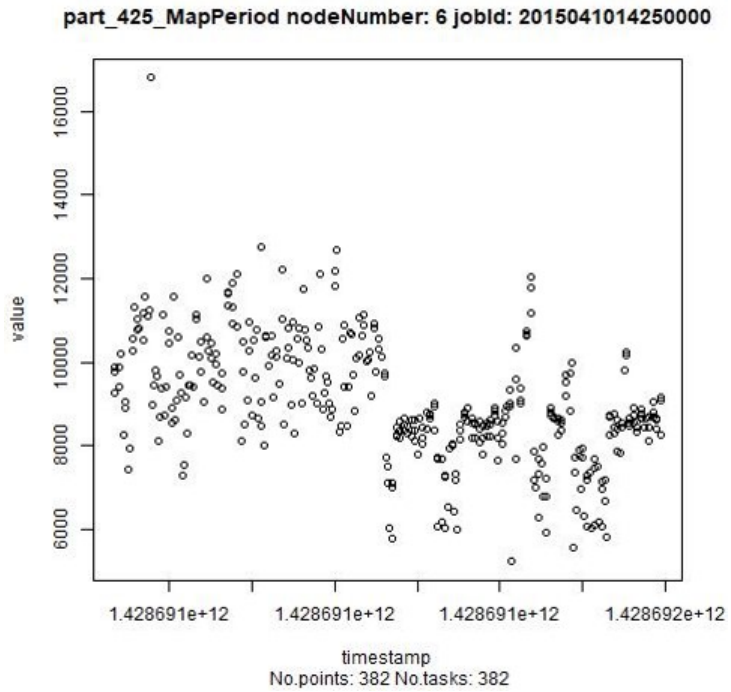


Figure 4.6: A time series for the event type MapPeriod in Hadoop dataset.

We originally planned to address two distinct yet similar issues with time series temporal abstraction techniques:

1. find a "good" segmentation that transforms a time series into a sequence of interval-based symbols;
2. find a representation which can be eventually expressed by means of Kleene closures (thus recognized by the SASE language) for further temporal analysis which focus on the potentialities of the NFA-based CEP systems.

An example of this last objective is found in Fig.4.6 where one can express the depicted time series (using SASE [Wu et al., 2006] language) as:

```
SEQ(MapPeriod+ a[], MapPeriod+ b[])
WHERE avg(a[].value) == 10000 AND avg(b[].value) == 8000
WITHIN 1000 s
```

4.4.1 Trend-based segmentation

As discussed in Sec. 1.5, the concept of *trend* is of common interest in many domains such as finance or climatology.
We believe that identifying trends can help to detect interesting "unit blocks" that can be combined to obtain a complex rule for pattern matching over events.
The problem for a trend-based segmentation of a time series is defined as follows:

> Given an event stream (for a specific type in a certain partition), represented as a time series, segment it by means of trend abstraction, i.e. such that each segment can be represented by a `trend`/`flat` segment.

We label a segment as `trend` or `flat` segment according to the following definitions:
Definition 5. *Trend-based labels.* We say that a trend exists for a sequence of points $X = \{x_1, x_2, ...x_n\}$ notably $\tau(X)$, if the alternate hypothesis H_a (a monotonic trend exists) is accepted using a statistical trend test T with a type I error ratio $0 < \alpha < 0.05$. We label it as a `trend` sequence with a slope/sign (i.e. «increasing» or «decreasing») determined by the test if the trend exists, `flat` otherwise.

The problem thus becomes:

> let $X = \{x_1, x_2, ...x_n\}$ be a sequence of points, represent it using only `trend` sequences (increasing or decreasing) and `flat` sequences.

The main challenge of this task ("Which is the best representation?") is shown in Fig.4.7, where we have the same time series represented in two acceptable yet different ways (i.e. two different outcomes of the algorithms we describe in the following).

Inspired by [Keogh et al., 2001], we developed three different algorithms (with slight modifications):

1. A **Sliding Window** approach that grows a segment until the statistical Mann-Kendall test is rejected; two modifications include a maximum length and a successive merging operations that merges together adjacent compatible trends.
2. A **Bottom-Up** algorithm that tries to merge together segments, starting from the finest segmentation, whenever a trend is detected; variants include a max length

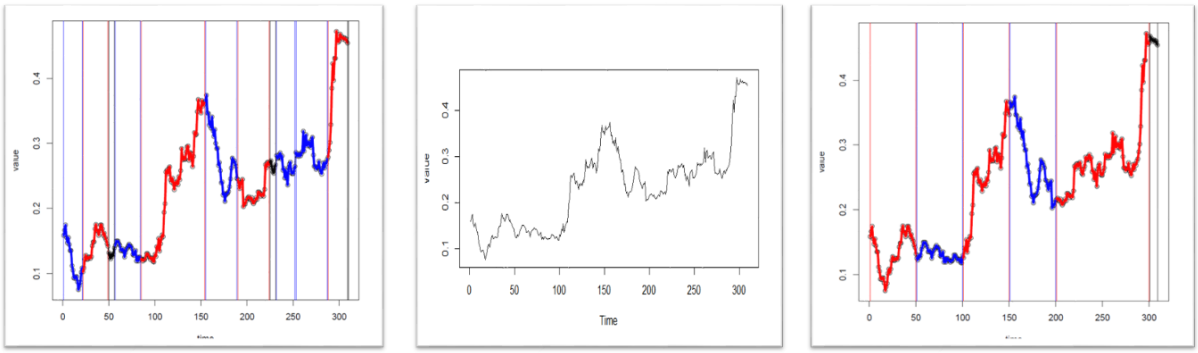


Figure 4.7: A time series segmented in two different ways (Legend: red=increasing, blue=decreasing, black=flat).

and a (optimal) greedy approach.

3. A **Sliding Window And Bottom-up (SWAB)** procedure that combines both previous flavours by applying a large sliding window and performing bottom-up in it.

They all have some parameters that needs to be specified by the user: a min/max length for a trend, the size of the sliding window and the significance value for the test; their design is flexible for adding any additional constraints (e.g. a desired "shape" for a trend, etc).

Nonetheless, the hyperparameter selection is an open problem: existing methods for Piecewise Linear Approximation focus on minimizing the error produced by a certain segmentation and Changepoint analysis techniques precisely define a min-optimization problem using a predefined cost function, whereas in our implementation we do not have an equivalent measure that allows us to discriminate different results (the *p-value* can definitely not be treated as a similarity measure).

Since there are no benchmarks nor other existing techniques for evaluating a trend-based segmentation, we eventually generated our own synthetic dataset and tested our algorithms on it.

As our approach is built on some non-parametric assumptions on data (since we use Mann Kendall test) we reproduced a dataset of time series which are characterized by non-linear trends: we "concatenated" together sequences of observations that are either

flat or represent a monotonic (upward/downward) non-linear trend; the latter is the following arbitrary "superlinear" function:

$$Y_i = \text{sign}_i(X_i + X_i \arctan(X_i) + \sqrt{X_i} + \log^3 X_i) + \text{offset}_i$$

where i is the i-th sequence and the sign s is randomly chosen at each step $\{+1, 0, -1\}$; offset_i represents the offset we add at each step to get an overall continuous signal.
In the end we add a percentage (1%, 5%, 10%) of noise, i.e. a uniform distribution with zero mean and as standard deviation the empirical one of the overall time series. You can see a few examples with their segmentation obtained using different methods in Fig.4.11. All the pseudocode is present in the Appendix.

More precisely, we built three datasets (40 time series of 2000 observations), one for each of the three different amounts of noise, and we splitted it into two halves: 20 time series to perform hyperparameter selection, 20 time series to evaluate the best configuration of each algorithm.
We thus ran the different techniques with different parameters:

- min length $w \in \{7, 10, 20, 50, 100\}$
- max length[7] $M \in \{w, 2w, 3w, 4w, 5w\}$
- $\alpha \in \{0.01, 0.02, 0.05\}$

In order to measure the accuracy of each segmentation we defined an error function based on edit distance: when applying segmentation or building a synthetic signal, we also label each point depending on the behaviour of the sequence it belongs to (0 if `flat`, 1 if `increasing` and -1 if `decreasing`), thus obtaining for each time series a string of 0 and 1; the error corresponds then to the edit distance between the ground truth, i.e. the array of labels of the synthetic signal, and the labels obtained using the segmentation.

We took the best configuration for each method in the three scenarios as the one with the lowest error and we ran again the algorithms on the other half of the three datasets accordingly to this analysis.

[7]It corresponds to the size of the sliding window in the SWAB cases.

The performances of all the algorithms are finally shown in Fig.4.8, Fig.4.9 and Fig.4.10 with each row containing the method and its best configuration: the precision is computed as

$$\text{precision} = 1 - \frac{\text{average\_error}}{2000}$$

where the average error represents the (average) percentage of misclassified points in a segmentation of a time series of 2000 points with the relative amount of noise.

We can see that the best method overall is `win-max` (the "merging" enhancement of `win-max-merge` does not actually affect the precision), with a precision up to 85% in the noisier scenario; `bottom-up-opt-max` and `SWAB` are quite competitive but we noticed that they struggle to identify `flat` segments.
In the end, imposing a maximum length for a trend segment proved to be a good idea for obtaining a good segmentation.
You can see some results in Fig.4.11.

NOISE = 1%	
Method	**Precision**
win-max W=20 M=20 alpha=0.01	95.68 %
win-max-merge W=20 M=20 alpha=0.01	95.68%
bottom-up-opt-max W=20 M=20 alpha=0.05	78.63%
bottom-up-opt-max-merge W=20 M=20 alpha=0.05	78.63%
SWAB W=10 M=20 alpha=0.02	78.23%
SWAB-merge W=10 M=20 alpha=0.02	78.23%
window W=10 M=INF alpha=0.01	59.36%
bottom-up W=10 M=INF alpha=0.01	52.94%
bottom-up-opt W=50 M=INF alpha=0.05	48.33%

Figure 4.8: The final ranking of the performances of each algorithm on the dataset with noise=1%.

NOISE = 5%	
Method	**Precision**
win-max W=50 M=50 alpha=0.01	85.91%
win-max-merge W=50 M=50 alpha=0.01	85.91%
bottom-up-opt-max-merge W=50 M=50 alpha=0.01	77.04%
SWAB W=20 M=40 alpha=0.02	75.48%
SWAB-merge W=20 M=40 alpha=0.02	75.48%
bottom-up-opt-max W=50 M=50 alpha=0.01	74.58%
window W=20 M=INF alpha=0.01	48.21%
bottom-up W=20 M=INF alpha=0.01	45.58%
bottom-up-opt W=20 M=INF alpha=0.01	41.98%

Figure 4.9: The final ranking of the performances of each algorithm on the dataset with noise=5%.

NOISE = 10%	
Method	**Precision**
win-max W=50 M=50 alpha=0.01	84.75%
win-max-merge W=50 M=50 alpha=0.01	84.75%
bottom-up-opt-max W=100 M=100 alpha=0.05	79.39%
bottom-up-opt-max-merge W=100 M=100 alpha=0.05	79.39%
SWAB W=50 M=100 alpha=0.01	77.9%
SWAB-merge W=50 M=100 alpha=0.01	77.9%
window W=50 M=INF alpha=0.01	56.77%
bottom-up W=50 M=INF alpha=0.02	55.1%
bottom-up-opt W=20 M=INF alpha=0.01	51.27%

Figure 4.10: The final ranking of the performances of each algorithm on the dataset with noise=10%.

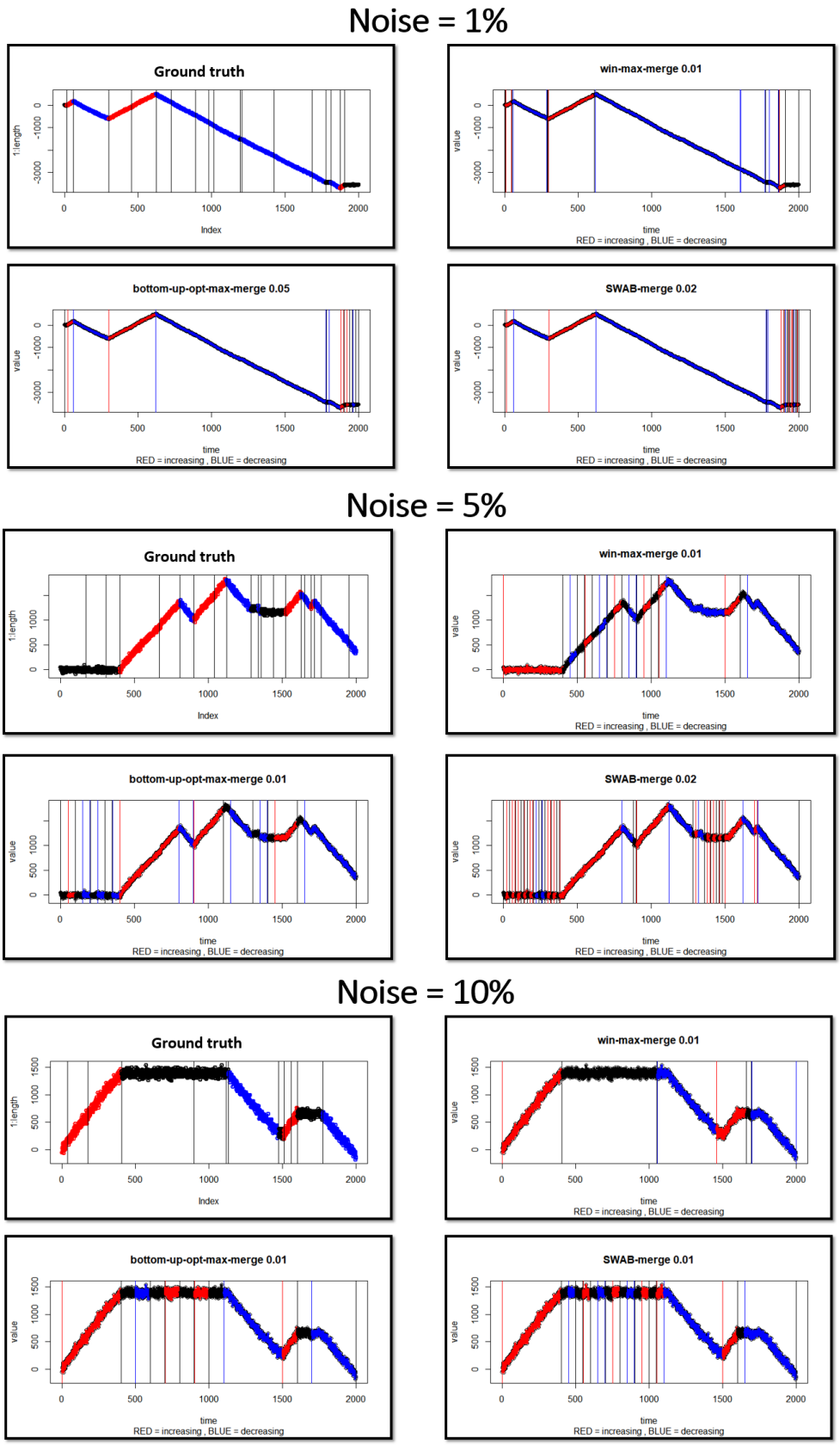


Figure 4.11: Different realizations of the best segmentation methods applied to three different noisy signals.

4.4.2 Level-based segmentation

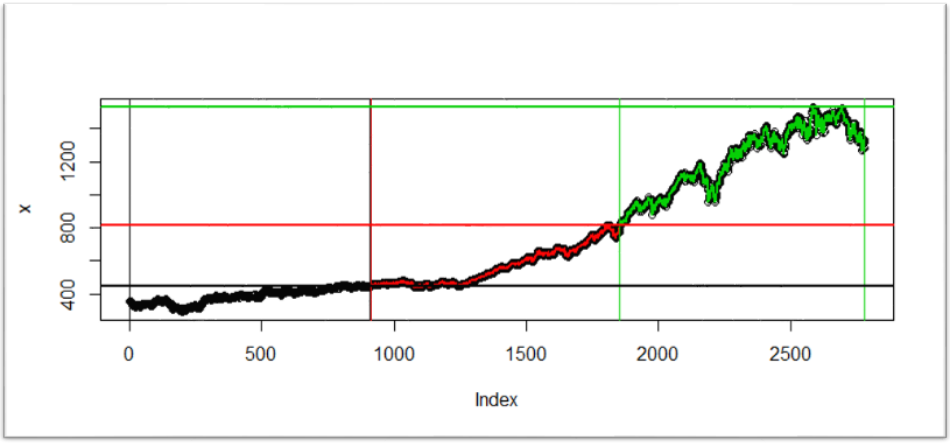


Figure 4.12: A time series segmented using level-based abstraction with three levels that correspond to the three quantiles with probabilities=(0.33, 0.67, 1).

The idea for this kind of segmentation lies in the nature of certain event types (such as memory consumption or volume of goods), that makes interesting not to focus on the precise value of a given point but rather on a set of levels (such as "LOW", "MEDIUM" and "HIGH").
A rough definition of the problem of level-based segmentation is the following:

> given a sequence X of points and $Q+1$ levels on the y-axis defined by Q values (e.g. percentiles), find a segmentation such that each segments contains up to a tolerance α of outliers chosen by the user, i.e. such that the error computed (using a customized function) on that segment is less or equal α.

For such goal we developed two simple different algorithms: one based on a sliding window and another using a bottom-up approach (similarly to the algorithms of previous section). An example of this kind of segmentation is shown in Fig. 4.12 and the pseudocode can be found in the Appendix.

4.4.3 Changepoint analysis

As already discussed in Sec. 2.2, the problem of changepoint detection is to estimate the point(s) at which the statistical properties of a sequence of points change. The traditional setting is to identify changes in mean or in variance.

This kind of analysis is actually strictly connected to the problem of the segmentation of a time series, as these particular points allow to divide a sequence of points into segments, each characterized by different statistical parameters.
Detecting statistical changes in mean and/or variance is consistent with our final goal of detecting interesting scenarios in time series and, moreover, they can be easily expressed by means of SASE language.
However, we did not apply any changepoint analysis for time series segmentation (as domain knowledge is required to perform model selection) but rather considered it as part of our future directions which do not focus on frequent pattern mining but statistically perform pattern and motif discovery in time series.

4.5 Towards CEP rules

So far, the pipeline we developed allows to get consistent partitions of event streams based on `event_type` and a set of *dimensions* and *measures*; events that carry continuous information are in turn modeled as time series and abstracted using different kind of representations.
The next step is therefore to extract interesting patterns in these datasets of time series: roughly speaking we could consider each partition as a "transaction" (in the traditional transactional database setting for pattern mining) and compute the *interestingness* of a pattern across all transactions, i.e. partitions.

In the following, we will use the notions of "intervals" and "symbols" to refer to the same concept, i.e. a symbol with duration over an interval as seen in Section. 2.4.
We carried out an exploratory analysis on this data mining technique in order to see if interesting behaviours (whether normal or abnormal) can be detected by means of pattern mining on interval sequences obtained from properly processing event streams.
This is achieved by first developing a simple mining algorithm and in turn setting up three basic scenarios where to test this approach.
Lastly, the goal would be to convert some sort of association rules into CEP pattern expressions.

4.5.1 An interval-based pattern mining algorithm

In order to apply the pattern mining paradigm discussed previously we developed a very simple algorithm that is built on the following assumptions:

- Given a sequence of intervals, time gaps between symbols must be taken into account: intervals should not be correlated with other intervals which are too far away in time;
- using Allen's relationships is too much costly: given S symbols and K relationships, the number of possible existing patterns is $\frac{K^{|S|}-1}{2}$.

Keeping these considerations in mind, we finally designed an algorithm based on a sliding window which can use two different subsets of Allen's 13 relations: (*before*, *co-occurs*) and (*before*, *overlaps*, *contains*).
Moreover, we allowed the user to specify two different *windowing* conditions: one based on a "step" concept and another based on a "time window".

Practically speaking (the pseudocode is provided in the Appendix) the heuristic is the following: given a partition ordered lexicographically by `start`, `end` and `symbol`, we scan all the symbols, applying for each of them, say *current*, a *windowing* condition in order to obtain a set of symbols, namely *candidates*; we finally build all the possible existing relations between `current` and all the possible subsets of the set of candidates.
Given a current symbol *curr* the two windowing conditions define two different sets of candidates:

$$C_{step} = \{s_j \in C_{step} : s_j.\text{index} \leq curr.\text{index} + \text{step}\} \quad \text{(step approach)}$$

$$C_{time} = \{s_j \in C_{time} : s_j.\text{end} \leq curr.\text{start} + \text{limit}\} \quad \text{(time approach)}$$

Basically with the first approach we consider in our set of *candidate* symbols the first n=step events that follow *current* in the sequence of intervals, whereas with the second approach all the symbols that do not end after *current* has ended.

Building only existing patterns (and not all the possible relations between symbols) "reduces" the computation cost to $2^{|S|} - 1$ patterns for $|S|$ symbols.

4.5.2 Simulation

The following is a series of simulations that we ran in order to validate our approach.
We reproduced streams of events and then applied the entire pipeline: partitioning into event types (and according to some dimensions), abstracting time series and running the frequent temporal pattern mining algorithm.

Retail store

This scenario simulates a set of customers in a retail store equipped with RFID sensors[8] which can make three different possible actions: buy a product, taking a product and putting back on the shelf ("indecision") and shoplift (i.e. taking a product from the shelf and exiting without passing by the counter).
We thus define three different event types (`shelf`, `counter`, `exit`) and the following attributes for each of them:

- *dimensions*: `customer_id` and `product_id`
- *measures*: `price`, `quantity`

The actions are represented by a sequence `shelf-counter-exit` for a normal buying, a sequence `shelf-shelf` for indecision and a sequence `shelf-exit` for shoplifting.
We assigned some probabilities to each action and modeled the occurrence of each event with a Poisson distribution.
The preliminary phase is to partition by both dimensions, therefore extracting frequent patterns across all partitions, each of which contains the events relative to a particular customer and a specific product.
We take into account just the occurrences of the events in this scenario (there is no time series segmentation) and we prove that partitioning events is useful to detect interesting situations (either normal or abnormal).

[8] Similarly to AmazonGo https://www.amazon.com/b?node=16008589011.

You can see two different results for the most frequent patterns obtained with both the "step" (`step`=2) and the "window" approach (we used a size of 1.5λ) in Fig.4.14 and Fig.4.13. We obtained these results by using the following parameters:

- No. products = 10
- No. customers = 5
- No. actions per customer = 100
- $Pr(\text{Normal buying}) = 0.8$
- $Pr(\text{Indecision}) = 0.1$
- $Pr(\text{Shoplift}) = 0.05$
- $\lambda = 1/60$ (we modeled the occurrences as described in http://preshing.com/20111007/how-to-generate-random-timings-for-a-poisson-process/)

N.B: In all the results that are shown, the y-axis is of no interest: plots were produced in the R environment and the y-value is only used in order to show the events in a friendly way, i.e. with a proper spacing. The x-axis, instead, correctly shows the time axis.

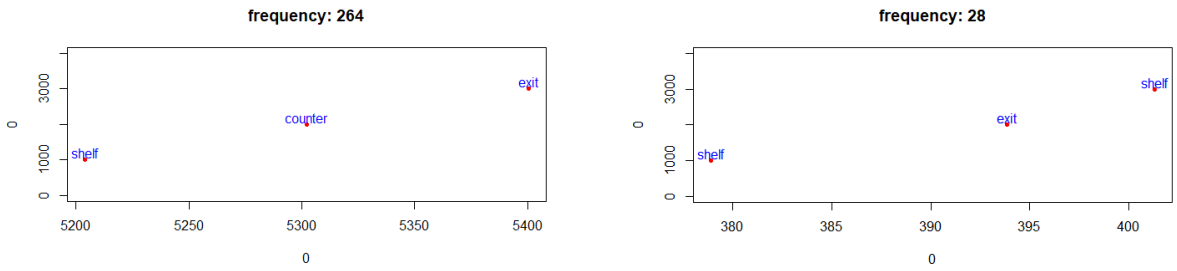


Figure 4.13: The most frequent pattern for retail store scenario using "window" approach and another 3-length patterns that interestingly captures some shoplifts (20 against the actual 26).

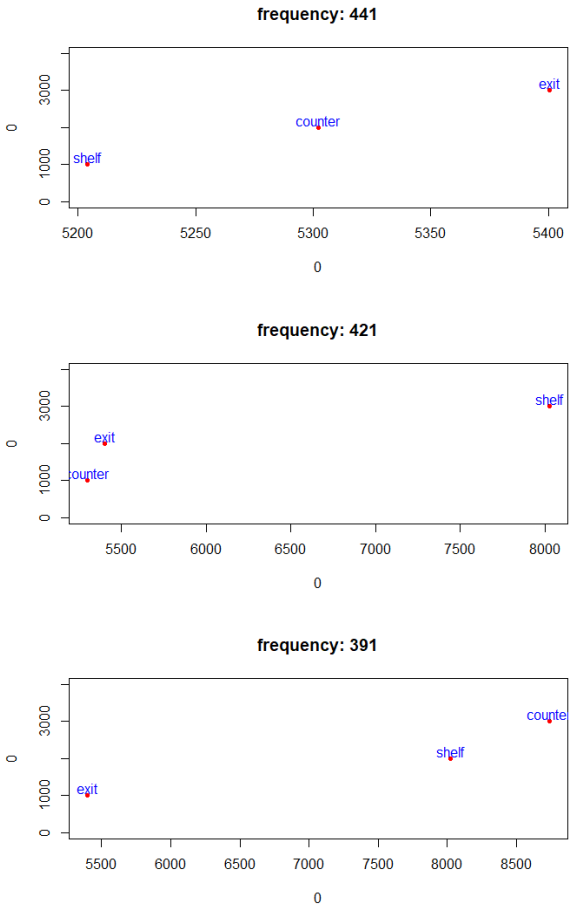


Figure 4.14: The top-3 frequent patterns for the retail store scenario using "step" approach: you can see that they are all equivalent in terms of meaning as they indicate a normal buying sequence.

Stocks

The next scenario has been designed to combine temporal pattern mining with time series segmentation: we built a set of signals representing stock indexes which are similar to the synthetic time series that we employed for evaluating trend-based segmentation.
We in fact concatenated sequences of "unit segments" such that the normal behaviour would consist of monotonic trends always followed by a flat region, while a "strange" behaviour would result in any abrupt changes, i.e. a sequence of two monotonic trends with opposite slope.
You can see an example of such signals with 4 consecutive strange behaviours highlighted in red in Fig.4.15.

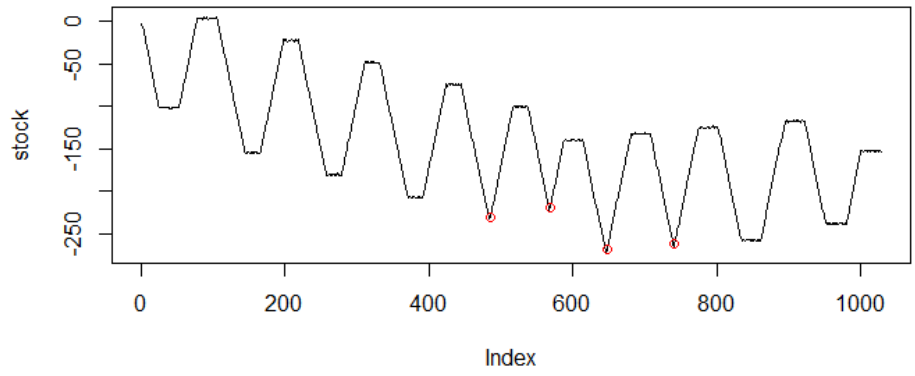


Figure 4.15: A stock signal with four abrupt changes; the noise was limited to 0.01%.

We do not defined dimensions or measures as each partition practically corresponds to a signal. We used the following parameters:

- No. stocks ids = 10
- No. observations per time series = 1000
- Pr(Normal behaviour) = 0.8
- Pr(Abrupt change) = 0.2
- $\lambda = 1/60$

- Trend segmentation = `win_max_merge` (with $w = 20$, $M = 20$ and $\alpha = 0.01$)

The results for the "step" algorithm (`step`=2) are shown in Fig. 4.16.

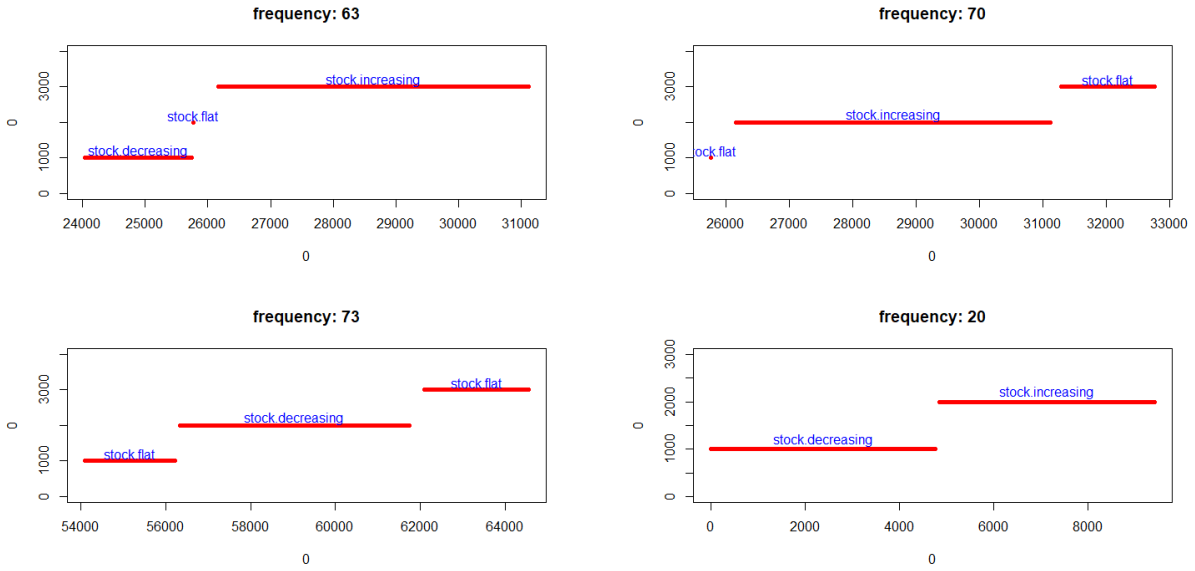


Figure 4.16: The top-3 frequent patterns for the stock scenario using "step" approach and an interesting pattern that captures the total number of abrupt changes (20 against the actual 30).

Fire detection

In this last scenario we aim to combine level-based segmentation together with interval-based pattern mining (and, of course, partitioning along dimensions).
We thus simulate a certain numbers of areas where the levels of temperature and humidity are recorded and an additional sensor tells whether there is a fire or not (as in the examples from [Giatrakos et al., 2017] described in the Section.**??**).
Therefore we have three different `event_type` (`temperature`, `humidity`, `fire`) and different areas with several events corresponding to the sensors measures in such areas. We model the occurrence of the events with a Poisson distribution as in the previous examples.
The attributes are, for each event:

- *dimensions*: `area` and `type`
- *measures*: `value`.

The level-based segmentation is applied on temperature and humidity time series (using the attribute `value`) whose values are randomly chosen at each occurrence; precisely, they are picked from two different uniform distributions depending on the kind of situations they describe.
Overall we have:

- No. areas = 5
- No. events in each area = 300
- Pr(No fire) = 0.9; this corresponds to values of temperature in [1,50] and humidity in [1,10]
- Pr(Fire) = 0.1; this corresponds to values of temperature in [51,100] and humidity in [11,20]
- $\lambda = 1/60$
- Level segmentation = `window` with 5 levels corresponding to the quantiles of probabilities (0.2, 0.4, 0.6, 0.8, 1); the 5th level contains exactly those values of temperature and humidity present in "fire:yes" situations

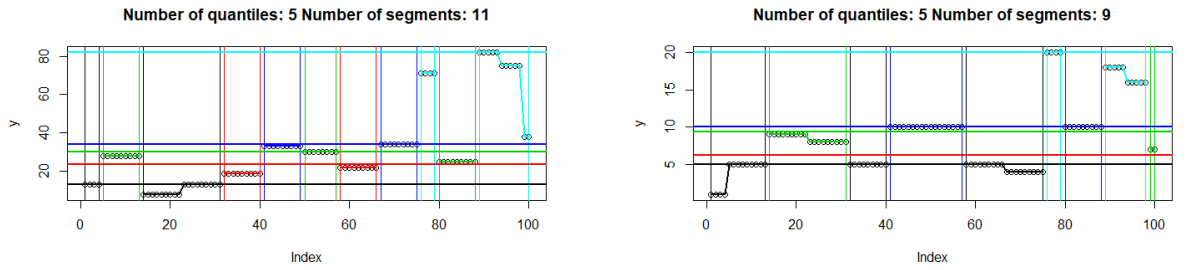


Figure 4.17: A level-segmentation for both temperature and humidity values. The fifth level corresponds to those values that trigger fire.

You can see an example of the segmentation in Fig.4.17 and some results of the pattern mining procedure in Fig.4.18.

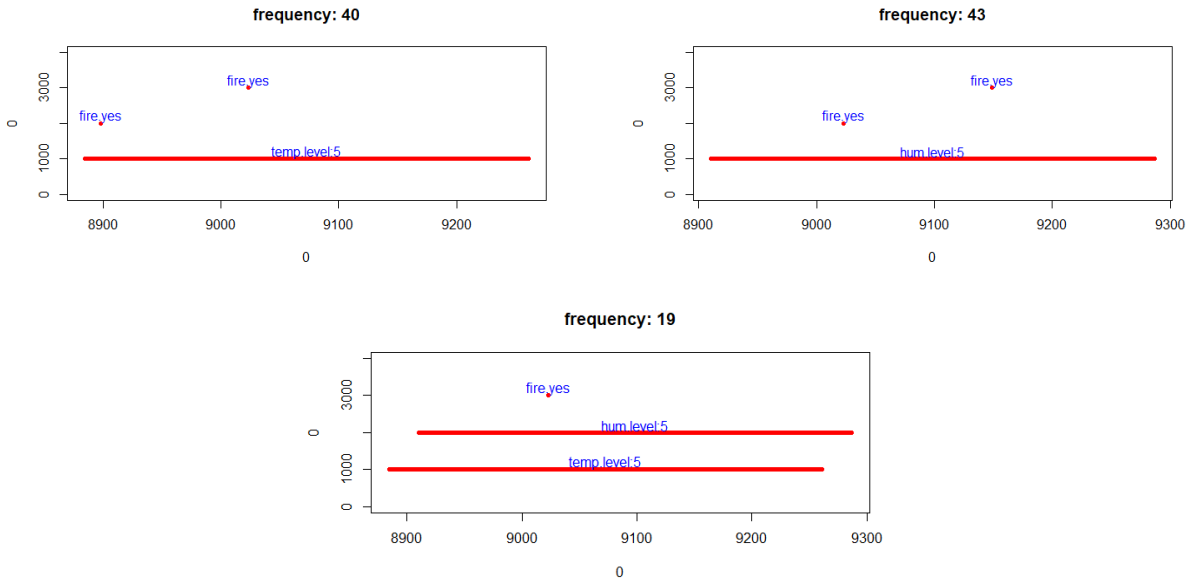


Figure 4.18: Some results from the fire detection scenario: they all suggest useful association rules which capture the relation between fire and levels of temperature and humidity.

4.5.3 Some remarks on the temporal pattern mining algorithm

So far, we have seen that combining temporal pattern mining with time series and segmentation allows to extract potential correlations between data, at least in some simple synthetic scenarios.

Nonetheless, the results also showed several critical aspects that we will resume in the following.

Firstly, when scanning the dataset to extract patterns we observe that the number of patterns has an exponential blow-up: given N symbols in a particular window (built using the two criteria of our algorithm), there are $(2^N - 1)$ existing patterns that are built, one for each non-empty subset. This clearly does not scale when considering large sequences of intervals.

Moreover, we can argue that the representation clearly affects the final results: e.g. in the second scenario (Stocks) if no flat segments are detected by the trend-segmentation algorithm than it is harder to recognize the normal behaviour. Therefore, how to choose the best representation (and the best parameters) for a given dataset?
Answering to this question implies a preliminary exploratory analysis of the data which does not facilitate at all the path to a completely automatic overall procedure, where no user intervention is needed.

Concerning the pattern mining algorithm, a choice must be made on the criterion and its parameter: different sizes of the resulting window of candidates may severely affect the final results.
This choice again depends on the preliminary exploration of the data that we suggested beforehand.

Moving on, the concept of frequency may not be always relevant to detect interesting correlations as some patterns usually show some "semantics" redundancy: in the first scenario different frequent patterns actually refer to the same underlying behaviour; moreover, some segmentations may produce representations where the "normality" is not frequent as there can be a few yet very large interval-based symbols that describe this behaviour.
To this extent, different criteria should be further analyzed such as the concept of set coverage (tiles) or the interestingness based on entropy [Lee et al., 2014].

Chapter 5

Conclusion and future directions

As stated in the preface, the intention of this work was to pave the way towards a completely automatic rules learning tool for Complex Event Processing (CEP), by means of a brand new approach that combined time series and temporal pattern analysis.
As large-scale event-based distributed systems are more and more gaining adoption in many application domains (from finance to health-care) we believe that the issue of manually writing complex rules for pattern matching over streams of events is a huge limitation for developing proactive and reactive solutions in distributed architectures.

We have preliminary discussed how the main obstacle of such goal resides in the expressiveness of many Event Processing Languages. This, in fact, opens up a vast set of correlations that can be described by Complex Event Processing engines when processing flows of information items.
As a matter of fact, we proved that applying a simple brute-force search procedure to generate all the possible existing rules that match a given input stream is not feasible, even when considering simple patterns such as sequences of events.

The main contribution of this work, is a complete pipeline that operates in a off-line manner, i.e. by processing a finite batch of events at a time. However, all the algorithms of each module work in a streaming fashion and the approach can be easily extended to a real-time scenario where streams are processed on-the-fly.

The first step consists of partitioning the events along some consistent attributes (*dimensions*) and successively model some other attributes values (*measures*) as (multivariate) time series. In a simplistic scenario we assumed that the user is always capable of providing such context along with the data.
In this way, we can apply time series segmentation and combine several different temporal abstractions with a temporal pattern mining algorithm.

To this extent, we developed two "customized" kinds of segmentations for time series, one based on the concept of *trend* and the other on the concept of *level* ("on the Y-axis"). Hyperparameter tuning is also performed on the former by using some synthetic datasets. We eventually designed our own temporal pattern mining algorithm for the final stage of the pipeline and we tested it against some synthetic scenarios where the entire approach proved to be promising.
The entire software of our work can be found on:
https://github.com/Piru93/learning-cep-rules-from-multivariate-time-series.

In the future, we plan to further investigate several open issues that we encountered during our work as well as employing our technique against some real-world massive datasets.

As a first future direction, we aim to look into the different time series segmentations as to be able to perform automatically model (and hyperparameter) selection and pick always the best representation for the sake of the later phase of pattern analysis.
As a matter of fact, different representations can lead to different interpretations of the data.

Moving on, as our results showed to be promising enough to extract association rules that can be converted to Complex Event Processing rules, we plan to apply our temporal pattern mining technique to a real world dataset (such as the Hadoop's one): this requires a preliminary exploratory analysis of the data as to recover ground truths and, especially, to tune the parameters for both segmentation and pattern mining phases.
There is currently a need to understand how both "step" and "time window" parameters affect the final outcome of the mining algorithm; also the concept of frequency may not be

always the most feasible measure when trying to extract insights and different measures should be further investigated (such as tiles or entropy).

On the other hand, we also intent to consider some completely different approaches that also apply on time series models: changepoint analysis and motif and pattern discovery in multivariate time series.
The former can be exploited to detect interesting situations, i.e. changepoints, and extract correlations. The latter is a statistical approach to detect patterns in multivariate time series by considering all the components, i.e. univariate time series, at once.
They seem both promising contexts where our approach of converting insight in CEP rules by means of an Event Processing Language seem feasible.

Appendix

Pseudocode of the algorithms

Trend-based segmentation

Sliding window

Algorithm 1: Sliding window algorithm for trend segmentation.

Data: A sequence of points X, a minimum length W and a significance α.
Result: A list of segments where a monotonic trend is present.
trends= $\emptyset$
L =length(X)
$i = 1$
while $i \leq L - W + 1$ **do**
    $j = i + W - 1$
    if MannKendall($X[i : j]$).pvalue $\leq \alpha$ **then**
        repeat
            $j = j + 1$
        until $j > L$ || MannKendall($X[i : j]$).pvalue $> \alpha$;
        $a = i$
        $i = j$
        $j = j - 1$
        $b = j$
        trends.add(a,b,MannKendall($X[a : b]$)
    end
end
return(trends)

Algorithm 2: Sliding window algorithm with maximum size for trend segmentation.

Data: A sequence of points X, a minimum length W, a maximum length $M > W$ and a significance α.

Result: A list of segments where a monotonic trend is present.

trends= $\emptyset$

L =length(X)

$i = 1$

while $i \leq L - W + 1$ **do**

  $j = i + W - 1$

  if MannKendall($X[i : j]$).pvalue $\leq \alpha$ **then**

    repeat

      $j = j + 1$

    until $j > L$ || $(j - i) > M$ || MannKendall($X[i : j]$).pvalue $> \alpha$;

    $a = i$

    $i = j$

    $j = j - 1$

    $b = j$

    trends.add(a,b,MannKendall($X[a : b]$)

  end

end

return(trends)

Algorithm 3: Sliding window algorithm with maximum size and merging for trend segmentation.

Data: A sequence of points X, a minimum length W, a maximum length $M > W$ and a significance α.

Result: A list of segments where a monotonic trend is present.

```
trends= ∅
L =length(X)
i = 1
while i ≤ L − W + 1 do
    j = i + W − 1
    if MannKendall(X[i : j]).pvalue ≤ α then
        repeat
            j = j + 1
        until j > L || (j − i) > M || MannKendall(X[i : j]).pvalue > α;
        a = i
        i = j
        j = j − 1
        b = j
        trends.add(a,b,MannKendall(X[a : b])
    end
end
while no more merges are possible do
    i = 1
    while i < trends.length() do
        if adjacent(trends[i],trends[i + 1]) && accept(trends[i],trends[i + 1]) then
            merge(trends[i],trends[i + 1])
        end
        i = i + 1
    end
end
return(trends)
```

Bottom-up

Algorithm 4: Bottom-up algorithm for trend segmentation.

Data: A sequence of points X, a minimum length W and a significance α.
Result: A list of segments where a monotonic trend is present.

```
trends= ∅
flag= 0
i = 1
S =make_segments(X, W)
repeat
    flag= 1
    i = 1
    while i <length(S) do
        s = S[i].append(S[i + 1])
        pvalue = MannKendall(s).pvalue
        if pvalue ≤ α then
            flag = 0
            merge(S[i], S[i + 1])
        end
        i = i + 1
    end
until flag == 1;
foreach s in S do
    test = MannKendall(s)
    if test.pvalue ≤ α then
        trends.add(s.start, s.end, test)
    end
end
return(trends)
```

Algorithm 5: Bottom-up algorithm with best merge for trend segmentation.

Data: A sequence of points X, a minimum length W and a significance α.
Result: A list of segments where a monotonic trend is present.

```
trends= ∅
flag= 0
i = 1
S =make_segments(X, W)
repeat
    flag= 1
    i = 1
    min=MAX_INTEGER_VALUE
    while i <length(S) do
        s = S[i].append(S[i + 1])
        pvalue = MannKendall(s).pvalue
        if pvalue ≤ α && pvalue < min then
            min=MannKendall(s).pvalue
            index= i
            flag = 0
        end
        i = i + 1
    end
    if flag==0 then
        merge(S[index], S[index+1])
    end
until flag == 1;
foreach s in S do
    test = MannKendall(s)
    if test.pvalue ≤ α then
        trends.add(s.start, s.end, test)
    end
end
return(trends)
```

Algorithm 6: Bottom-up algorithm with best merge and maximum size for trend segmentation.

```
Data: A sequence of points X, a minimum length W, a maximum size M > W
      and a significance α.
Result: A list of segments where a monotonic trend is present.
trends= ∅
flag= 0
i = 1
S =make_segments(X, W)
repeat
    flag= 1
    i = 1
    min=MAX_INTEGER_VALUE
    while i <length(S) do
        s = S[i].append(S[i + 1])
        if length(s) ≤ M then
            pvalue = MannKendall(s).pvalue
        else
            pvalue = 1
        if pvalue ≤ α && pvalue < min then
            min=MannKendall(s).pvalue
            index= i
            flag = 0
        end
        i = i + 1
    end
    if flag==0 then
        merge(S[index], S[index+1])
    end
until flag == 1;
foreach s in S do
    test = MannKendall(s)
    if test.pvalue ≤ α then
        trends.add(s.start, s.end, test)
    end
end
return(trends)
```

Hybrid

Algorithm 7: Hybrid algorithm (SWAB) for trend segmentation.

Data: A sequence of points X, a minimum length M, a window size $W \sim 5M$ and a significance α.

Result: A list of segments where a monotonic trend is present.

```
trends= ∅
L =length(X)
i = 1
while i ≤ L − W + 1 do
    j = i + W − 1
    t =bottom_up(X[i : j]).first()
    if MannKendall(t).pvalue ≤ α then
        trends.add(t.start,t.end,MannKendall(X[t.start : t.end])
    end
    i =t.end+1
end
return(trends)
```

Algorithm 8: Hybrid algorithm (SWAB) with merge for trend segmentation.

Data: A sequence of points X, a minimum length M, a window size $W \sim 5M$ and a significance α.

Result: A list of segments where a monotonic trend is present.

```
trends= ∅
L =length(X)
i = 1
while i ≤ L − W + 1 do
    j = i + W − 1
    t =bottom_up(X[i : j]).first()
    if MannKendall(t).pvalue ≤ α then
        trends.add(t.start,t.end,MannKendall(X[t.start : t.end])
    end
    i =t.end+1
end
while no more merges are possible do
    i = 1
    while i < trends.length() do
        if adjacent(trends[i],trends[i + 1]) && accept(trends[i],trends[i + 1]) then
            merge(trends[i],trends[i + 1])
        end
        i = i + 1
    end
end
return(trends)
```

Level-based segmentation

Algorithm 9: Bottom up algorithm for level based segmentation.

Data: A sequence of points X, a number of levels L and a tolerance α.
Result: A list of labeled segments.

```
S = ∅
levels = make_levels(X, L)
flag = 0
i = 1
repeat
    flag = 1
    i = 1
    min = MAX_INTEGER_VALUE
    while i <length(X) do
        s = X[i].append(X[i + 1])
        error = compute_error(s,levels)
        if error ≤ α && error < min then
            min = error
            index= i
            flag = 0
        end
        i = i + 1
    end
    if flag == 0 then
        s = merge(X[index], X[index+1])
        S.add(s)
    end
until flag == 1;
foreach s in S do
    s.level = compute_level(s, levels)
end
if s.end≠ length(X) then
    s = X[length(X)]
    s.level = compute_level(s, levels)
    S.add(s)
end
return(S)
```

Algorithm 10: Sliding window algorithm for level based segmentation.

Data: A sequence of points X, a number of levels L and a tolerance α.
Result: A list of labeled segments.

```
S = ∅
levels = make_levels(X, L)
i = 1
while i ≤ length(X) do
    j = i
    repeat
        j = j + 1
        if j > length(X) then
            error = MAX_INTEGER_VALUE
        else
            error = compute_error(X[i : j], levels)
    until error > α;
    a = i
    b = j − 1
    S.add(X[a : b], compute_level(X[a : b], level)
    i = j
end
return(S)
```

Synthetic dataset for trend-based segmentation evaluation

Algorithm 11: Building a synthetic signal for trend-segmentation evaluation.

Data: A maximum number of breakpoints B, a length L and a noise N.
Result: A list of symbols.
Y =array()
repeat
  bp =random_breakpoints$(1, B, L)$
until min distance between breakpoints is 10;
S =make_segments(Y, bp, L)
foreach s in S **do**
  $sign$ =random$(-1, 0, +1)$
  $s = sign$*non_linear_(s)
end
std =empirical_std(S)
$Y = S$+uniform_errors(mean= 0,std= $N * std$)
return(S)

Interval-based pattern mining

Algorithm 12: A temporal pattern mining algorithm.

Data: A sequence of intervals I, a windowing condition *cond* and the size of the window *win*.

Result: A list of temporal patterns with their frequency.

```
P = ∅
i = 1
while i ≤length(I) do
    current = I[i]
    if cond == "time" then
        candidates = build_candidates_time(I, i, current.start + win)
    else
        candidates = build_candidates_step(I, i, i + win)
    P.add(build_patterns(current, candidates)
    i = i + 1
end
return(P)
```

Algorithm 13: Building candidates set using "step".

Data: A sequence of intervals I, a current *index* and a *step*.

Result: A list of symbols.

```
i = index + 1
C = ∅
while i ≤length(I) − step do
    C.add(I[i])
end
return(C)
```

Algorithm 14: Building candidates set using "time".

Data: A sequence of intervals I, a current *index* and a *time*.

Result: A list of symbols.

```
i = index + 1
C = ∅
while i ≤length(I) do
    if I[i].end> time then
        break
    end
    C.add(I[i])
end
return(C)
```

OS metrics in Ganglia

The following table describes the metrics that were monitored using Ganglia's daemon `gmond` in each of the nodes of the Hadoop's architecture we analyzed in Section.?? (taken from their official GitHub page https://github.com/ganglia/monitor-core/blob/master/ganglia.pod).

Metric Name	Description
boottime	System boot timestamp
bytes_in	Number of bytes in per second
bytes_out	Number of bytes out per second
cpu_aidle	Percent of time since boot idle CPU
cpu_idle	Percent CPU idle
cpu_nice	Percent CPU nice
cpu_num	Number of CPUs
cpu_speed	Speed in MHz of CPU
cpu_system	Percent CPU system
cpu_user	Percent CPU user
disk_free	Total free disk space
disk_total	Total available disk space
load_fifteen	Fifteen minute load average
load_five	Five minute load average
load_one	One minute load average
mem_buffers	Amount of buffered memory
mem_cached	Amount of cached memory
mem_free	Amount of available memory
mem_shared	Amount of shared memory
mem_sreclaimable	Amount of slab reclaimable memory
mem_total	Amount of available memory
part_max_used	Maximum percent used for all partitions
pkts_in	Packets in per second
pkts_out	Packets out per second
proc_run	Total number of running processes
proc_total	Total number of processes
swap_free	Amount of available swap memory
swap_total	Total amount of swap memory

Bibliography

[Agarwal et al., 2001] Agarwal, R. C., Aggarwal, C. C., and Prasad, V. (2001). A Tree Projection Algorithm for Generation of Frequent Item Sets. *Journal of Parallel and Distributed Computing*, 61(3):350–371.

[Agrawal et al., 2008] Agrawal, J., Diao, Y., Gyllstrom, D., and Immerman, N. (2008). Efficient pattern matching over event streams. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, page 147.

[Agrawal and Srikant, 1994] Agrawal, R. and Srikant, R. (1994). Fast Algorithms for Mining Association Rules in Large Databases. *Journal of Computer Science and Technology*, 15(6):487–499.

[Batal et al., 2016] Batal, I., Cooper, G. F., Fradkin, D., Harrison, J., Moerchen, F., and Hauskrecht, M. (2016). An efficient pattern mining approach for event detection in multivariate temporal data. *Knowledge and Information Systems*, 46(1):115–150.

[Batal et al., 2012] Batal, I., Fradkin, D., Harrison, J., Moerchen, F., and Hauskrecht, M. (2012). Mining recent temporal patterns for event detection in multivariate time series data. *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '12*, page 280.

[Chan and Pătraşcu, 2010] Chan, T. M. and Pătraşcu, M. (2010). Counting inversions, offline orthogonal range counting, and related problems. *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete algorithms*.

[Chen et al., 2015] Chen, Y. C., Peng, W. C., and Lee, S. Y. (2015). Mining Temporal Patterns in Time Interval-Based Data. *IEEE Transactions on Knowledge and Data Engineering*, 27(12):3318–3331.

[Cugola and Margara, 2010] Cugola, G. and Margara, A. (2010). Tesla: a formally defined event specification language. pages 50–61.

[Cugola and Margara, 2012] Cugola, G. and Margara, A. (2012). Complex event processing with t-rex. *Journal of Systems and Software*, 85(8):1709–1728.

[Demers et al., 2007] Demers, A. J., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., White, W. M., et al. (2007). Cayuga: A general purpose event monitoring system. 7:412–422.

[Esling and Agon, 2012a] Esling, P. and Agon, C. (2012a). Time-series data mining. *ACM Computing Surveys*, 45(1):1–34.

[Esling and Agon, 2012b] Esling, P. and Agon, C. (2012b). Time-series data mining. *ACM Computing Surveys*, 45(1):1–34.

[Faloutsos et al., 1994] Faloutsos, C., Ranganathan, M., and Manolopoulos, Y. (1994). Fast subsequence matching in time-series databases. *SIGMOD Rec.*, 23(2):419–429.

[Fryzlewicz, 2014] Fryzlewicz, P. (2014). Wild Binary Segmentation for multiple change-point detection Segmentation in a simple function + noise model. (January).

[Fu, 2011] Fu, T. C. (2011). A review on time series data mining. *Engineering Applications of Artificial Intelligence*, 24(1):164–181.

[George et al., 2016a] George, L., Cadonna, B., and Weidlich, M. (2016a). IL-Miner: Instance-Level Discovery of Complex Event Patterns. *Proceedings of the VLDB Endowment (PVLDB)*, 10(1):25–36.

[George et al., 2016b] George, L., Cadonna, B., and Weidlich, M. (2016b). IL-Miner: Instance-Level Discovery of Complex Event Patterns. *Proceedings of the VLDB Endowment (PVLDB)*, 10(1):25–36.

[Giatrakos et al., 2017] Giatrakos, N., Artikis, A., Deligiannakis, A., and Garofalakis, M. (2017). Complex Event Recognition in the Big Data Era. 10(12):1996–1999.

[Guralnik and Srivastava, 1999] Guralnik, V. and Srivastava, J. (1999). Event detection from time series data. *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 33–42.

[Gyllstrom et al., 2006] Gyllstrom, D., Diao, Y., and Wu, E. (2006). SASE: Complex Event Processing over Streams.

[Han et al., 2011] Han, J., Pei, J., and Kamber, M. (2011). *Data mining: concepts and techniques*. Elsevier.

[Han et al., 2000] Han, J., Pei, J., and Yin, Y. (2000). Mining frequent patterns without candidate generation. *Proceedings of the 2000 ACM SIGMOD international conference on Management of data - SIGMOD '00*, pages 1–12.

[Haynes et al., 2014] Haynes, K., Eckley, I. A., Fearnhead, P., and Dec, C. O. (2014). Efficient penalty search for multiple changepoint problems. pages 1–23.

[Haynes et al., 2017] Haynes, K., Fearnhead, P., and Eckley, I. A. (2017). A computationally efficient nonparametric approach for changepoint detection. *Statistics and Computing*, 27(5):1293–1305.

[Helsel and Hirsch, 2002] Helsel, D. and Hirsch, R. M. (2002). *Statistical Methods in Water Resources Techniques of Water Resources Investigations*.

[Hyndman and Athanasopoulos, 2017] Hyndman, R. J. and Athanasopoulos, G. (2017). *Forecasting: principles and practices*.

[Keogh et al., 2001] Keogh, E., Chu, S., Hart, D., and Pazzani, M. (2001). An online algorithm for segmenting time series. *Proceedings 2001 IEEE International Conference on Data Mining*, pages 289–296.

[Killick et al., 2012] Killick, R., Fearnhead, P., and Eckley, I. A. (2012). Optimal detection of changepoints with a linear computational cost. *Journal of the American Statistical Association*, 107(500):1590–1598.

[Knight, 1966] Knight, W. (1966). A computer method for calculating kendall's tau with ungrouped data. *Journal of the American Statistical Association*.

[Lee et al., 2014] Lee, V. E., Jin, R., and Agrawal, G. (2014). *Frequent Pattern Mining*.

[Lin et al., 2003] Lin, J., Keogh, E., Lonardi, S., and Chiu, B. (2003). A symbolic representation of time series, with implications for streaming algorithms. *Proceedings of*

the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery - DMKD '03, page 2.

[Maidstone et al., 2017] Maidstone, R., Hocking, T., Rigaill, G., and Fearnhead, P. (2017). On optimal multiple changepoint algorithms for large data. *Statistics and Computing*, 27(2):519–533.

[Margara et al., 2014a] Margara, A., Cugola, G., and Tamburrelli, G. (2014a). Learning from the past: automated rule generation for complex event processing. *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems - DEBS '14*, pages 47–58.

[Margara et al., 2014b] Margara, A., Cugola, G., and Tamburrelli, G. (2014b). Learning from the past: automated rule generation for complex event processing. *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems - DEBS '14*, pages 47–58.

[Meals et al., 2011] Meals, D., J., S., S.A., D., and J.B., H. (2011). Statistical Analysis for Monotonic Trends. *TechNotes 6*, pages 1–23.

[Mehdiyev et al., 2015] Mehdiyev, N., Krumeich, J., Enke, D., Werth, D., and Loos, P. (2015). Determination of Rule Patterns in Complex Event Processing Using Machine Learning Techniques. *Procedia Computer Science*, 61:395–401.

[Mehdiyev et al., 2016] Mehdiyev, N., Krumeich, J., Werth, D., and Loos, P. (2016). Determination of Event Patterns for Complex Event Processing Using Fuzzy Unordered Rule Induction Algorithm with Multi-objective Evolutionary Feature Subset Selection. *49th Hawaii International Conference on System Sciences (HICSS)*, pages 1719–1728.

[Mei and Madden, 2009] Mei, Y. and Madden, S. (2009). Zstream: a cost-based query processor for adaptively detecting composite events. pages 193–206.

[Moerchen, 2006] Moerchen, F. (2006). Time Series Knowledge Mining Fabian Moerchen Dissertation. *Time*, page 178.

[Mörchen, 2007] Mörchen, F. (2007). Unsupervised pattern mining from symbolic temporal data. *ACM SIGKDD Explorations Newsletter*, 9(1):41–55.

[Moskovitch and Shahar, 2015a] Moskovitch, R. and Shahar, Y. (2015a). Classification of multivariate time series via temporal abstraction and time intervals mining. *Knowledge and Information Systems*, 45(1):35–74.

[Moskovitch and Shahar, 2015b] Moskovitch, R. and Shahar, Y. (2015b). Fast time intervals mining using the transitivity of temporal relations. *Knowledge and Information Systems*, 42(1):21–48.

[Mousheimish et al., 2016] Mousheimish, R., Taher, Y., and Zeitouni, K. (2016). Doctoral symposium: Automatic learning of predictive rules for complex event processing. *DEBS 2016 - Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*, pages 414–417.

[Onoz Bayazit, M., 2003] Onoz Bayazit, M., B. (2003). The Power of Statistical Test for Trend Detection. *Turkish Journal of Engineering and Environmental Sciences*, 27:247–251.

[Petersen et al., 2016] Petersen, E., To, M. A., and Maag, S. (2016). An online learning based approach for CEP rule generation. *2016 8th IEEE Latin-American Conference on Communications (LATINCOM)*, pages 1–6.

[Rawassizadeh et al., 2016] Rawassizadeh, R., Momeni, E., Dobbins, C., Gharibshah, J., and Pazzani, M. (2016). Scalable Daily Human Behavioral Pattern Mining from Multivariate Temporal Data. *IEEE Transactions on Knowledge and Data Engineering*, 28(11):3098–3112.

[Scott and Knott, 1974] Scott, A. J. and Knott, M. (1974). A cluster analysis method for grouping means in the analysis of variance. *Biometrics*, pages 507–512.

[Tang et al., 2017] Tang, B., Han, S., Lung, M., Rui, Y., and Dongmei, D. (2017). Extracting Top-K Insights from Multi-dimensional Data. *Sigmod*, pages 1509–1524.

[Vartak et al., 2015] Vartak, M., Rahman, S., Madden, S., Parameswaran, A., and Polyzotis, N. (2015). SeeDB: Efficient Data-Driven Visualization Recommendations to Support Visual Analytics. *Proceedings of the VLDB Endowment*, 8(13):2182–2193.

[White, 2009] White, T. (2009). *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition.

[Wu et al., 2006] Wu, E., Diao, Y., and Rizvi, S. (2006). High-performance complex event processing over streams. *Proceedings of the 2006 ACM SIGMOD international conference on Management of data - SIGMOD '06*, 10(1):407.

[Yang et al., 2006] Yang, Q., Wu, X., Domingos, P., Elkan, C., Gehrke, J., Han, J., Heckerman, D., Keim, D., Liu, J., Madigan, D., Piatetsky-Shapiro, G., Raghavan, V. V., Rastogi, R., Stolfo, S. J., Tuzhilin, A., and Wah, B. W. (2006). 10 Challenging Problems in Data Mining Research. *International Journal of Information Technology & Decision Making*, 5(4):597–604.

[Yue et al., 2002] Yue, S., Pilon, P., and Cavadias, G. (2002). Power of the Mann-Kendall and Spearman's rho tests for detecting monotonic trends in hydrological series. *Journal of Hydrology*, 259(1-4):254–271.

[Zhang et al., 2014] Zhang, H., Diao, Y., and Immerman, N. (2014). On complexity and optimization of expensive queries in complex event processing. *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14*, pages 217–228.

[Zhang et al., 2017] Zhang, H., Diao, Y., and Meliou, A. (2017). EXstream: Explaining Anomalies in Event Stream Monitoring. *Proceedings of 20th International Conference on Extending Database Technology*, pages pp. 156–167.

[Zhangn et al., 2013] Zhangn, H., Diao, Y., and Immerman, N. (2013). Recognizing patterns in streams with imprecise timestamps. *Information Systems*, 38(8):1187–1211.