Noname manuscript No.
(will be inserted by the editor)

Complex Event Forecasting with Prediction Suffix Trees

Elias Alevizos · Alexander Artikis · Georgios Paliouras

Received: date / Accepted: date

Abstract Complex Event Recognition (CER) systems have become popular in the past two decades due to their ability to "instantly" detect patterns on real-time streams of events. However, there is a lack of methods for forecasting when a pattern might occur before such an occurrence is actually detected by a CER engine. We present a formal framework that attempts to address the issue of Complex Event Forecasting (CEF). Our framework combines two formalisms: a) symbolic automata which are used to encode complex event patterns; and b) prediction suffix trees which can provide a succinct probabilistic description of an automaton's behavior. We compare our proposed approach against state-of-the-art methods and show its advantage in terms of accuracy and efficiency. In particular, prediction suffix trees, being variable-order Markov models, have the ability to capture long-term dependencies in a stream by remembering only those past sequences that are informative enough. We also discuss how CEF solutions should be best evaluated on the quality of their forecasts.

Elias Alevizos
Department of Informatics, National and Kapodistrian University of Athens, Greece
Institute of Informatics & Telecommunications, National Center for Scientific Research "Demokritos", Greece
E-mail: ilalev@di.uoa.gr,alevizos.elias@iit.demokritos.gr

Alexander Artikis
Department of Maritime Studies, University of Piraeus, Greece
Institute of Informatics & Telecommunications, National Center for Scientific Research "Demokritos", Greece
E-mail: a.artikis@unipi.gr

Georgios Paliouras
Institute of Informatics & Telecommunications, National Center for Scientific Research "Demokritos", Greece
E-mail: paliourg@iit.demokritos.gr

Keywords Finite Automata · Regular Expressions · Complex Event Recognition · Complex Event Processing · Symbolic Automata · Variable-order Markov Models

1 Introduction

The avalanche of streaming data in the last decade has sparked an interest in technologies processing high-velocity data streams. One of these technologies which have enjoyed increased popularity is Complex Event Recognition (CER) [15,20]. The main goal of a CER system is to detect interesting activity patterns occurring within a stream of events, coming from sensors or other devices. Complex Events must be detected with minimal latency. As a result, a significant body of work has been devoted to computational optimization issues. Less attention has been paid to forecasting event patterns [20], despite the fact that forecasting has attracted considerable attention in various related research areas, such as time-series forecasting [28], sequence prediction [10,37,14,42], temporal mining [40,23,43,12] and process mining [27]. The need for Complex Event Forecasting (CEF) has been acknowledged though, as evidenced by several conceptual proposals [19,13,17].

The field of moving object monitoring (for ships at sea, aircrafts in the air or vehicles on the ground) provides an example where CEF could be a crucial functionality [41]. Collision avoidance is obviously of paramount importance for this domain. A monitoring system with the ability to infer that two (or more) moving objects are on a collision course and forecast that they will indeed collide if no action is taken would provide significant help to the relevant authorities. CEF could play an important role even in in-silico biology,

where computationally demanding simulations of biological systems are often executed to determine the properties of these systems and their response to treatments [30]. These simulations are typically run on supercomputers and are evaluated afterwards to determine which of them seem promising enough from a therapeutic point of view. A system that could monitor these simulations as they run, forecast which of them will turn out to be non-pertinent and decide to terminate them at an early stage, could thus save valuable computational resources and significantly speed-up the execution of such in-silico experiments. Note that these are domains with different characteristics. For example, some of them have a strong geospatial component (monitoring of moving entities), whereas in others this component is minimal (in-silico biology). Domain-specific solutions (e.g., trajectory prediction for moving objects) cannot thus be universally applied. We need a more general framework.

Towards this direction, we present a formal framework for CEF, along with an implementation and extensive experimental results on real and synthetic data from diverse application domains. Our framework allows a user to define a pattern for a complex event, e.g., a pattern for two moving objects moving in close proximity and towards each other. It then constructs a probabilistic model for such a pattern in order to forecast, on the basis of an event stream, if and when a complex event is expected to occur. We use the formalism of symbolic automata [16] to encode a pattern and that of prediction suffix trees [37,36] to learn a probabilistic model for the pattern. We formally show how symbolic automata can be combined with prediction suffix trees to perform CEF. Prediction suffix trees fall under the class of the so-called variable-order Markov models, i.e., Markov models whose order (how deep into the past they can look for dependencies) can be increased beyond what is computationally possible with full-order models. They can do this by avoiding a full enumeration of every possible dependency and focusing only on "meaningful" dependencies.

Our empirical analysis shows the advantage of being able to use high-order models over related non-Markov methods for CEF and methods based on low-order Markov models (or Hidden Markov Models). The price we have to pay for this increased accuracy is a decrease in throughput, which still however remains high (typically tens of thousands of events per second). The training time is also increased, but still remains within the same order of magnitude. This fact allows us to be confident that training could also be performed online. Our contributions may be summarized as follows:

- We present a CEF framework that is both formal and easy to use, with clear compositional semantics.
- Our framework can uncover deep probabilistic dependencies in a stream by using a variable-order Markov model. By being able to look deeper into the past, we achieve higher accuracy scores compared to other state-of-the-art solutions for CEF, as shown in our extensive empirical analysis.
- We propose a more comprehensive set of metrics that takes into account the idiosyncrasies of CEF. Besides accuracy itself, the usefulness of forecasts is also judged by their "earliness". We discuss how the notion of earliness may be quantified.

Due to space limitations, all proofs are presented in an extended technical report[1].

Running Example As an example of a CER system, consider the scenario of a system receiving an input stream consisting of events emitted from vessels sailing at sea. These events may contain information regarding the status of a vessel, e.g., its location, speed and heading. This is indeed a real-world scenario and the emitted messages are called AIS (Automatic Identification System) messages. Besides information about a vessel's kinematic behavior, each such message may contain additional information about the vessel's status (e.g., whether it is fishing), along with a timestamp and a unique vessel identifier. A maritime expert may be interested to detect several activity patterns for the monitored vessels, such as sudden changes in the kinematic behavior of a vessel (e.g., sudden accelerations), sailing in protected (e.g., NATURA) areas, etc. The typical workflow consists of the analyst first writing these patterns in some declarative language, which are then used by a computational model applied on the stream of SDEs to detect CEs.

Structure of the Paper The rest of the paper is structured as follows. We start by presenting in Section 2 the relevant literature on CEF. Since work on CEF has been limited thus far, we also briefly mention forecasting ideas from some other related fields that can provide inspiration to CEF. Subsequently, in Section 3 we discuss the formalism of symbolic automata and how it can be adapted to perform recognition on real-time event streams. Section 4 shows how we can create a probabilistic model for a symbolic automaton by using prediction suffix trees, while Section 5 presents a detailed complexity analysis. We then discuss how we can quantify the quality of forecasts in Section 6. We finally

[1] The report may be found here:`https://arxiv.org/abs/2109.00287`

demonstrate the efficacy of our framework in Section 7, by showing experimental results on two application domains. We conclude with Section 8, discussing some possible directions for future work.

2 Related Work

Forecasting has not received much attention in the field of CER, although some conceptual proposals have acknowledged the need for CEF [19,17,13]. However, it has similarities to forecasting methods developed in other fields. We first present a brief overview of these related fields and explain the main differences between them and CEF. We then discuss previous work focused directly on CEF.

Time-series forecasting is an area with some similarities to CEF and a significant history of contributions [28]. However, it is not possible to directly apply techniques from time-series forecasting to CEF. Time-series forecasting typically focuses on streams of (mostly) real-valued variables and the goal is to forecast relatively simple patterns. On the contrary, in CEF we are also interested in categorical values, related through complex patterns and involving multiple variables. Time-series forecasting methods do not provide a language with which we can define complex patterns, but simply try to forecast the next value(s) from the input stream/series. In CER, the equivalent task would be to forecast the next input event(s) (SDEs). This task in itself is not very useful for CER though, since the majority of SDE instances should be ignored and do not contribute to the detection of CEs. CEs are more like "anomalies" and their number is typically orders of magnitude lower than the number of SDEs. One could possibly try to leverage techniques from SDE forecasting to perform CE forecasting. At every timepoint, we could try to estimate the most probable sequence of future SDEs, then perform recognition on this future stream of SDEs and check whether any future CEs are detected. We have experimentally observed that such an approach yields sub-optimal results. It almost always fails to detect any future CEs. This behavior is due to the fact that CEs are rare. As a result, projecting the input stream into the future creates a "path" with high probability but fails to include the rare "paths" that lead to a CE detection.

Another related field is that of prediction of discrete sequences over finite alphabets and is closely related to the field of compression, as any compression algorithm can be used for prediction and vice versa [10,37,36, 14,42]. The main limitation of these methods is that they also do not provide a language for patterns and focus exclusively on next symbol prediction, i.e., they try to forecast the next symbol(s) in a stream/string of discrete symbols. As already discussed, this is a serious limitation for CER. An additional limitation is that they work on single-variable discrete sequences of symbols, whereas CER systems consume streams of events, i.e., streams of tuples with multiple variables, both numerical and categorical.

Forecasting methods have also appeared in the field of temporal pattern mining [40,23,43,12]. From the perspective of CER, the disadvantage of these methods is that they usually target simple patterns, defined either as strict sequences or as sets of input events. Moreover, the input stream is composed of symbols from a finite alphabet, as is the case with the compression methods mentioned above.

Lately, a significant body of work has focused on event sequence prediction and point-of-interest recommendations through the use of neural networks (see, for example, [25,11]). These methods are powerful in predicting the next input event(s) in a sequence of events, but they suffer from limitations already mentioned above. They do not provide a language for defining complex patterns among events and their focus is thus on SDE forecasting. An additional motivation for us to first try a statistical method rather than going directly to neural networks is that, in other related fields, such as time series forecasting, statistical methods have often been proven to be more accurate and less demanding in terms of computational resources than ML ones [26].

Compared to the previous categories for forecasting, the field of process mining is more closely related to CER [38]. An important difference between CER and process mining is that processes are usually given directly as transition systems, whereas CER patterns are defined in a declarative manner. The transition systems defining processes are usually composed of long sequences of events. On the other hand, CER patterns are shorter, may involve Kleene-star, iteration operators (usually not present in processes) and may even be instantaneous. A CEF system cannot always rely on the memory implicitly encoded in a transition system and has to be able to learn the sequences of events that lead to a (possibly instantaneous) CE. Another important difference is that process prediction focuses on traces, which are complete, full matches, whereas CER focuses on continuously evolving streams which may contain many irrelevant events. A learning method has to take into account the presence of these irrelevant events.

We now move on to forecasting methods derived directly from the field of CER which are thus comparable to the one we present here. In what follows, we present previous work on CEF in order of publication date. The first concrete attempt at CEF was pre-

sented in [29]. A variant of regular expressions was used to define CE patterns, which were then compiled into automata. These automata were translated to Markov chains through a direct mapping, where each automaton state was mapped to a Markov chain state. Frequency counters on the transitions were used to estimate the Markov chain's transition matrix. This Markov chain was finally used to estimate if a CE was expected to occur within some future window. In the worst case, however, such an approach assumes that all SDEs are independent (even when the states of the Markov chain are not independent) and is thus unable to encode higher-order dependencies. (see Section 4.2).

Another example of event forecasting was presented in [5]. Using Support Vector Regression, the proposed method was able to predict the next input event(s) within some future window. This technique is similar to time-series forecasting [28], as it mainly targets the prediction of the (numerical) values of the attributes of the input (SDE) events (specifically, traffic speed and intensity from a traffic monitoring system). Strictly speaking, it cannot therefore be considered a CE forecasting method, but a SDE forecasting one. Nevertheless, the authors of [5] proposed the idea that these future SDEs may be used by a CER engine to detect future CEs. Namely, at each timepoint, one could estimate the most probable sequence of future SDEs, then perform recognition on this future stream of SDEs and check whether any future CEs are detected. We have experimentally observed that such an approach fails to detect future CEs. This is due to the fact that CEs are rare. As a result, projecting the input stream into the future creates a "path" with high probability but fails to include the rare "paths" that lead to a CE detection. Because of this serious under-performance of this method, we exclude it from our detailed experiments.

In [31], Hidden Markov Models (HMM) are used to construct a probabilistic model for the behavior of a transition system describing a CE. The observable variable of the HMM corresponds to the states of the transition system, i.e., an observation sequence of length l for the HMM consists of the sequence of states visited by the system after consuming l SDEs. These l SDEs are mapped to the hidden variable, i.e., the last l values of the hidden variable are the last l SDEs. In principle, HMMs are more powerful than Markov chains. In practice, however, HMMs are hard to train ([10,4]) and require elaborate domain modeling, since mapping a CE pattern to a HMM is not straightforward (see Section 4.2 for details). In contrast, our approach constructs seamlessly a probabilistic model from a given CE pattern (declaratively defined).

Automata and Markov chains are again used in [6, 7]. The main difference of these methods compared to [29] is that they can accommodate higher-order dependencies by creating extra states for the automaton of a pattern. The method presented in [6] has two important limitations: first, it works only on discrete sequences of finite alphabets; second, the number of states required to encode long-term dependencies grows exponentially. The first issue was addressed in [7], where symbolic automata are used that can handle infinite alphabets. However, the problem of the exponential growth of the number of states still remains, which can be addressed by using variable-order Markov models.

A different approach is followed in [24], where knowledge graphs are used to encode events and their timing relationships. Stochastic gradient descent is employed to learn the weights of the graph's edges that determine how important an event is with respect to another target event. However, this approach falls in the category of SDE forecasting, as it does not target complex events. More precisely, it tries to forecast which predicates the forthcoming SDEs will satisfy, without taking into account relationships between the events themselves (e.g., through simple sequences).

3 Complex Event Recognition with Symbolic Automata

Our approach for CEF is based on a specific formal framework for CER, which we are presenting here. There are various surveys of CER methods, presenting various CER systems and languages [15,8,20]. Despite this fact though, there is still no consensus about which operators must be supported by a CER language and what their semantics should be. In this paper, we follow [20] and [21], which have established some core operators that are most often used. In a spirit similar to [21], we use automata as our computational model and define a CER language whose expressions can readily be converted to automata. We employ symbolic regular expressions and automata [16,39]. The rationale behind our choice is that, contrary to other automata-based CER models, symbolic regular expressions and automata have nice closure properties and clear (both declarative and operational), compositional semantics (see [21] for a similar line of work, based on symbolic transducers). In previous automata-based CER systems, it is unclear which operators may be used and if they can be arbitrarily combined (see [21,20] for a discussion of this issue). On the contrary, the use of symbolic automata allows us to construct any pattern that one may desire through an arbitrary use of the provided operators. In previous methods, there is also a

lack of understanding with respect to the properties of the employed computational models, e.g., whether the proposed automata are determinizable, an important feature for our work. Symbolic automata, on the other hand, have nice closure properties and are well-studied. Notice that this would also be an important feature for possible optimizations based on pattern re-writing, since such re-writing would require us to have a mechanism determining whether two expressions are equivalent. Our framework provides such a mechanism.

The main idea behind symbolic automata is that each transition, instead of being labeled with a symbol from an alphabet, is equipped with a unary formula from an effective Boolean algebra [16]. A symbolic automaton can then read strings of elements and, upon reading an element while in a given state, can apply the predicates of this state's outgoing transitions to that element. The transitions whose predicates evaluate to TRUE are said to be "enabled" and the automaton moves to their target states.

The formal definition of an effective Boolean algebra is the following:

Definition 1 (Effective Boolean algebra [16]) An effective Boolean algebra is a tuple $(\mathcal{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ where $\mathcal{D}$ is a set of domain elements, Ψ a set of predicates and $[\![\_]\!] : \Psi \rightarrow 2^{\mathcal{D}}$ a function mapping predicates to the powerset of $\mathcal{D}$. We assume that $\bot, \top \in \Psi$, i.e., $\bot$ and $\top$ are predicates that are always available and correspond to FALSE and TRUE respectively. In addition, the predicates in Ψ are closed under the Boolean connectives, i.e., any predicate that can be constructed from other predicates in Ψ, using the connectives of conjunction $\wedge$, disjunction $\vee$ and negation $\neg$, also belongs to Ψ. Finally, the function $[\![\_]\!]$ allows us to determine which elements from $\mathcal{D}$ satisfy which predicates according to the following rules:

- $[\![\bot]\!] = \emptyset$
- $[\![\top]\!] = \mathcal{D}$
- and $\forall \phi, \psi \in \Psi$:
  - $[\![\phi \vee \psi]\!] = [\![\phi]\!] \cup [\![\psi]\!]$
  - $[\![\phi \wedge \psi]\!] = [\![\phi]\!] \cap [\![\psi]\!]$
  - $[\![\neg\phi]\!] = \mathcal{D} \setminus [\![\phi]\!]$

It is also required that checking satisfiability of ϕ, i.e., whether $[\![\phi]\!] \neq \emptyset$, is decidable and that the operations of $\vee$, $\wedge$ and $\neg$ are computable. ◄

Using our running example, such an algebra could be one consisting of AIS messages, corresponding to $\mathcal{D}$, along with two predicates about the speed of a vessel, e.g., *speed* < 5 and *speed* > 20. These two predicates would make up Ψ. The predicate *speed* < 5 would be mapped, via $[\![\_]\!]$, to the set of all AIS messages whose speed level is below 5 knots. According to the definition above, $\bot$ and $\top$ should also belong to Ψ, along with all the combinations of the original two predicates constructed from the Boolean connectives, e.g., $\neg(\textit{speed} < 5) \wedge \neg(\textit{speed} > 20)$. Elements of $\mathcal{D}$ are called *characters* and finite sequences of characters are called *strings*. A set of strings $\mathcal{L}$ constructed from elements of $\mathcal{D}$ ($\mathcal{L} \subseteq \mathcal{D}^*$, where * denotes Kleene-star) is called a language over $\mathcal{D}$.

As with classical regular expressions [22], we can use symbolic regular expressions to represent a class of languages over $\mathcal{D}$.

Definition 2 (Symbolic regular expression) A symbolic regular expression (*SRE*) over an effective Boolean algebra $(\mathcal{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ is recursively defined as follows:

- The constants ϵ and $\emptyset$ are symbolic regular expressions with $\mathcal{L}(\epsilon) = \{\epsilon\}$ and $\mathcal{L}(\emptyset) = \{\emptyset\}$;
- If $\psi \in \Psi$, then $R := \psi$ is a symbolic regular expression, with $\mathcal{L}(\psi) = [\![\psi]\!]$, i.e., the language of ψ is the subset of $\mathcal{D}$ for which ψ evaluates to TRUE;
- Disjunction / Union: If R_1 and R_2 are symbolic regular expressions, then $R := R_1 + R_2$ is also a symbolic regular expression, with $\mathcal{L}(R) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$;
- Concatenation / Sequence: If R_1 and R_2 are symbolic regular expressions, then $R := R_1 \cdot R_2$ is also a symbolic regular expression, with $\mathcal{L}(R) = \mathcal{L}(R_1) \cdot \mathcal{L}(R_2)$, where $\cdot$ denotes concatenation. $\mathcal{L}(R)$ is then the set of all strings constructed from concatenating each element of $\mathcal{L}(R_1)$ with each element of $\mathcal{L}(R_2)$;
- Iteration / Kleene-star: If R is a symbolic regular expression, then $R' := R^*$ is a symbolic regular expression, with $\mathcal{L}(R^*) = (\mathcal{L}(R))^*$, where $\mathcal{L}^* = \bigcup_{i \geq 0} \mathcal{L}^i$ and $\mathcal{L}^i$ is the concatenation of $\mathcal{L}$ with itself i times.

◄

As an example, if we want to detect instances of a vessel accelerating suddenly, we could write the expression $R := (\textit{speed} < 5) \cdot (\textit{speed} > 20)$. Note that, with the help of the above basic operators, it is possible to define various other operators, like *complement / negation* (symbolic automata are also closed under complement), *quantifiers / bounded iteration*, *conjunction* (if $\otimes$ denotes conjunction, then $R := R_1 \otimes R_2 := (R_1 \cdot R_2) + (R_2 \cdot R_1)$), as well as *selection policies* [20]. Due to space limitations, we do not elaborate further on these operators here.

Given a Boolean algebra, we can also define symbolic automata. The definition of a symbolic automaton is the following:

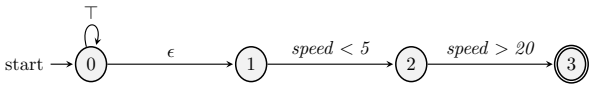


Fig. 1: Streaming *SFA* for $R := (speed < 5) \cdot (speed > 20)$. $\top$ is a special predicate that always evaluates to TRUE. $\top$ transitions are thus triggered for every event. ϵ transitions triggered even in the absence of an event.

Definition 3 (Symbolic finite automaton [16]) A symbolic finite automaton (*SFA*) is a tuple $M = (\mathcal{A}, Q, q^s, Q^f, \Delta)$, where $\mathcal{A}$ is an effective Boolean algebra; Q is a finite set of states; $q^s \in Q$ is the initial state; $Q^f \subseteq Q$ is the set of final states; $\Delta \subseteq Q \times \Psi_{\mathcal{A}} \times Q$ is a finite set of transitions. ◄

A string $w = a_1 a_2 \cdots a_k$ is accepted by a *SFA* M iff, for $1 \leq i \leq k$, there exist transitions $q_{i-1} \xrightarrow{a_i} q_i$ such that $q_0 = q^s$ and $q_k \in Q^f$. The set of strings accepted by M is the language of M, denoted by $\mathcal{L}(M)$ [16].

As with classical regular expressions and automata, we can prove that every symbolic regular expression can be translated to an equivalent (i.e., with the same language) symbolic automaton.

Proposition 1 *For every symbolic regular expression R there exists a symbolic finite automaton M such that $\mathcal{L}(R) = \mathcal{L}(M)$.*

Proof. Proof presented in the technical report. □

Our discussion thus far has focused on how *SRE* and *SFA* can be applied to bounded strings that are known in their totality before recognition. We feed a string to a *SFA* and we expect an answer about whether the whole string belongs to the automaton's language or not. However, in CER and CEF we need to handle continuously updated streams of events and detect instances of *SRE* satisfaction as soon as they appear in a stream. In order to accommodate this scenario, slight modifications are required so that *SRE* and *SFA* may work in a streaming setting. First, we need to make sure that the automaton can start its recognition after every new element. In our case, events come in the form of tuples with both numerical and categorical values. Using database systems terminology we can speak of tuples from relations of a database schema [21]. These tuples constitute the set of domain elements $\mathcal{D}$. A stream S then has the form of an infinite sequence $S = t_1, t_2, \cdots$, where each t_i is a tuple ($t_i \in \mathcal{D}$). Our goal is to report the indices i at which a CE is detected.

More precisely, if $S_{1..k} = \cdots, t_{k-1}, t_k$ is the prefix of S up to the index k, we say that an instance of a *SRE* R is detected at k iff there exists a suffix $S_{m..k}$ of $S_{1..k}$ such that $S_{m..k} \in \mathcal{L}(R)$. In order to detect CEs of a *SRE* R on a stream, we use a streaming version of *SRE* and *SFA*. If R is a *SRE*, then $R_s = \top^* \cdot R$ is the streaming *SRE* (*sSRE*) corresponding to R, and the automaton for R_s is the streaming *SFA* (*sSFA*) of R. Using R_s we can detect CEs of R on a stream S, since a stream segment $S_{m..k}$ belongs to the language of R iff the prefix $S_{1..k}$ belongs to the language of R_s. The prefix $\top^*$ lets us skip any number of events from the stream and start recognition at any index $m, 1 \leq m \leq k$. As an example, if $R := (speed < 5) \cdot (speed > 20)$ is the pattern for sudden acceleration, then its *sSRE* would be $R_s := \top^* \cdot (speed < 5) \cdot (speed > 20)$. Note that *sSRE* and *sSFA* are just special cases of *SRE* and *SFA* respectively. Therefore, every result that holds for *SRE* and *SFA* also holds for *sSRE* and *sSFA*. Figure 1 shows an example *sSFA*.

The streaming behavior of a *sSFA* as it consumes a stream S can be formally defined using the notion of configuration:

Definition 4 (Configuration of sSFA) Assume $S = t_1, t_2, \cdots$ is a stream of domain elements from an effective Boolean algebra, R a symbolic regular expression over the same algebra and M_{R_s} a *sSFA* corresponding to R. A configuration c of M_{R_s} is a tuple $[i, q]$, where i is the current position of the stream, i.e., the index of the next event to be consumed, and q the current state of M_{R_s}. We say that $c' = [i', q']$ is a successor configuration of c iff:

- $\exists \delta \in M_{R_s}.\Delta : \delta = (q, \psi, q') \wedge (t_i \in [\![\psi]\!] \vee \psi = \epsilon)$;
- $i = i'$ if $\delta = \epsilon$. Otherwise, $i' = i + 1$.

We denote a succession by $[i, q] \xrightarrow{\delta} [i', q']$. ◄

For the initial configuration c^s, before consuming any events, we have that $i = 1$ and $c^s.q = M_{R_s}.q^s$, i.e. the state of the first configuration is the initial state of M_{R_s}. In other words, for every index i, we move from our current state q to another state q' if there is an outgoing transition from q to q' and the predicate on this transition evaluates to TRUE for t_i. We then increase the reading position by 1. Alternatively, if the transition is an ϵ-transition, we move to q' without increasing the reading position.

The actual behavior of a *sSFA* upon reading a stream is captured by the notion of the run:

Definition 5 (Run of sSFA over stream) A run ϱ of a *sSFA* M over a stream $S_{1..k}$ is a sequence of successor configurations $[1, q_1 = M.q^s] \xrightarrow{\delta_1} [2, q_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_k} [k+1, q_{k+1}]$. ϱ is called accepting iff $q_{k+1} \in M.Q^f$. ◄

A run ϱ of a *sSFA* M_{R_s} over a stream $S_{1..k}$ is accepting iff $S_{1..k} \in \mathcal{L}(R_s)$, since M_{R_s}, after reading $S_{1..k}$,

must have reached a final state. For a *sSFA* reading a stream, the existence of an accepting run with configuration index $k+1$ implies that a CE for the *SRE* R has been detected at the stream index k.

As far as the temporal model is concerned, we assume that all SDEs are instantaneous. They all carry a *timestamp* attribute which is single, unique numerical value. We also assume that the stream of SDEs is temporally sorted. A sequence/concatenation operator is thus satisfied if the event of its first operand precedes in time the event of its second operand. Another general assumption is that there is no imposed limit on the time elapsed between consecutive events in a sequence operation.

4 Building a Probabilistic Model

The main idea behind our forecasting method is the following: Given a pattern R in the form of a *SRE*, we first construct a *sSFA* as described in the previous section. For event recognition, this would already be enough, but in order to perform event forecasting, we translate the *sSFA* to an equivalent deterministic *SFA* (*DSFA*). This *DSFA* can then be used to learn a probabilistic model, typically a Markov chain, that encodes dependencies among the events in an input stream. Note that a non-deterministic automaton cannot be directly converted to a Markov chain, since from each state we might be able to move to multiple other target states with a given event. Therefore, we first determinize the automaton. The probabilistic model is learned from a portion of the input stream which acts as a training dataset and it is then used to derive forecasts about the expected occurrence of the CE encoded by the automaton. The issue that we address in this paper is how to build a model which retains long-term dependencies that are useful for forecasting.

4.1 Deterministic Symbolic Automata

The definition of *DSFA* is similar to that of classical deterministic automata. Intuitively, we require that, for every state and every tuple/character, the *SFA* can move to at most one next state upon reading that tuple/character. We note though that it is not enough to require that all outgoing transitions from a state have different predicates as guards. Symbolic automata differ from classical in one important aspect. For the latter, if we start from a given state and we have two outgoing transitions with different labels, then it is not possible for both of these transition to be triggered simultaneously (i.e., with the same character). For symbolic automata, on the other hand, two predicates may be different but still both evaluate to TRUE for the same tuple and thus two transitions with different predicates may both be triggered with the same tuple. Therefore, the formal definition for *DSFA* must take this into account:

Definition 6 (Deterministic SFA [16]) A *SFA* M is deterministic if, for all transitions $(q, \psi_1, q_1), (q, \psi_2, q_2) \in M.\Delta$, if $q_1 \neq q_2$ then $[\![\psi_1 \wedge \psi_2]\!] = \emptyset$. ◄

Using this definition for *DSFA* it can be proven that *SFA* are indeed closed under determinization [16]. The determinization process first needs to create the *minterms* of the predicates of a *SFA* M, i.e., the set of maximal satisfiable Boolean combinations of such predicates, denoted by $N = \mathit{Minterms}(\mathit{Predicates}(M))$, and then use these minterms as guards for the *DSFA* [16].

Before moving to the discussion about how a *DSFA* can be converted to a Markov chain, we present a useful lemma. We will show that a *DSFA* always has an equivalent (through an isomorphism) deterministic classical automaton. This result is important because: a) it allows us to use methods developed for classical automata without having to always prove that they are indeed applicable to symbolic automata as well, and b) it will help us in simplifying our notation, since we can use the standard notation of symbols instead of predicates.

First note that the set of minterms N induces a finite set of equivalence classes on the (possibly infinite) set of domain elements of M [16]. For example, if $\mathit{Predicates}(M) = \{\psi_1, \psi_2\}$, then $N = \{\psi_1 \wedge \psi_2, \psi_1 \wedge \neg\psi_2, \neg\psi_1 \wedge \psi_2, \neg\psi_1 \wedge \neg\psi_2\}$, and we can map each domain element (in our case, a tuple) to exactly one of these 4 minterms: the one that evaluates to TRUE when applied to the element. Similarly, the set of minterms induces a set of equivalence classes on the set of strings (event streams in our case). For example, if $S{=}t_1, \cdots, t_k$ is an event stream, then it could be mapped to $S'{=}a, \cdots, b$, with a corresponding to $\psi_1 \wedge \neg\psi_2$ if $\psi_1(t_1) \wedge \neg\psi_2(t_1) =$ TRUE, b to $\psi_1 \wedge \psi_2$, etc.

Definition 7 (Stream induced by the minterms of a *DSFA*) Let $N = \mathit{Minterms}(\mathit{Predicates}(M))$. If S is a stream from the domain elements of the algebra of a *DSFA* M, then the stream S' induced by applying N on S is the equivalence class of S induced by N. ◄

We can now present the lemma:

Lemma 1 *For every DSFA M_s there exists a deterministic classical finite automaton (DFA) M_c such that $\mathcal{L}(M_c)$ is the set of strings induced by applying $N = \mathit{Minterms}(\mathit{Predicates}(M_s))$ to $\mathcal{L}(M_s)$.*

Proof. Proof presented in the technical report. □

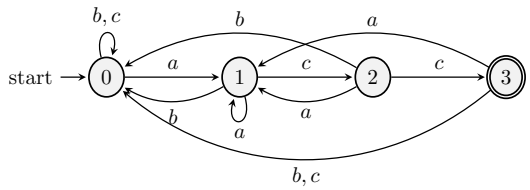


Fig. 2: A classical automaton for the expression $R := a \cdot c \cdot c$ with alphabet $\Sigma = \{a, b, c\}$. State 1 can always remember the last symbol seen, since it can be reached only with a. State 0 can be reached with b or c.

Henceforth, we will be using symbols and strings as in classical theories of automata and strings (simple lowercase letters to denote symbols), but the reader should bear in mind that, in our case, each symbol always corresponds to a predicate and, more precisely, to a minterm of a *DSFA*.

4.2 Variable-order Markov Models

Assuming that we have a deterministic automaton, the next question is how we can build a probabilistic model that captures the statistical properties of the streams to be processed by this automaton. With such a model, we could then make inferences about the automaton's expected behavior as it reads event streams. One approach would be to map each state of the automaton to a state of a Markov chain, then apply the automaton on a training stream of symbols, count the number of transitions from each state to every other target state and use these counts to calculate the transition probabilities. This is the approach followed in [29]. However, there is an important issue with the way in which this approach models transition probabilities. Namely, a probability is attached to the transition between two states, say state 1 and state 2, ignoring the way in which state 1 has been reached, i.e., failing to capture the sequence of symbols. For example, in Figure 2, state 0 can be reached after observing symbol b or symbol c. The outgoing transition probabilities do not distinguish between the two cases. Instead, they just capture the probability of a given that the previous symbol was b or c. This introduces ambiguity and if there are many such states in the automaton, we may end up with a Markov chain that is first-order (with respect to its states), but nevertheless provides no memory of the stream itself. It may be unable to capture first-order (or higher order) dependencies in the stream of events. In the worst case (if every state can be reached with any symbol), such a Markov chain may essentially assume that the stream is composed of i.i.d. events.

An alternative approach, followed in [7,6], is to first set a maximum order m that we need to capture and then iteratively split each state of the original automaton into as many states as required so that each new state can remember the past m symbols that have led to it. The new automaton that results from this splitting process is equivalent to the original, in the sense that they recognize the same language, but can always remember the last m symbols of the stream. With this approach, it is indeed possible to guarantee that m-order dependencies can be captured. As expected though, higher values of m can quickly lead to an exponential growth of the number of states and the approach may be practical only for low values of m.

We propose the use of a variable-order Markov model (VMM) to mitigate the high cost of increasing the order m [10,37,36,14,42]. This allows us to increase m to values not possible with the previous approaches and thus capture longer-term dependencies, which can lead to a better accuracy. An alternative would be to use hidden Markov models (HMMs) [34], which are generally more expressive than bounded-order (either full or variable) Markov models. However, HMMs often require large training datasets [10,4]. Another problem is that it is not always obvious how a domain can be modeled through HMMs and a deep understanding of the domain may be required [10]. Consider, for example, our case of automata-based CER. The relation between an automaton and the observed state of a HMM is not straightforward and it is not evident how a HMM would capture an automaton's behavior.

Different Markov models of variable order have been proposed in the literature (see [10] for a nice comparative study). The general approach of such models is as follows: let Σ denote an alphabet, $\sigma \in \Sigma$ a symbol from that alphabet and $s \in \Sigma^m$ a string of length m of symbols from that alphabet. The aim is to derive a predictor $\hat{P}$ from the training data such that the average log-loss on a test sequence $S_{1..k}$ is minimized. The loss is given by $l(\hat{P}, S_{1..k}) = -\frac{1}{T}\sum_{i=1}^{k} log\hat{P}(t_i \mid t_1 \cdots t_{i-1})$. Minimizing the log-loss is equivalent to maximizing the likelihood $\hat{P}(S_{1..k}) = \prod_{i=1}^{k} \hat{P}(t_i \mid t_1 \ldots t_{i-1})$. The average log-loss may also be viewed as a measure of the average compression rate achieved on the test sequence [10]. The mean (or expected) log-loss ($-\boldsymbol{E}_P\{log\hat{P}(S_{1..k})\}$) is minimized if the derived predictor $\hat{P}$ is indeed the actual distribution P of the source emitting sequences.

For full-order Markov models, the predictor $\hat{P}$ is derived through the estimation of conditional distributions $\hat{P}(\sigma \mid s)$, with m constant and equal to the assumed order of the Markov model. On the other hand, variable-order Markov Models (VMMs) relax the assumption of m being fixed. The length of the "con-

text" s (as is usually called) may vary, up to a *maximum* order m, according to the statistics of the training dataset. By looking deeper into the past only when it is statistically meaningful, VMMs can capture both short- and long-term dependencies.

4.3 Prediction Suffix Trees

We use Prediction Suffix Trees (*PST*), as described in [37,36], as our VMM of choice. The reason is that, once a *PST* has been learned, it can be readily converted to a probabilistic automaton. More precisely, we learn a probabilistic suffix automaton (*PSA*), whose states correspond to contexts of variable length. The outgoing transitions from each state of the *PSA* encode the conditional distribution of seeing a symbol given the context of that state. As we will show, this probabilistic automaton (or the tree itself) can then be combined with a symbolic automaton in a way that allows us to infer when a CE is expected to occur.

The formal definition of a PST is the following:

Definition 8 (Prediction Suffix Tree [37]) Let Σ be an alphabet. A PST T over Σ is a tree whose edges are labeled by symbols $\sigma \in \Sigma$ and each internal node has exactly one edge for every $\sigma \in \Sigma$ (hence, the degree is $\mid \Sigma \mid$). Each node is labeled by a pair (s, γ_s), where s is the string associated with the walk starting from that node and ending at the root, and $\gamma_s : \Sigma \rightarrow [0,1]$ is the next symbol probability function related with s. For every string s labeling a node, $\sum_{\sigma \in \Sigma} \gamma_s(\sigma) = 1$. The depth of the tree is its order m. ◀

Figure 3a shows an example of a *PST* of order $m = 2$. According to this tree, if the last symbol that we have encountered in a stream is a and we ignore any other symbols that may have preceded it, then the probability of the next input symbol being again a is 0.7. However, we can obtain a better estimate of the next symbol probability by extending the context and looking one more symbol deeper into the past. Thus, if the last two symbols encountered are b, a, then the probability of seeing a again is very different (0.1). On the other hand, if the last symbol encountered is b, the next symbol probability distribution is $(0.5, 0.5)$ and, since the node $b, (0.5, 0.5)$ has not been expanded, this implies that its children would have the same distribution if they had been created. Therefore, the past does not affect the prediction and will not be used. A *PST* whose leaves are all of equal depth m corresponds to a full-order Markov model of order m, as its paths from the root to the leaves correspond to every possible context of length m.

Our goal is to incrementally learn a *PST* $\hat{T}$ by adding new nodes only when it is necessary and then use $\hat{T}$ to construct a *PSA* $\hat{M}$ that will approximate the actual *PSA* M that has generated the training data. Assuming that we have derived an initial predictor $\hat{P}$ (as described in more detail in Section 4.5), the learning algorithm in [37] starts with a tree having only a single node, corresponding to the empty string ϵ. Then, it decides whether to add a new context/node s by checking two conditions:

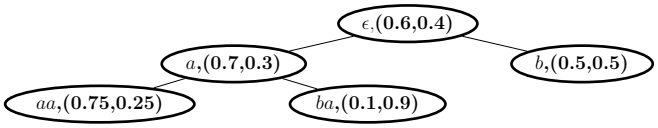
- First, there must exist $\sigma \in \Sigma$ such that $\hat{P}(\sigma \mid s) > \theta_1$ must hold, i.e., σ must appear "often enough" after the suffix s;
- Second, $\frac{\hat{P}(\sigma|s)}{\hat{P}(\sigma|suffix(s))} > \theta_2$ (or $\frac{\hat{P}(\sigma|s)}{\hat{P}(\sigma|suffix(s))} < \frac{1}{\theta_2}$) must hold, i.e., it is "meaningful enough" to expand to s because there is a significant difference in the conditional probability of σ given s with respect to the same probability given the shorter context $suffix(s)$, where $suffix(s)$ is the longest suffix of s that is different from s.

The thresholds θ_1 and θ_2 depend, among others, on parameters α, n and m, α being an approximation parameter, measuring how close we want the estimated *PSA* $\hat{M}$ to be compared to the actual *PSA* M, n denoting the maximum number of states that we allow $\hat{M}$ to have and m denoting the maximum order/length of the dependencies we want to capture. For example, consider node a in Figure 3a and assume that we are at a stage of the learning process where we have not yet added its children, aa and ba. We now want to check whether it is meaningful to add ba as a node. Assuming that the first condition is satisfied, we can then check the ratio $\frac{\hat{P}(\sigma|s)}{\hat{P}(\sigma|suffix(s))} = \frac{\hat{P}(a|ba)}{\hat{P}(a|a)} = \frac{0.1}{0.7} \approx 0.14$. If $\theta_2 = 1.05$, then $\frac{1}{\theta_2} \approx 0.95$ and the condition is satisfied, leading to the addition of node ba to the tree [37].

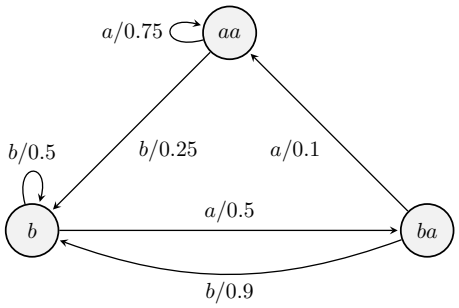
Once a *PST* $\hat{T}$ has been learned, we can convert it to a *PSA* $\hat{M}$. The definition for *PSA* is the following:

Definition 9 (Probabilistic Suffix Automaton [37]) A Probabilistic Suffix Automaton M is a tuple $(Q, \Sigma, \tau, \gamma, \pi)$, where Q is a finite set of states; Σ is a finite alphabet; $\tau : Q \times \Sigma \rightarrow Q$ is the transition function; $\gamma : Q \times \Sigma \rightarrow [0,1]$ is the next symbol probability function; $\pi : Q \rightarrow [0,1]$ is the initial probability distribution over the starting states. The following conditions must hold:

- For every $q \in Q$, it must hold that $\sum_{\sigma \in \Sigma} \gamma(q, \sigma) = 1$ and $\sum_{q \in Q} \pi(q) = 1$;
- Each $q \in Q$ is labeled by a string $s \in \Sigma^*$ and the set of labels is suffix free, i.e., no label s is a suffix of another label s';



(a) Example *PST* T for $\Sigma = \{a, b\}$ and $m = 2$. Each node contains the label and the next symbol probability distribution for a and b.



(b) Example *PSA* M_S constructed from the above tree. Each state contains its label. Each transition is composed of the next symbol to be encountered along with that symbol's probability.

Fig. 3: Example of a prediction suffix tree and its corresponding probabilistic suffix automaton.

- For every two states $q_1, q_2 \in Q$ and for every symbol $\sigma \in \Sigma$, if $\tau(q_1, \sigma) = q_2$ and q_1 is labeled by s_1, then q_2 is labeled by s_2, such that s_2 is a suffix of $s_1 \cdot \sigma$;
- For every s labeling some state q, and every symbol σ for which $\gamma(q, \sigma) > 0$, there exists a label which is a suffix of $s \cdot \sigma$;
- Finally, the graph of M is strongly connected.

◀

Note that a *PSA* is a Markov chain. τ and γ can be combined into a single function, ignoring the symbols, and this function, together with the first condition of Definition 9, would define the transition matrix of a Markov chain. The last condition about M being strongly connected also ensures that the Markov chain is composed of a single recurrent class of states. Figure 3b shows an example of a *PSA*, the one that we construct from the *PST* of Figure 3a, using the leaves of the tree as automaton states. A full-order *PSA* for $m = 2$ would require a total of 4 states, given that we have two symbols. If we use the *PST* of Figure 3a, we can construct the *PSA* of Figure 3b which has 3 states. State b does not need to be expanded to states bb and ab, since the tree tells us that such an expansion is not statistically meaningful.

Using a *PSA* we can process a stream of symbols and at every point be able to provide an estimate about the next symbols that will be encountered along with their probabilities. The state of the *PSA* at every moment corresponds to a suffix of the stream. For example, according to the *PSA* of Figure 3b, if the last symbol consumed from the stream is b, then the *PSA* would be in state b and the probability of the next symbol being a would be 0.5. If the last symbol in the stream is a, we would need to expand this suffix to look at one more symbol in the past. If the last two symbols are aa, then the *PSA* would be in state aa and the probability of the next symbol being a again would be 0.75.

The above discussion seems to suggest that a *PSA* is constructed from the leaves of a *PST*. Thus, it should be expected that the number of states of a *PSA* should always be smaller than the total number of nodes of its *PST*. However, this is not true in the general case. In fact, in some cases the *PST* nodes might be significantly less than the *PSA* states. The reason is that a *PST*, as is produced by the learning algorithm described previously, might not be sufficient to construct a *PSA*. To remedy this situation, we need to expand the original *PST* $\hat{T}$ by adding more nodes in order to get a suitable *PST* $\hat{T}'$ and then construct the *PSA* from $\hat{T}'$. The leaves of $\hat{T}'$ (and thus the states of the *PSA*) could be significantly more than the leaves of $\hat{T}$. This issue is further discussed in Section 4.4.2.

4.4 Emitting Forecasts

Our ultimate goal is to use the statistical properties of a stream, as encoded in a *PST* or a *PSA*, in order to infer when a Complex Event (CE) with a given Symbolic Regular Expression (*SRE*) R will be detected. Equivalently, we are interested in inferring when the *SFA* of R will reach one of its final states. To achieve this goal, we work as follows. We start with a *SRE* R and a training stream S. We first use R to construct an equivalent *sSFA* and then determinize this *sSFA* into a *DSFA* M_R. M_R can be used to perform recognition on any given stream, but cannot be used for probabilistic inference. Next, we use the minterms of M_R (acting as "symbols", see Lemma 1) and the training stream S to learn a *PST* T and (if required) a *PSA* M_S which encode the statistical properties of S. These probabilistic models do not yet have any knowledge of the structure of R (they only know its minterms), are not acceptors (the *PSA* does not have any final states) and cannot be used for recognition. We therefore need to combine the learned probabilistic model (T or M_S) with the automaton used for recognition (M_R).

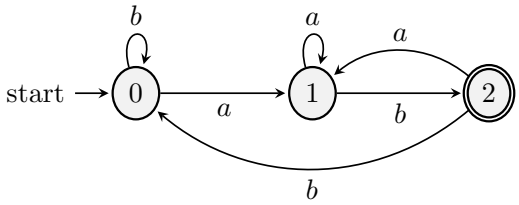
At this point, there is a trade-off between memory and computation efficiency. If the online performance of our system is critical and we are not willing to make significant sacrifices in terms of computation

efficiency, then we should combine the recognition automaton M_R with the *PSA* M_S. Using the *PSA* we can have a very efficient solution with minimal overhead on throughput. The downside of this approach is its memory footprint, which may limit the order of the model. Although we may increase the order beyond what is possible with full-order models, we may still not achieve the desired values, due to the significant memory requirements. Hence, if high accuracy and thus high order values are necessary, then we should combine the recognition automaton M_R directly with the *PST* T, bypassing the construction of the *PSA*. In practice prediction suffix trees often turn out to be more compact and memory efficient than probabilistic suffix automata, but trees need to be constantly traversed from root to leaves whereas an automaton simply needs to find the triggered transition and immediately jump to the next state. In the remainder of this Section, we present these two alternatives.

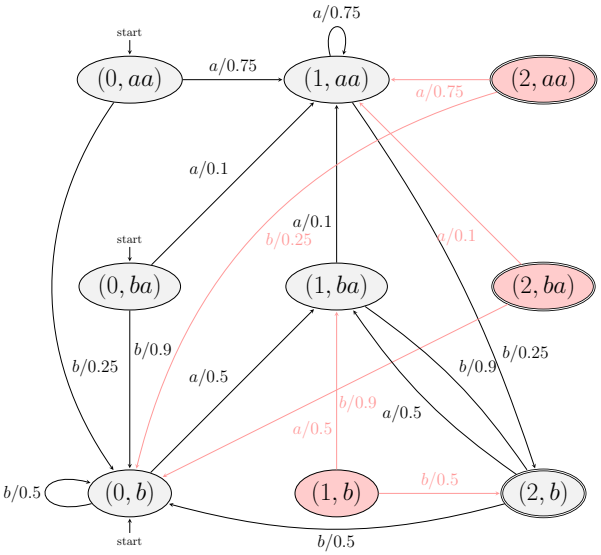
4.4.1 Using a Probabilistic Suffix Automaton (PSA)

We can combine a recognition automaton M_R and a *PSA* M_S into a single automaton M that has the power of both and can be used for recognizing and for forecasting occurrences of CEs of the expression R. We call M the *embedding* of M_S in M_R. The reason for merging the two automata is that we need to know at every point in time the state of M_R in order to estimate which future paths might actually lead to a final state (and thus a complex event). If only SDE forecasting was required, this merging would not be necessary. We could use M_R for recognition and then M_S for SDE forecasting. In our case, we need information about the structure of the pattern automaton and its current state to determine if and when it might reach a final state. The formal definition of an embedding is given below, where, in order to simplify notation, we use Lemma 1 and represent *DSFA* as classical deterministic automata.

Definition 10 (Embedding of a *PSA* in a *DSFA*) Let M_R be a *DSFA* (actually its mapping to a classical automaton) and M_S a *PSA* with the same alphabet. An embedding of M_S in M_R is a tuple $M = (Q, Q^s, Q^f, \Sigma, \Delta, \Gamma, \pi)$, where Q is a finite set of states; $Q^s \subseteq Q$ is the set of initial states; $Q^f \subseteq Q$ is the set of final states; Σ is a finite alphabet; $\Delta : Q \times \Sigma \to Q$ is the transition function; $\Gamma : Q \times \Sigma \to [0,1]$ is the next symbol probability function; $\pi : Q \to [0,1]$ is the initial probability distribution. The language $\mathcal{L}(M)$ of M is defined, as usual, as the set of strings that lead M to a final state. The following conditions must hold, in order for M to be an embedding of M_S in M_R: a) $\Sigma = M_R.\Sigma = M_S.\Sigma$; b) $\mathcal{L}(M) = \mathcal{L}(M_R)$; c) For every string/stream $S_{1..k}$, $P_M(S_{1..k}) = P_{M_S}(S_{1..k})$, where P_M denotes the probability of a string calculated by M (through Γ) and P_{M_S} the probability calculated by M_S (through γ). ◀



(a) *DSFA* M_R for $R := a \cdot b$ and $\Sigma = \{a, b\}$.



(b) Embedding of M_S of Figure 3b in M_R of Figure 4a.

Fig. 4: Embedding example.

The first condition ensures that all automata have the same alphabet. The second ensures that M is equivalent to M_R by having the same language. The third ensures that M is also equivalent to M_S, since both automata return the same probability for every string.

It can be shown that such an equivalent embedding can indeed be constructed for every *DSFA* and *PSA*.

Theorem 1 *For every DSFA M_R and PSA M_S constructed using the minterms of M_R, there exists an embedding of M_S in M_R.*

Proof. Proof presented in the technical report. It is a constructive proof where we take the Cartesian product of the states of M_R and the states of M_S and set the transition and probability functions accordingly. □

As an example, consider the *DSFA* M_R of Figure 4a for the expression $R = a \cdot b$ with $\Sigma = \{a, b\}$. We present it as a classical automaton, but we remind readers that

symbols in Σ correspond to minterms. Figure 3a depicts a possible *PST* T that could be learned from a training stream composed of symbols from Σ. Figure 3b shows the *PSA* M_S constructed from T. Figure 4b shows the embedding M of M_S in M_R. Notice, however, that this embedding has some redundant states and transitions; namely the states indicated with red that have no incoming transitions and are thus inaccessible. The reason is that some states of M_R in Figure 4a have a "memory" imbued to them from the structure of the automaton itself. For example, state 2 of M_R has only a single incoming transition with b as its symbol. Therefore, there is no point in merging this state with all the states of M_S, but only with state b. To avoid the inclusion of red states, we can merge M_R and M_S in an incremental fashion. The resulting automaton would then consist only of the black states and transitions of Figure 4b. Notice that this automaton has multiple start states. In a streaming setting, we would thus have to wait at the beginning of the stream for some input events to arrive before deciding the start state with which to begin. For example, if b were the first input event, we would then begin with the bottom left state $(0, b)$. On the other hand, if a were the first input event, we would have to wait for yet another event. If another a arrived as the second event, we would begin with the top left state $(0, aa)$. In general, if m is our maximum order, we would need to wait for at most m input events before deciding.

After constructing an embedding M from a *DSFA* M_R and a *PSA* M_S, we can use M to perform forecasting on a test stream. Since M is equivalent to M_R, it can also consume a stream and detect the same instances of the expression R as M_R would detect. However, our goal is to use M to forecast the detection of an instance of R. More precisely, we want to estimate the number of transitions from any state in which M might be until it reaches for the first time one of its final states. Towards this goal, we can use the theory of Markov chains. Let N denote the set of non-final states of M and F the set of its final states. We can organize the transition matrix of M in the following way (we use bold symbols to refer to matrices and vectors and normal ones to refer to scalars or sets):

$$\boldsymbol{\Pi} = \begin{pmatrix} \boldsymbol{N} & \boldsymbol{N_F} \\ \boldsymbol{F_N} & \boldsymbol{F} \end{pmatrix} \tag{1}$$

where $\boldsymbol{N}$ is the sub-matrix containing the probabilities of transitions from non-final to non-final states, $\boldsymbol{F}$ the probabilities from final to final states, $\boldsymbol{F_N}$ the probabilities from final to non-final states and $\boldsymbol{N_F}$ the probabilities from non-final to final states. By partitioning the states of a Markov chain into two sets, such as N and F, the following theorem can be used to estimate the probability of reaching a state in F:

Theorem 2 *Let $\boldsymbol{\Pi}$ be the transition probability matrix of a homogeneous Markov chain Y_t in the form of Equation (1) and $\boldsymbol{\xi}_{init}$ its initial state distribution. The probability for the time index n when the system first enters the set of states F, starting from a state in N, can be obtained from*

$$P(Y_n \in F, Y_{n-1} \in N, \cdots, Y_2 \in N, Y_1 \in N \mid \boldsymbol{\xi_{init}}) = \boldsymbol{\xi_N}^T \boldsymbol{N}^{n-1}(\boldsymbol{I} - \boldsymbol{N})\mathbf{1} \tag{2}$$

where ξ_N is the vector consisting of the elements of ξ_{init} corresponding to the states of N. When starting from a state in F, the formula is the following:

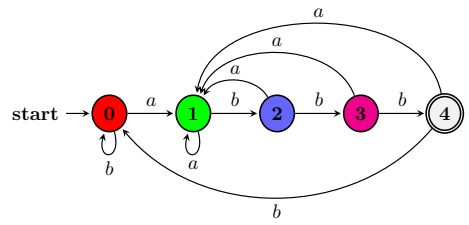
$$P(Y_n \in F, Y_{n-1} \in N, \cdots, Y_2 \in N, Y_1 \in F \mid \boldsymbol{\xi_{init}}) = \begin{cases} \boldsymbol{\xi_F}^T \boldsymbol{F}\mathbf{1} & \text{if } n = 2 \\ \boldsymbol{\xi_F}^T \boldsymbol{F_N} \boldsymbol{N}^{n-2}(\boldsymbol{I} - \boldsymbol{N})\mathbf{1} & \text{otherwise} \end{cases} \tag{3}$$

Proof. The proof of Eq. 2 may be found in [18]. The proof of Eq. 3 is presented in the technical report. □

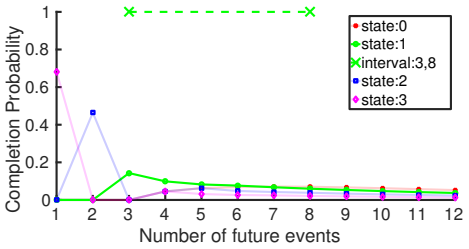
Note that the above formulas do not use $\boldsymbol{N_F}$, as it is not needed when dealing with probability distributions. As the sum of the probabilities is equal to 1, we can derive $\boldsymbol{N_F}$ from $\boldsymbol{N}$. This is the role of the term $(\boldsymbol{I}-\boldsymbol{N})\mathbf{1}$ in the formulas, which is equal to $\boldsymbol{N_F}$ when there is only a single final state and equal to the sum of the columns of $\boldsymbol{N_F}$ when there are multiple final states, i.e., each element of the matrix corresponds to the probability of reaching one of the final states from a given non-final state.

Using Theorem 2, we can calculate the so-called waiting-time distributions for any state q of the automaton, i.e., the distribution of the index n, given by the waiting-time variable $W_q = inf\{n : Y_0, Y_1, ..., Y_n, Y_0 = q, q \in Q \backslash F, Y_n \in F\}$. Theorem 2 provides a way to calculate the probability of reaching a final state, given an initial state distribution $\boldsymbol{\xi_{init}}$. In our case, as the automaton is moving through its various states, $\boldsymbol{\xi_{init}}$ takes a special form. At any point in time, the automaton is (with certainty) in a specific state q. In that state, $\boldsymbol{\xi_{init}}$ is a vector of 0, except for the element corresponding to the current state of the automaton, which is equal to 1.

Figure 5 shows an automaton along with the waiting-time distributions for its non-final states. For this example, if the automaton is in state 2, the probability of



(a) DFA.



(b) Waiting-time distributions and shortest interval, i.e. $[3, 8]$, exceeding a confidence threshold $\theta_{fc} = 50\%$ for state 1.

Fig. 5: Automaton and waiting-time distributions for $R = a \cdot b \cdot b \cdot b$, $\Sigma = \{a, b\}$.

reaching the final state 4 for the first time in 2 transitions is $\approx 50\%$. However, it is 0% for 3 transitions, as there is no path of length 3 from state 2 to state 4.

We can use the waiting-time distributions to produce various kinds of forecasts. In the simplest case, we can select the future point with the highest probability and return this point as a forecast. We call this type of forecasting *REGRESSION-ARGMAX*. Alternatively, we may want to know how likely it is that a CE will occur within the next w input events. For this, we can sum the probabilities of the first w points of a distribution and if this sum exceeds a given threshold we emit a "positive" forecast (meaning that a CE is indeed expected to occur); otherwise a "negative" (no CE is expected) forecast is emitted. We call this type of forecasting *CLASSIFICATION-NEXTW*. These kinds of forecasts are easy to compute. There is another kind of useful forecasts, which are however more computationally demanding. Given that we are in a state q, we may want to forecast whether the automaton, with confidence at least θ_{fc}, will have reached its final state(s) in n transitions, where n belongs to a future interval $I = [start, end]$. The confidence threshold θ_{fc} is a parameter set by the user. The forecasting objective is to select the shortest possible interval I that satisfies θ_{fc}. Figure 5b shows the forecast interval produced for state 1 of the automaton of Figure 5a, with $\theta_{fc} = 50\%$. We call this third type of forecasting *REGRESSION-INTERVAL*. We have implemented all of the above types of forecasting. Due to space limitations, in this paper we focus on *CLASSIFICATION-NEXTW*.

Note that the domain of a waiting-time distribution is not composed of timepoints and thus a forecast does not explicitly refer to time. Each value of the index n on the x axis essentially refers to the number of transitions that the automaton needs to take before reaching a final state, or, equivalently, to the number of future input events to be consumed. If we were required to output forecasts referring to time, we would need to convert these basic event-related forecasts to time-related ones (e.g., by trying to model the time elapsed between events via a Poisson process). In this paper we decided to focus on events, instead of timepoints, for two reasons: a) Sometimes it might not be desirable to give time-related forecasts. Event-related forecasts might be more suitable, as is the case, for example, in the domain of credit card fraud management, where we need to know whether or not the next transaction(s) will be fraudulent, independent of the actual time it(they) will happen. We examine this use case in Section 7.2. b) Time-related forecasts might be very difficult (or almost impossible) to produce if the underlying process exhibits a high degree of randomness, as is the case in the maritime domain, where AIS messages are produced in a random fashion and depend on many (even human-related) factors, e.g., the crew of a vessel simply forgetting to switch on the AIS equipment. In such cases, it might be preferable to perform some form of sampling or interpolation on the original stream of input events in order to derive another stream similar to the original one but with regular intervals. We follow this approach in our experiments in the maritime domain (Section 7.3).

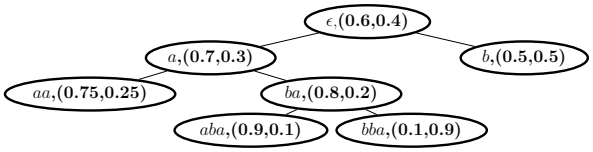
4.4.2 Using a Prediction Suffix Tree (PST)

The reason for constructing an embedding of the *PSA* M_S learned from the data into the automaton M_R used for recognition, as described in the previous section, is that the embedding is based on a variable-order model that will consist on average of much fewer states than a full-order model. There is, however, one specific step in the process of creating an embedding that may act as a bottleneck and prevent us from increasing the order to desired values: the step of converting a *PST* to a *PSA*. The number of nodes of a *PST* is often order of magnitudes smaller than the number of states of the *PSA* constructed from that *PST*. Motivated by this observation, we devised a way to estimate the required waiting-time distributions without actually constructing the embedding. Instead, we make direct use of the *PST*, which is more memory efficient. Thus, given a

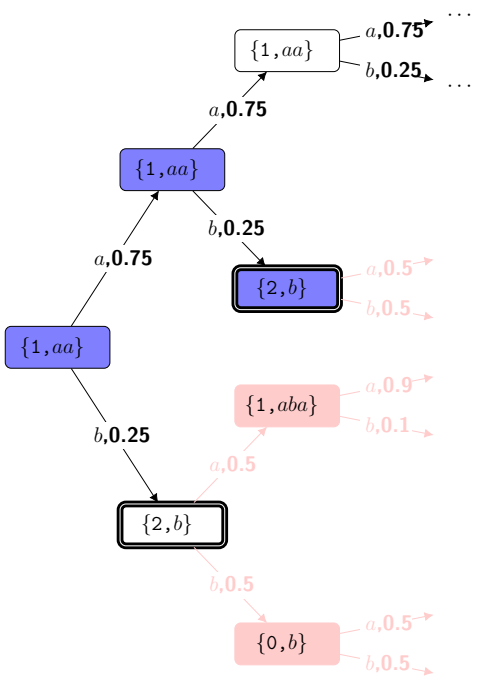
DSFA M_R and its *PST* T, we can estimate the probability for M_R to reach for the first time one of its final states in the following manner.

As the system processes events from the input stream, besides feeding them to M_R, it also stores them in a buffer that holds the m most recent events, where m is equal to the maximum order of the *PST* T. After updating the buffer with a new event, the system traverses T according to the contents of the buffer and arrives at a leaf l of T. The probability of any future sequence of events can be estimated with the use of the probability distribution at l. In other words, if $S_{1..k} = \cdots, t_{k-1}, t_k$ is the stream seen thus far, then the next symbol probability for t_{k+1}, i.e., $P(t_{k+1} \mid t_{k-m+1}, \cdots, t_k)$, can be directly retrieved from the distribution of the leaf l. If we want to look further into the future, e.g., into t_{k+2}, we can repeat the same process as necessary. Namely, if we fix t_{k+1}, then the probability for t_{k+2}, $P(t_{k+2} \mid t_{k-m+2}, \cdots, t_{k+1})$, can be retrieved from T, by retrieving the leaf l' reached with $t_{k+1}, \cdots, t_{k-m+2}$. In this manner, we can estimate the probability of any future sequence of events. Consequently, we can also estimate the probability of any future sequence of states of the *DSFA* M_R, since we can simply feed these future event sequences to M_R and let it perform "forward" recognition with these projected events. In other words, we can let M_R "generate" a sequence of future states, based on the sequence of projected events, in order to determine when M_R will reach a final state. Finally, since we can estimate the probability for any future sequence of states of M_R, we can use the definition of the waiting-time variable $(W_q = inf\{n : Y_0, Y_1, ..., Y_n, Y_0 = q, q \in Q \backslash F, Y_n \in F\})$ to calculate the waiting-time distributions. Figure 6 shows an example of this process for the automaton M_R of Figure 4a. Figure 6a displays an example *PST* T learned with the minterms/symbols of M_R.

One remark should be made at this point in order to showcase how an attempt to convert T to a *PSA* could lead to a blow-up in the number of states. The basic step in such a conversion is to take the leaves of T and use them as states for the *PSA*. If this was sufficient, the resulting *PSA* would always have fewer states than the *PST*. As this example shows, this is not the case. Imagine that the states of the *PSA* are just the leaves of T and that we are in the right-most state/node, $b, (0.5, 0.5)$. What will happen if an a event arrives? We would be unable to find a proper next state. The state $aa, (0.75, 0.25)$ is obviously not the correct one, whereas states $aba, (0.9, 0.1)$ and $bba, (0.1, 0.9)$ are both "correct", in the sense that ba is a suffix of both aba and bba. In order to overcome this ambiguity regarding the correct next state, we would have to first



(a) The *PST* T for the automaton M_R of Figure 4a.



(b) Future paths followed by M_R and T starting from state 1 of M_R and node aa of T. Purple nodes correspond to the only path of length $k = 2$ that leads to a final state. Pink nodes are pruned. Nodes with double borders correspond to final states of M_R.

Fig. 6: Example of estimating a waiting-time distribution without a Markov chain.

expand node $b, (0.5, 0.5)$ of T and then use the children of this node as states of the *PSA*. In this simple example, this expansion of a single problematic node would not have serious consequences. But for deep trees and large alphabets, the number of states generated by such expansions are far more than the number of the original leaves. For this reason, the size of the *PSA* is far greater than that of the original, unexpanded *PST*.

Figure 6b illustrates how we can estimate the probability for any future sequence of states of the *DSFA* M_R, using the distributions of the *PST* T. Let us assume that, after consuming the last event, M_R is in state 1 and T has reached its left-most node, $aa, (0.75, 0.25)$. This is shown as the left-most node also in Figure 6b. Each node in this figure has two elements: the first one

is the state of M_R and the second the node of T, starting with $\{1, aa\}$ as our current "configuration". Each node has two outgoing edges, one for a and one for b, indicating what might happen next and with what probability. For example, from the left-most node of Figure 6b, we know that, according to T, we might see a with probability 0.75 and b with probability 0.25. If we do encounter b, then M_R will move to state 2 and T will reach leaf $b, (0.5, 0.5)$. This is shown in Figure 6b as the white node $\{2, b\}$. This node has a double border to indicate that M_R has reached a final state.

In a similar manner, we can keep expanding this tree into the future and use it to estimate the waiting-time distribution for its node $\{1, aa\}$. In order to estimate the probability of reaching a final state for the first time in k transitions, we first find all the paths of length k which start from the original node and end in a final state without including another final state. In our example of Figure 6b, if $k = 1$, then the path from $\{1, aa\}$ to $\{2, b\}$ is such a path and its probability is 0.25. Thus, $P(W_{\{1,aa\}} = 1) = 0.25$. For $k = 2$, the path with the purple nodes leads to a final state after 2 transitions. Its probability is $0.75 * 0.25 = 0.1875$, i.e., the product of the probabilities on the path edges. Thus, $P(W_{\{1,aa\}} = 2) = 0.1875$. If there were more such alternative paths, we would have to add their probabilities.

Note that the tree-like structure of Figure 6b is not an actual data structure that we need to construct and maintain. It is only a graphical illustration of the required computation steps. The actual computation is performed recursively on demand. At each recursive call, a new frontier of virtual future nodes at level k is generated. We thus do not maintain all the nodes of this tree in memory, but only access the *PST* T, which is typically much more compact than a *PSA*. Despite this fact though, the size of the frontier after each recursive call grows exponentially as we try to look deeper into the future. This cost can be significantly reduced by employing the following optimizations. First, note in Figure 6b, that the paths starting from the two $\{2, b\}$ nodes are pink. This indicates that these paths do not actually need to be generated, as they start from a final state. We are only interested in the first time M_R reaches a final state and not in the second, third, etc. As a result, paths with more than one final states are not useful. With this optimization, we can still do an exact estimation of the waiting-time distribution. Another useful optimization is to prune paths that we know will have a very low probability, even if they are necessary for an exact estimation of the distributions. The intuition is that such paths will not contribute significantly to the probabilities of our waiting-time distribution, even if we do expand them. We can prune such paths, accepting the risk that we will have an approximate estimation of the waiting-time distribution. This pruning can be done without generating the paths in their totality. As soon as a partial path has a low probability, we can stop its expansion, since any deeper paths will have even lower probabilities. We have found this optimization to be very efficient while having negligible impact on the distribution for a wide range of cut-off thresholds.

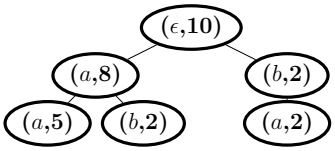


Fig. 7: Example of a Counter Suffix Tree with $m = 2$ and $S = aaabaabaaa$.

4.5 Estimation of Empirical Probabilities

We have thus far described how an embedding of a *PSA* M_S in a *DSFA* M_R can be constructed and how we can estimate the forecasts for this embedding. We have also presented how this can be done directly via a *PST*, without going through a *PSA*. However, before learning the *PST*, as described in Section 4.3, we first need to estimate the empirical probabilities for the various symbols. We describe here this extra initial step.

First, note that the empirical probabilities of the strings (s) and the expected next symbols (σ) observed in a stream are given by the following formulas [37]:

$$\hat{P}(s) = \frac{1}{k-m} \sum_{j=m}^{k-1} \chi_j(s) \tag{4}$$

$$\hat{P}(\sigma \mid s) = \frac{\sum_{j=m}^{k-1} \chi_{j+1}(s \cdot \sigma)}{\sum_{j=m}^{k-1} \chi_j(s)} \tag{5}$$

where k is the length of the training stream $S_{1..k}$, m is the maximum length of the strings (s) that will be considered and

$$\chi_j(s) = \begin{cases} 1 & \text{if } S_{(j-|s|+1)\cdots j} = s \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

In other words, we need to count the number of occurrences of the various candidate strings s in $S_{1..k}$. The numerators and denominators in Eq. (4) and (5) are essentially counters for the various strings.

In order to keep track of these counters, we can use a tree data structure which allows to scan the training stream only once. We call this structure a *Counter Suffix Tree* (*CST*). Each node in a *CST* is a tuple

(σ, c) where σ is a symbol from the alphabet (or ϵ only for the root node) and c a counter. By following a path from the root to a node, we get a string $s = \sigma_0 \cdot \sigma_1 \cdots \sigma_n$, where $\sigma_0 = \epsilon$ corresponds to the root node. The property maintained as a *CST* is built from a stream $S_{1..k}$ is that the counter of the node σ_n that is reached with s gives us the number of occurrences of the string $\sigma_n \cdot \sigma_{n-1} \cdots \sigma_1$ (the reversed version of s) in $S_{1..k}$. As an example, see Figure 7, which depicts the *CST* of maximum depth 2 for the stream $S = aaabaabaaa$. If we want to retrieve the number of occurrences of the string $b \cdot a$ in S, we follow the left child $(a, 7)$ of the root and then the right child of this. We thus reach $(b, 2)$ and indeed $b \cdot a$ occurs twice in S.

A *CST* can be incrementally constructed by maintaining a buffer of size m that always holds the last m symbols of S. The contents of the buffer are fed into the *CST* after the arrival of a new symbol. The update algorithm follows a path through the *CST* according to the whole string provided by the buffer. For every node that already exists, its counter is incremented by 1. If a node does not exist, it is created and its counter is set to 1. At any point, having been updated with the training stream, the *CST* can be used to retrieve the necessary counters and estimate the empirical probabilities of Equations (4) and (5) that are subsequently used in the *PST* construction.

5 Complexity Analysis

We now describe the steps required for estimating forecasts, along with the input required for each of them and their complexity. The first step takes as input the minterms of a *DSFA*, the maximum order m of dependencies to be captured and a training stream. Its output is a *CST* of maximum depth m (Section 4.5). In the second step, the *CST* is converted to a *PST*, using an approximation parameter α and a parameter n for the maximum number of states for the *PSA* to be constructed subsequently (Section 4.3). For the third step, we have two options: we can either use the *PST* to directly estimate the waiting-time distributions (Section 4.4.2) or we can convert the *PST* to a *PSA*, by using the leaves of the *PST* as states of the *PSA* (Section 4.3). If we follow the first path, we can then move on directly to the last step of estimating the actual forecasts, using the confidence threshold θ_{fc} provided by the user. If we follow the alternative path, the *PSA* is merged with the initial *DSFA* to create the embedding of the *PSA* in the *DSFA* (Section 4.4.1). From the embedding we can calculate the waiting-time distributions, which can be used to derive the forecasts.

The learning algorithm of step 2, as presented in [37], is polynomial in m, n, $\frac{1}{\alpha}$ and the size of the alphabet (number of minterms in our case). Below, we give complexity results for the remaining steps.

Proposition 2 (Step 1) *Let $S_{1..k}$ be a stream and m the maximum depth of the Counter Suffix Tree T to be constructed from $S_{1..k}$. The complexity of constructing T is $O(m(k-m))$.*

Proposition 3 (Step 3a) *Let T be a PST of maximum depth m, learned with the t minterms of a DSFA M_R. The complexity of constructing a PSA M_S from T is $O(t^{m+1} \cdot m)$.*

Proposition 4 (Step 3b) *Let T be a PST of maximum depth m, learned with the t minterms of a DSFA M_R. The complexity of estimating the waiting-time distribution for a state of M_R and a horizon of length h directly from T is $O((m+3)\frac{t-t^{h+1}}{1-t})$.*

Proposition 5 (Step 4) *Let M_R be a DSFA with t minterms and M_S a PSA learned with the minterms of M_R. The complexity of constructing an embedding M of M_S in M_S is $O(t \cdot |M_R.Q \times M_S.Q|)$.*

Proposition 6 (Step 5) *Let M be the embedding of a PSA M_S in a DSFA M_R. The complexity of estimating the waiting-time distribution for a state of M and a horizon of length h using Theorem 2 is $O((h-1)k^{2.37})$, where k is the dimension of the square matrix $\boldsymbol{N}$.*

Proposition 7 (Step 6) *For a waiting-time distribution with horizon h, the complexity of REGRESSION-INTERVAL is $O(h)$ and that of CLASSIFICATION-NEXTW is $O(w)$ for a future window of length w.*

Detailed proofs for the complexities of the algorithms for all the above steps may be found in the extended technical report.

6 Measuring the Quality of Forecasts

Our system can perform all types of forecasting described in Section 4.4.1. In this paper we focus on classification forecasting, since it can be evaluated in a straightforward manner. Regression forecasting does not allow us to test how well a system performs in the absence of CEs as it always assumes that a CE does indeed occur in the future. Our system can also perform SDE forecasting, e.g., by using the next symbol distributions of a suffix tree's leaves. On the other hand, as already mentioned (Section 2), CE forecasting is not a straightforward extension of SDE forecasting.

In order to properly measure the quality of the produced forecasts, we first need to decide when it makes sense to emit forecasts, i.e., to establish *checkpoints* in the stream. Eagerly emitting forecasts after every new SDE is feasible, but not very useful and can also produce results that are misleading. By their very nature, CEs are relatively rare within a stream of input SDEs. As a result, if we emit a forecast after every new SDE, some of these forecasts (possibly even the majority) will have a significant temporal distance from the CE to which they refer. As an example, consider a pattern from the maritime domain which detects the entrance of a vessel in the port of Tangiers. We can also try to use this pattern for forecasting when the vessel will arrive at the port of Tangiers. However, the majority of the vessel's messages may lie in areas so distant from the port (e.g., in the Pacific ocean) that it would be practically useless to emit forecasts when the vessel is in these areas. Moreover, if we do emit forecasts from these distant areas, the scores and metrics that we use to evaluate the quality of the forecasts will be dominated by mostly low-quality, distant forecasts.

Our proposed solution is to emit forecasts only when the CE is relatively close in the future. This can be achieved by using a pattern's automaton to estimate at every point how close the automaton is to reaching a final state and thus detecting a CE. We can then establish checkpoints only at these points in the stream where the automaton is not very "far" from reaching a final state. We can use the structure of the automaton itself to estimate these distances to CEs. We may not know the actual distance to a CE, but the automaton can provide us with an "expected" or "possible" distance, as follows. For an automaton that is already in a final state, it can be said that the distance to a CE is 0. More conveniently, we can say that the "process" that the automaton describes has been completed or, equivalently, that there remains 0% of the process until completion. For an automaton that is in a non-final state but separated from a final state by 1 transition, it can be said that the "expected" distance is 1. We use the term "expected" because we are not interested in whether the automaton will actually make the transition to a final state. We want to establish checkpoints both for the presence and the absence of CEs. When the automaton fails to make the transition to a final state, this "expected" distance remains a "possible" one that failed to materialize. We also note that there might also exist other walks from this non-final state to a final one whose length could be greater than 1 (in fact, there might exist walks with "infinite length", in the case of loops). In order to estimate the "expected" distance of a non-final state, we only use the shortest walk to a final state. Note that these distances are completely independent of any stream to be processed by the automaton. They are estimated by looking exclusively at the structure of the automaton.

After estimating the expected distances of all states, we can then express them as percentages by dividing them by the greatest among them. A 0% distance will thus refer to final states, whereas a 100% distance to the state(s) that are the most distant to a final state, i.e., the automaton has to take the most transitions to reach a final state. These are the start states. We can then determine our checkpoints by specifying the states in which the automaton is permitted to emit forecasts, according to their "expected" distance. For example, we may establish checkpoints by allowing only states with a distance between 40% and 60% to emit forecasts. The intuition here is that, by increasing the allowed distance, we make the forecasting task more difficult.

The evaluation task itself consists of the following steps. At the arrival of every new input event, we first check whether the distance of the new automaton state falls within the range of allowed distances, as explained above. If the new state is allowed to emit a forecast, we use its waiting-time distribution to produce the forecast. Two parameters are taken into account: the length of the future window w within which we want to know whether a CE will occur and the confidence threshold θ_{fc}. If the probability of the first w points of the distribution exceeds the threshold θ_{fc}, we emit a positive forecast, essentially affirming that a CE will occur within the next w events; otherwise, we emit a negative forecast, essentially rejecting the hypothesis that a CE will occur. We thus obtain a binary classification task.

As a consequence, we can make use of standard classification measures, like precision and recall. Each forecast is evaluated: a) as a *true positive* (TP) if the forecast is positive and the CE does indeed occur within the next w events from the forecast; b) as a *false positive* (FP) if the forecast is positive and the CE does not occur; c) as a *true negative* (TN) if the forecast is negative and the CE does not occur and d) as a *false negative* (FN) if the forecast is negative and the CE does occur; Precision is then defined as $Precision = \frac{TP}{TP+FP}$ and recall (also called sensitivity or true positive rate) as $Recall = \frac{TP}{TP+FN}$. As already mentioned, CEs are relatively rare in a stream. It is thus important for a forecasting engine to be as specific as possible in identifying the true negatives. For this reason, besides precision and recall, we also use *specificity* (also called true negative rate), defined as $Specificity = \frac{TN}{TN+FP}$.

A classification experiment is performed as follows. For various values of the "expected" distance and the confidence threshold θ_{fc}, we estimate precision, recall

and specificity on a test dataset. For a given distance, θ_{fc} acts as a cut-off parameter. For each value of θ_{fc}, we estimate the recall (sensitivity) and specificity scores and we plot these scores as a ROC curve. For each distance, we then estimate the area under curve (AUC) for the ROC curves. The higher the AUC value, the better the model is assumed to be.

The setting described above is the most suitable for evaluation purposes, but might not be the most appropriate when such a system is actually deployed. For deployment purposes, another option would be to simply set a best, fixed confidence threshold (e.g., by selecting, after evaluation, the threshold with the highest F1-score or Matthews correlation coefficient) and emit only positive forecasts, regardless of their distance. Forecasts with low probabilities (i.e., negative forecasts) will thus be ignored/suppressed. This is justified by the fact that a user would typically be more interested in positive forecasts. For evaluation purposes, this would not be an appropriate experimental setting, but it would suffice for deployment purposes, where we would then be focused on fine-tuning the confidence threshold. In this paper, we focus on evaluating our system and thus do not discuss further any deployment solution.

7 Empirical Evaluation

We now present experimental results on two datasets, a synthetic one (Section 7.2) and a real-world one (Section 7.3). We first briefly discuss the models that we evaluated and present our software and hardware settings in Section 7.1.

7.1 Models Tested and Settings

In the experiments that we present, we evaluated the variable-order Markov model that we have presented in this paper in its two versions: the memory efficient one that bypasses the construction of a Markov chain and makes direct use of the *PST* learned from a stream (Section 4.4.2) and the computationally efficient one that constructs a *PSA* (Section 4.4.1). We compared these against four other models inspired by the relevant literature.

The first, described in [6, 7], is the most similar in its general outline to our proposed method. It is a previous version of our system presented in this paper and is also based on automata and Markov chains. The main difference is that it attempts to construct full-order Markov models of order m and is thus typically restricted to low values for m. The second model is presented in [29], where automata and Markov chains are used once again. However, the automata are directly mapped to Markov chains and no attempt is made to ensure that the Markov chain is of a certain order. Thus, in the worst case, this model essentially makes the assumption that SDEs are i.i.d. and $m = 0$.

As a third alternative, we evaluated a model that is based on Hidden Markov Models (HMM), similar to the work presented in [31]. That work uses the Esper event processing engine [1] and attempts to model a business process as a HMM. For our purposes, we use a HMM to describe the behavior of an automaton, constructed from a given symbolic regular expression. The observation variable of the HMM corresponds to the states of the automaton. Thus, the set of possible values of the observation variable is the set of automaton states. An observation sequence of length l for the HMM consists of the sequence of l states visited by the automaton after consuming l SDEs. The l SDEs (symbols) are used as values for the hidden variable. The last l symbols are the last l values of the hidden variable. Therefore, this HMM always has l hidden states, whose values are taken from the SDEs, connected to l observations, whose values are taken from the automaton states. We can train such a HMM with the Baum-Welch algorithm, using the automaton to generate a training observation sequence from the original training stream. We can then use this learned HMM to produce forecasts on a test dataset. We produce forecasts in an online manner as follows: as the stream is consumed, we use a buffer to store the last l states visited by the pattern automaton. After every new event, we "unroll" the HMM using the contents of the buffer as the observation sequence and the transition and emission matrices learned during the training phase. We can then use the forward algorithm to estimate the probability of all possible future observation sequences (up to some length), which, in our case, correspond to future states visited by the automaton. Knowing the probability of every future sequence of states allows us to estimate the waiting-time distribution for the current state of the automaton and thus build a forecast, as already described. Note that, contrary to the previous approaches, the estimation of the waiting-time distribution via a HMM must be performed online. We cannot pre-compute the waiting-time distributions and store the forecasts in a look-up table, due to the possibly large number of entries. For example, assume that $l = 5$ and the size of the "alphabet" (SDE symbols) of our automaton is 10. For each state of the automaton, we would have to pre-compute 10^5 entries. In other words, as with Markov chains, we still have a problem of combinatorial explosion. We try to "avoid" this problem by estimating the waiting-time distributions online.

Our last model is inspired by the work presented in [3]. This method comes from the process mining community and has not been previously applied to CEF. However, due to its simplicity, we use it here as a baseline method. We again use a training dataset to learn the model. In the training phase, every time the pattern automaton reaches a certain state q, we simply count how long (how many transitions) we have to wait until it reaches a final state. After the training dataset has been consumed, we end up with a set of such "waiting times" for every state. The forecast to be produced by each state is then estimated simply by calculating the average "waiting time".

As far as the Markov models are concerned, we try to increase their order to the highest possible value, in order to determine if and how high-order values offer an advantage. We have empirically discovered that our system can efficiently handle automata and Markov chains that have up to about 1200 states. Beyond this point, it becomes almost prohibitive (with our hardware) to create and handle transition matrices with more than 1200^2 elements. We have thus set this number as an upper bound and increased the order of a model until this number is reached. This restriction is applied both to full-order models and variable-order models that use a *PSA* and an embedding, since in both of these cases we need to construct a Markov chain. For the variable-order models that make direct use of a *PST*, no Markov chain is constructed. We thus increase their order until their performance scores seem to reach a stable number or a very high number, beyond which it makes little sense to continue testing.

All experiments were run on a 64-bit Debian 10 machine with Intel Core i7-8700 CPU @ 3.20GHz X 12 processors and 16 GB of memory. Our framework was implemented in Scala 2.12.10. We used Java 1.8, with the default values for the heap size. For the HMM models, we relied on the Smile machine learning library [2]. All other models were developed by us. No attempt at parallelization was made.

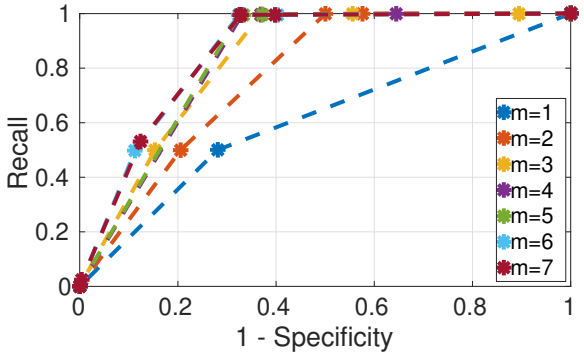
7.2 Credit Card Fraud Management

The first dataset used in our experiments is a synthetic one, inspired by the domain of credit card fraud management [9]. We start with a synthetically generated dataset in order to investigate how our method performs under conditions that are controlled and produce results more readily interpretable. The data generator was developed in collaboration with Feedzai[2], our partner in the SPEEDD project[3].

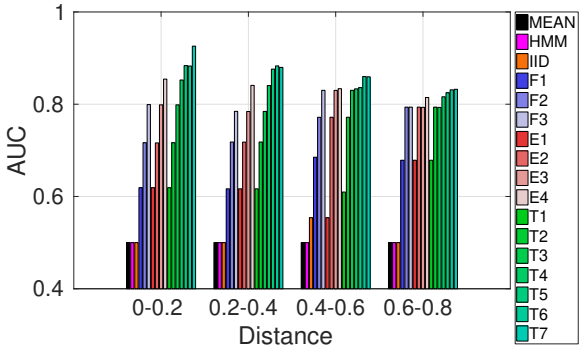
In this dataset, each event is supposed to be a credit card transaction, accompanied by several arguments, such as the time of the transaction, the card ID, the amount of money spent, the country where the transaction took place, etc. In the real world, a very small proportion of such transactions are fraudulent and the goal of a CER system would be to detect, with very low latency, fraud instances. To do so, a set of fraud patterns must be provided to the engine. For typical cases of such patterns in a simplified form, see [9]. In our experiments, we use one such pattern, consisting of a sequence of consecutive transactions, where the amount spent at each transaction is greater than that of the previous transaction. Such a trend of steadily increasing amounts constitutes a typical fraud pattern. The goal in our forecasting experiments is to predict if and when such a pattern will be completed, even before it is detected by the engine (if in fact a fraud instance occurs), so as to possibly provide a wider margin for action to an analyst.

We generated a dataset consisting of 1,000,000 transactions in total from 100 different cards. About 20% of the transactions are fraudulent. Not all of these instances of fraud belong to the pattern of increasing amounts. We actually inject seven different types of known fraudulent patterns in the dataset, including, for instance, a decreasing trend. Each fraudulent sequence for the increasing trend consists of eight consecutive transactions with increasing amounts, where the amount is increased each time by 100 monetary units or more. We additionally inject sequences of transactions with increasing amounts, which are not fraudulent. In those cases, we randomly interrupt the sequence before it reaches the eighth transaction. In the legitimate sequences the amount is increased each time by 0 or more units. With this setting, we want to test the effect of long-term dependencies on the quality of the forecasts. For example, a sequence of six transactions with increasing amounts, where all increases are 100 or more units is very likely to lead to a fraud detection. On the other hand, a sequence of just two transactions with the same characteristics, could still possibly lead to a detection, but with a significantly reduced probability. We thus expect that models with deeper memories will perform better. We used 75% of the dataset for training and the rest for testing. No k-fold cross validation is performed, since each fold would have exactly the same statistical properties.

[2] `https://feedzai.com`
[3] `http://speedd-project.eu`



(a) ROC curves for the variable-order model using the *PST* for various values of the maximum order m. $distance \in [0.4, 0.6]$.



(b) AUC for ROC curves for all models.

Fig. 8: Results for CEF for credit card fraud management. Fx stands for a Full-order Markov Model of order x. Ex stands for a Variable-order Markov Model of maximum order x that uses a *PSA* and creates an embedding. Tx stands for a Variable-order Markov Model of maximum order x that is constructed directly from a *PST*. MEAN stands for the method of estimating the mean of "waiting-times". HMM stands for Hidden Markov Model. IID stands for the method assuming (in the worst case) that SDEs are i.i.d. Ex and Tx models are the ones proposed in this paper.

Formally, the symbolic regular expression that we use to capture the pattern of an increasing trend in the amount spent is the following:

$$\begin{aligned} R := \; & (amountDiff > 0) \cdot (amountDiff > 0) \cdot \\ & (amountDiff > 0) \cdot (amountDiff > 0) \cdot \\ & (amountDiff > 0) \cdot (amountDiff > 0) \cdot \\ & (amountDiff > 0) \end{aligned} \tag{7}$$

amountDiff is an extra attribute (besides the card ID, the amount spent, the transaction country and the other standard attributes) with which we enrich each event and is equal to the difference between the amount spent by the current transaction and that spent by the immediately previous transaction from the same card. The expression consists of seven terminal sub-expressions, in order to capture eight consecutive events. The first terminal sub-expression captures an increasing amount between the first two events in a fraudulent pattern.

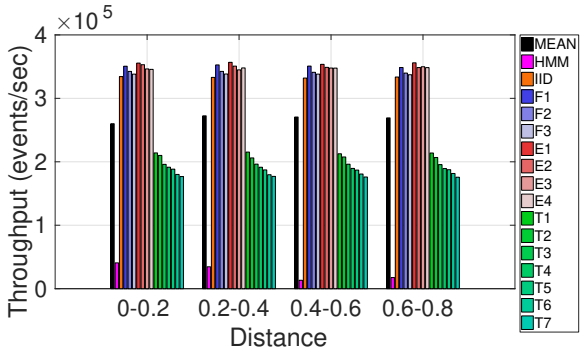
If we attempted to perform forecasting based solely on Pattern (7), then the minterms that would be created would be based only on the predicate $amountDiff > 0$: namely, the predicate itself, along with its negation $\neg(amountDiff > 0)$. As expected, such an approach does not yield good results, as the language is not expressive enough to differentiate between fraudulent and legitimate transaction sequences. In order to address this lack of informative (for forecasting purposes) predicates, we have incorporated a mechanism in our system that allows us to incorporate extra predicates when building a probabilistic model, without affecting the semantics of the initial expression (exactly the same matches are detected). We do this by using any such extra predicates during the construction of the minterms. For example, if $country = MA$ is such an extra predicate that we would like included, then we would construct the following minterms for Pattern (7): a) $m_1 = (amountDiff > 0) \wedge (country = MA)$; b) $m_2 = (amountDiff > 0) \wedge \neg(country = MA)$; c) $m_3 = \neg(amountDiff > 0) \wedge (country = MA)$; d) $m_4 = \neg(amountDiff > 0) \wedge \neg(country = MA)$). We can then use these enhanced minterms as guards on the automaton transitions in a way that does not affect the semantics of the expression. For example, if an initial transition has the guard $amountDiff > 0$, then we can split it into two derived transitions, one for m_1 and one for m_2. The derived transitions would be triggered exactly when the initial one is triggered, the only difference being that the derived transitions also have information about the country. For our experiments and for Pattern (7), if we include the extra predicate $amountDiff > 100$, we expect the forecasting model to be able to differentiate between sequences involving genuine transactions (where the difference in the amount can by any value above 0) and fraudulent sequences (where the difference in the amount is always above 100 units).

Figure 8 shows the ROC curves of the variable-order model that directly uses a *PST*. We show results for the "expected" distance range of $distance \in [0.4, 0.6]$ in Figure 8a. The ideal operating point in the ROC is the top-left corner and thus, the closer to that point the curve is, the better. Thus, the first observation is that by increasing the maximum order we obtain better results. Figure 8b displays ROC results for different distances and all models, in terms of the Area Under the ROC Curve (AUC), which is a measure of the models' classification accuracy. The first observation is that the MEAN and HMM methods consistently underper-

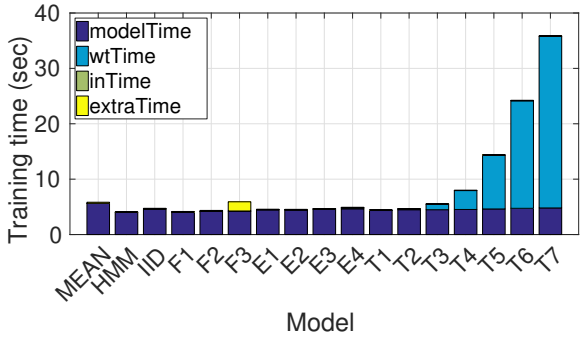
form, compared to the Markov models. Focusing on the Markov models, as expected, the task becomes more challenging and the ROC scores decrease, as the distance increases. It is also evident that higher orders lead to better results. The advantage of increasing the order becomes less pronounced (or even non-existent) as the distance increases. The variable-order models that use an embedding are only able to go as far as $m = 4$, due to increasing memory requirements, whereas the tree-based versions can go up to $m = 7$ (and possibly even further, but we did not try to extend the order beyond this point). Although the embedding (*PSA*) can indeed help achieve better scores than full-order models by reaching higher orders, this is especially true for the tree-based models which bypass the embedding. We can thus conclude that full-order models are doing well up to the order that they we can achieve with them. *PSA* models can reach roughly the same levels, as they are also practically restricted. The performance of *PST* models is similar to that of the other models for the same order, but the fact that they can use higher orders allows them to finally obtain superior performance.

We show performance results in Figure 9, in terms of computation and memory efficiency. Figure 9a displays throughput results. We can observe the trade-off between the high forecasting accuracy of the tree-based high-order models and the performance penalty that these models incur. The models based on *PST* have a throughput figure that is almost half that of the full-order models and the embedding-based variable-order ones. In order to emit a forecast, the tree-based models need to traverse a tree after every new event arrives at the system, as described in Section 4.4.2. The automata-based full- and variable-order models, on the contrary, only need to evaluate the minterms on the outgoing transitions of their current state and simply jump to the next state. It would be possible to improve the throughput of the tree-based models, by using caching techniques, so that we can reuse some of the previously estimated forecasts, but we reserve such optimizations for future work. By far the worst throughput, however, is observed for the HMM models. The reason is that the waiting-time distributions and forecasts are always estimated online, as explained in Section 7.1.

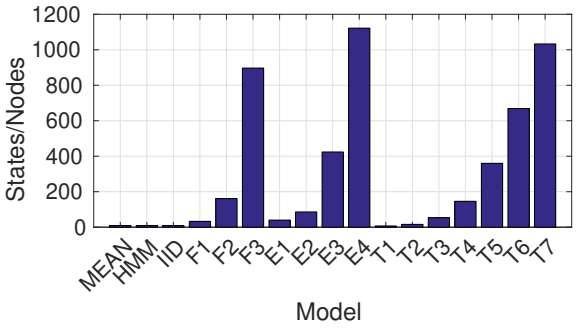
Figure 9b shows training times as a stacked, bar plot. For each model, the total training time is broken down into 4 different components, each corresponding to a different phase of the forecast building process. *modelTime* is the time required to actually construct the model from the training dataset. *wtTime* is the time required to estimate the waiting-time distributions, once the model has been constructed. *inTime*



(a) Throughput.



(b) Training time.



(c) Number of states/nodes.

Fig. 9: Throughput, training time and number of automaton states/tree nodes for classification CEF for credit card fraud management.

measures the time required to estimate the forecast of each waiting-time distribution. Finally, *extraTime* measures the time required to determinize the automaton of our initial pattern. For the full-order Markov models, it also includes the time required to convert the deterministic automaton into its equivalent, disambiguated automaton. We observe that the tree-based models exhibit significantly higher times than the rest, for high orders. The other models have similar training times, almost always below 5 seconds. Thus, if we need high accuracy, we again have to pay a price in terms of training time. Even in the case of high-order tree-based models though, the training time is almost half a minute for a training dataset composed of 750,000 transactions,

which allows us to be confident that training could be performed online.

Figure 9c shows the memory footprint of the models in terms of the size of their basic data structures. For automata-based methods, we show the number of states, whereas for the tree-based methods we show the number of nodes. We see that variable-order models, especially the tree-based ones, are significantly more compact than the full-order ones, for the same order. We also observe that the tree-based methods, for the same order, are much more compact (fewer nodes) than the ones based on the embedding (more states). This allows us to increase the order up to 7 with the tree-based approach, but only up to 4 with the embedding.

7.3 Maritime Situational Awareness

The second dataset that we used in our experiments is a real–world dataset coming from the field of maritime monitoring. It is composed of a set of trajectories from ships sailing at sea, emitting AIS (Automatic Identification System) messages that relay information about their position, heading, speed, etc., as described in the running example of Section 1. These trajectories can be analyzed, using the techniques of Complex Event Recognition, in order to detect interesting patterns in the behavior of vessels [32]. The dataset that we used is publicly available, contains AIS kinematic messages from vessels sailing in the Atlantic Ocean around the port of Brest, France, and spans a period from 1 October 2015 to 31 March 2016 [35]. We used a derivative dataset that contains clean and compressed trajectories, consisting only of critical points [33]. Critical points are the important points of a trajectory that indicate a significant change in the behavior of a vessel. Using critical points, one can reconstruct quite accurately the original trajectory [32]. We further processed the dataset by interpolating between the critical points in order to produce trajectories where two consecutive points have a temporal distance of exactly 60 seconds. The reason for this pre-processing step is that AIS messages typically arrive at unspecified time intervals. These intervals can exhibit a very wide variation, depending on many factors (e.g., human operators may turn on/off the AIS equipment), without any clear pattern that could be encoded by our probabilistic model. Consequently, our system performs this interpolation as a first step.

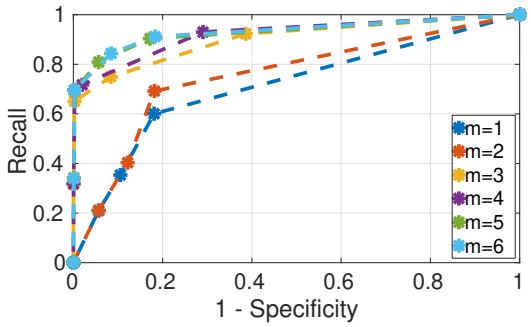
The pattern that we used in the experiments is a movement pattern in which a vessel approaches the main port of Brest. The goal is to forecast when a vessel will enter the port. This way, port traffic management may be optimized, in order to reduce the carbon emissions of vessels waiting to enter the port. The symbolic regular expression for this pattern is the following:

$$R := (\neg InsidePort(Brest))^* \cdot (\neg InsidePort(Brest)) \cdot (\neg InsidePort(Brest)) \cdot (InsidePort(Brest)) \quad (8)$$

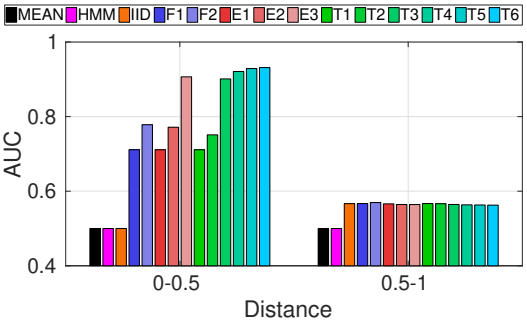
The intention is to detect the entrance of a vessel in the port of Brest. The predicate $InsidePort(Brest)$ evaluates to TRUE whenever a vessel has a distance of less than 5 km from the port of Brest. In fact, the predicate is generic and takes as arguments the longitude and latitude of any point, but we show here a simplified version, using the port of Brest, for reasons of readability. The pattern defines the entrance to the port as a sequence of at least 3 consecutive events, only the last of which satisfies the $InsidePort(Brest)$ predicate. In order to detect an entrance, we must first ensure that the previous event(s) indicated that the vessel was outside the port. For this reason, we require that, before the last event, there must have occurred at least 2 events where the vessel was outside the port. We require 2 or more such events to have occurred (instead of just one), in order to avoid detecting "noisy" entrances.

In addition to the $InsidePort(Brest)$ predicate, we included 5 extra ones providing information about the distance of a vessel from a port when it is outside the port. Each of these predicates evaluates to TRUE when a vessel lies within a specified range of distances from the port. The first returns TRUE when a vessel has a distance between 5 and 6 km from the port, the second when the distance is between 6 and 7 km and the other three extend similarly 1 km until 10 km. We investigated the sensitivity of our models to the presence of various extra predicates in the recognition pattern.

For all experimental results that follow, we always present average values over 4 folds of cross-validation. We start by analyzing the trajectories of a single vessel and then move to multiple, selected vessels. There are two issues that we tried to address by separating our experiments into single-vessel and multiple-vessel ones. First, we wanted to have enough data for training. For this reason, we only retained vessels for which we can detect a significant number of matches for Pattern (8). Second, our system can work in two modes: a) it can build a separate model for each monitored object and use this collection of models for personalized forecasting; b) it can build a global model out of all the monitored objects. We thus wanted to examine whether building a global model from multiple vessels could produce equally good results, as these obtained for a single vessel with sufficient training data.

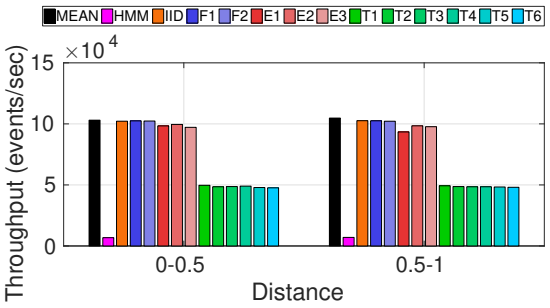


(a) ROC curves for the variable-order model using the *PST* for various values of the maximum order. *distance* $\in [0.0, 0.5]$.

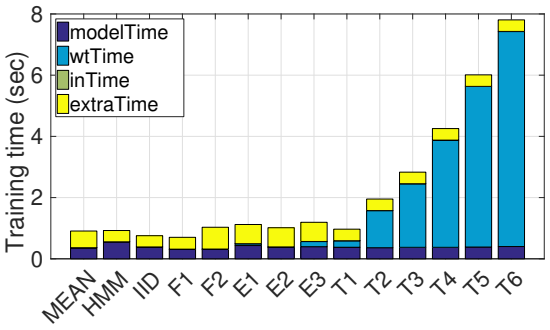


(b) AUC for ROC curves for all models.

Fig. 10: Results for CEF in the domain of maritime situational awareness. Ex and Tx models are the ones proposed in this paper.



(a) Throughput.



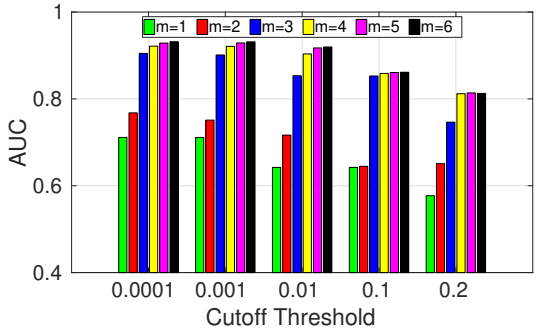
(b) Training time.

Fig. 11: Throughput and training time results for CEF for maritime situational awareness.

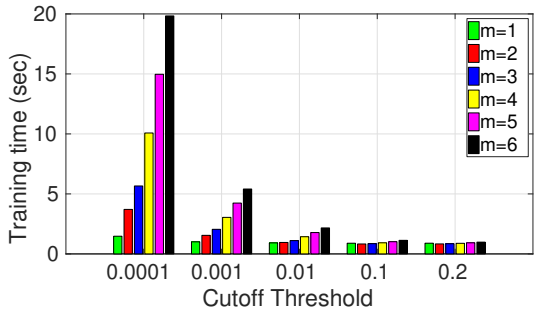
We first used Pattern (8) to perform recognition on the whole dataset in order to find the number of matches detected for each vessel. The vessel with the most matches was then isolated and we retained only the events emitted from this vessel. In total, we detected 368 matches for this vessel and the number of SDEs corresponding to it is ≈ 30.000.

Using this vessel, we obtained the results shown in Figures 10 and 11. Since the original *DSFA* is smaller in this case (one start and one final state plus two intermediate states), we have fewer distance ranges (e.g., there no states in the range $[0.4, 0.6]$). Thus, we use only two distance ranges: $[0, 0.5]$ and $[0.5, 1]$. We observe the importance of being able to increase the order of our models for distances smaller than 50%. For distances greater than 50%, the area under curve is ≈ 0.5 for all models. This implies that they cannot effectively differentiate between positives and negatives. Notice that the full-order Markov models can now only go up to $m = 2$, since the existence of multiple extra predicates makes it prohibitive to increase the order any further. Achieving higher accuracy with higher-order models comes at a computational cost, as shown in Figure 11. The results are similar to those in the credit card fraud experiments. The training time for variable-order models tends to increase as we increase the order, but is always less than 8 seconds. The effect on throughput is again significant for the tree-based variable-order models. Throughput figures are also lower here compared to the credit card fraud experiments, since the predicates that we need to evaluate for every new input event (like $InsidePort(Brest)$) involve more complex calculations (the $amountDiff > 0$ predicate is a simple comparison).

As a next step, we wanted to investigate the effect of the optimization technique mentioned at the end of Section 4.4.2 on the accuracy and performance of our system. The optimization prunes future paths whose probability is below a given cutoff threshold. We re-run the experiments described above for distances between 0% and 50% for various values of the cutoff threshold, starting from 0.0001 up to 0.2. Figure 12 shows the relevant results. We observe that the accuracy is affected only for high values of the cutoff threshold, above 0.1 (Figure 12a). We can also see that the training time is indeed significantly affected (Figure 12b). As expected, the result of increasing the value of the cutoff threshold is a reduction of the training time, as fewer paths are retained. Beyond a certain point though, further increases of the cutoff threshold affect the accuracy of the system. Therefore, the cutoff threshold should be below 0.01 so as not to compromise the accuracy of our



(a) AUC-ROC.



(b) Training time.

Fig. 12: Effect of cutoff threshold on accuracy and training time.

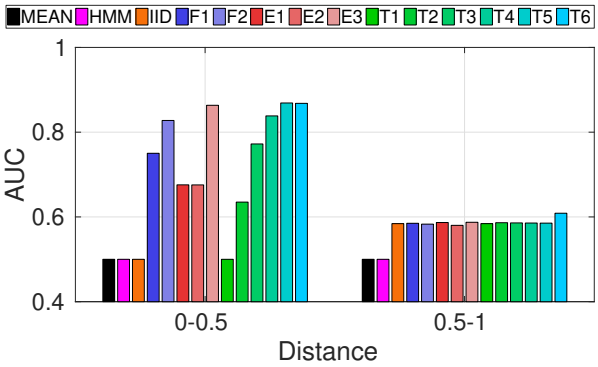


Fig. 13: AUC for ROC curves. Extra features included: concentric rings around the port every 1 km. Model constructed for the 9 vessels that have more than 100 matches.

forecasts. We do not show throughput results, because throughput remains essentially unaffected. This result is expected, since the cutoff threshold is only used in the estimation of the waiting-time distributions. Throughput reflects the online performance of our system, after the waiting-time distributions have been estimated, and is thus not affected by the choice of the cutoff threshold.

Finally, we also tested our method when more than one vessel need to be monitored. Instead of isolating the single vessel with the most matches, we isolated all vessels which had more than 100 matches. There are in total 9 such vessels in the dataset. The resulting dataset has $\approx$ 222.000 events. Out of the 9 retained vessels, we constructed a global probabilistic model and produced forecasts. An alternative option would be to build a single model for each vessel, but in this scenario we wanted to test the robustness of our aprroach when a global model is built from multiple entities. Figure 13 presents the corresponding results. Interestingly, the scores of the global model remain very close to the scores of the experiments for the single vessel with the most matches (Figure 10b). This is an indication of the ability of the global model to capture the peculiarities of individual vessels.

8 Summary

We have presented a framework for Complex Event Forecasting (CEF), based on a variable-order Markov model. It allows us to delve deeper into the past and capture long-term dependencies, not feasible with full-order models. Our comprehensive evaluation on two application domains has illustrated the advantages of being able to use such high-order models. Namely, the use of higher-order modeling allows us to achieve higher accuracy than what is possible with full-order models or other state-of-the-art solutions. We have described two alternative ways in which variable-order models may be used, depending on the imposed requirements. One option is to use a highly efficient but less accurate model, when online performance is a top priority. We also provide an option that achieves high accuracy scores, but with a performance cost. Another important feature of our proposed framework is that it requires minimal intervention by the user. A given Complex Event pattern is declaratively defined and subsequently automatically translated to an automaton and then to a Markov model, without requiring domain knowledge that should guide the modeling process.

Acknowledgements This work has received funding from the EU Horizon 2020 research and innovation program INFORE under grant agreement No 825070.

References

1. Esper. `http://www.espertech.com/esper`
2. Smile - statistical machine intelligence and learning engine. `http://haifengl.github.io/`
3. van der Aalst, W.M.P., Schonenberg, M.H., Song, M.: Time prediction based on process mining. Inf. Syst. **36**(2), 450–475 (2011)
4. Abe, N., Warmuth, M.K.: On the computational complexity of approximating distributions by probabilistic automata. Machine Learning **9**, 205–260 (1992)

5. Akbar, A., Carrez, F., Moessner, K., Zoha, A.: Predicting complex events for pro-active iot applications. In: WF-IoT, pp. 327–332. IEEE Computer Society (2015)
6. Alevizos, E., Artikis, A., Paliouras, G.: Event forecasting with pattern markov chains. In: DEBS, pp. 146–157. ACM (2017)
7. Alevizos, E., Artikis, A., Paliouras, G.: Wayeb: a tool for complex event forecasting. In: LPAR, *EPiC Series in Computing*, vol. 57, pp. 26–35. EasyChair (2018)
8. Alevizos, E., Skarlatidis, A., Artikis, A., Paliouras, G.: Probabilistic complex event recognition: A survey. ACM Comput. Surv. **50**(5), 71:1–71:31 (2017)
9. Artikis, A., Katzouris, N., Correia, I., Baber, C., Morar, N., Skarbovsky, I., Fournier, F., Paliouras, G.: A prototype for credit card fraud management: Industry paper. In: DEBS, pp. 249–260. ACM (2017)
10. Begleiter, R., El-Yaniv, R., Yona, G.: On prediction using variable order markov models. J. Artif. Intell. Res. **22**, 385–421 (2004)
11. Chang, B., Park, Y., Park, D., Kim, S., Kang, J.: Content-aware hierarchical point-of-interest embedding model for successive POI recommendation. In: IJCAI, pp. 3301–3307. ijcai.org (2018)
12. Cho, C., Wu, Y., Yen, S., Zheng, Y., Chen, A.L.P.: On-line rule matching for event prediction. VLDB J. **20**(3), 303–334 (2011)
13. Christ, M., Krumeich, J., Kempa-Liehr, A.W.: Integrating predictive analytics into complex event processing by using conditional density estimations. In: EDOC Workshops, pp. 1–8. IEEE Computer Society (2016)
14. Cleary, J.G., Witten, I.H.: Data compression using adaptive coding and partial string matching. IEEE Trans. Communications **32**(4), 396–402 (1984)
15. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Comput. Surv. **44**(3), 15:1–15:62 (2012)
16. D'Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: CAV (1), *Lecture Notes in Computer Science*, vol. 10426, pp. 47–67. Springer (2017)
17. Engel, Y., Etzion, O.: Towards proactive event-driven computing. In: DEBS, pp. 125–136. ACM (2011)
18. Fu, J.C., Lou, W.W.: Distribution theory of runs and patterns and its applications: a finite Markov chain imbedding approach. World Scientific (2003)
19. Fülöp, L.J., Beszédes, Á., Toth, G., Demeter, H., Vidács, L., Farkas, L.: Predictive complex event processing: a conceptual framework for combining complex event processing and predictive analytics. In: BCI, pp. 26–31. ACM (2012)
20. Giatrakos, N., Alevizos, E., Artikis, A., Deligiannakis, A., Garofalakis, M.N.: Complex event recognition in the big data era: a survey. VLDB J. **29**(1), 313–352 (2020)
21. Grez, A., Riveros, C., Ugarte, M.: A formal framework for complex event processing. In: ICDT, *LIPIcs*, vol. 127, pp. 5:1–5:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2019)
22. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation, 3rd Edition. Pearson international edition. Addison-Wesley (2007)
23. Laxman, S., Tankasali, V., White, R.W.: Stream prediction using a generative model based on frequent episodes in event sequences. In: KDD, pp. 453–461. ACM (2008)
24. Li, Y., Ge, T., Chen, C.: Data stream event prediction based on timing knowledge and state transitions. Proceedings of the VLDB Endowment **13**(10) (2020)
25. Li, Z., Ding, X., Liu, T.: Constructing narrative event evolutionary graph for script event prediction. In: IJCAI, pp. 4201–4207. ijcai.org (2018)
26. Makridakis, S., Spiliotis, E., Assimakopoulos, V.: Statistical and machine learning forecasting methods: Concerns and ways forward. PloS one **13**(3), e0194889 (2018)
27. Márquez-Chamorro, A.E., Resinas, M., Ruiz-Cortés, A.: Predictive monitoring of business processes: A survey. IEEE Trans. Services Computing **11**(6), 962–977 (2018)
28. Montgomery, D.C., Jennings, C.L., Kulahci, M.: Introduction to time series analysis and forecasting. John Wiley & Sons (2015)
29. Muthusamy, V., Liu, H., Jacobsen, H.: Predictive publish/subscribe matching. In: DEBS, pp. 14–25. ACM (2010)
30. Ozik, J., Collier, N., Heiland, R., An, G., Macklin, P.: Learning-accelerated discovery of immune-tumour interactions. Molecular systems design & engineering **4**(4), 747–760 (2019)
31. Pandey, S., Nepal, S., Chen, S.: A test-bed for the evaluation of business process prediction techniques. In: CollaborateCom, pp. 382–391. ICST / IEEE (2011)
32. Patroumpas, K., Alevizos, E., Artikis, A., Vodas, M., Pelekis, N., Theodoridis, Y.: Online event recognition from moving vessel trajectories. GeoInformatica **21**(2), 389–427 (2017)
33. Patroumpas, K., Spirelis, D., Chondrodima, E., Georgiou, H., P, P., P, T., S, S., N, P., Y, T.: Final dataset of Trajectory Synopses over AIS kinematic messages in Brest area (ver. 0.8) [Data set], 10.5281/zenodo.2563256 (2018). DOI 10.5281/zenodo.2563256. URL `http://doi.org/10.5281/zenodo.2563256`
34. Rabiner, L.R.: A tutorial on hidden markov models and selected applications in speech recognition. Proceedings of the IEEE **77**(2), 257–286 (1989)
35. Ray, C., Dreo, R., Camossi, E., Jousselme, A.: Heterogeneous Integrated Dataset for Maritime Intelligence, Surveillance, and Reconnaissance, 10.5281/zenodo.1167595 (2018). DOI 10.5281/zenodo.1167595. URL `https://doi.org/10.5281/zenodo.1167595`
36. Ron, D., Singer, Y., Tishby, N.: The power of amnesia. In: NIPS, pp. 176–183. Morgan Kaufmann (1993)
37. Ron, D., Singer, Y., Tishby, N.: The power of amnesia: Learning probabilistic automata with variable memory length. Machine Learning **25**(2-3), 117–149 (1996)
38. Van Der Aalst, W.: Process mining: discovery, conformance and enhancement of business processes. Springer (2011)
39. Veanes, M., de Halleux, P., Tillmann, N.: Rex: Symbolic regular expression explorer. In: ICST, pp. 498–507. IEEE Computer Society (2010)
40. Vilalta, R., Ma, S.: Predicting rare events in temporal domains. In: ICDM, pp. 474–481. IEEE Computer Society (2002)
41. Vouros, G.A., Vlachou, A., Santipantakis, G.M., Doulkeridis, C., Pelekis, N., Georgiou, H.V., Theodoridis, Y., Patroumpas, K., Alevizos, E., Artikis, A., Claramunt, C., Ray, C., Scarlatti, D., Fuchs, G., Andrienko, G.L., Andrienko, N.V., Mock, M., Camossi, E., Jousselme, A., Garcia, J.M.C.: Big data analytics for time critical mobility forecasting: Recent progress and research challenges. In: EDBT, pp. 612–623. OpenProceedings.org (2018)
42. Willems, F.M.J., Shtarkov, Y.M., Tjalkens, T.J.: The context-tree weighting method: basic properties. IEEE Trans. Information Theory **41**(3), 653–664 (1995)
43. Zhou, C., Cule, B., Goethals, B.: A pattern based predictor for event streams. Expert Syst. Appl. **42**(23), 9294–9306 (2015)