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

Complex Event Forecasting with Prediction Suffix Trees: a Formal Framework

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 is based on 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 show how these two formalisms can be combined in order to accurately and efficiently perform complex event forecasting. We compare our proposed approach against previous state-of-the-art methods and show the advantage of using prediction suffix trees, which, as a variable-order Markov model, have the ability to capture long-term dependencies in a stream by remembering only those past sequences that are informative enough. Our experimental results demonstrate the benefits, in terms of accuracy, of being able to capture such long-term dependencies. This is achieved by increasing the order of our model beyond what is possible with full-order Markov models that need to perform an exhaustive enumeration of all possible past sequences of a given order. We also discuss extensively how CEF solutions should be evaluated with respect to the quality of their forecasts.

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

This work was supported by the INFORE project, which has received funding from the European Union's Horizon 2020 research and innovation program, under grant agreement No 825070.

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

1 Introduction

The avalanche of real-time data in the last decade has sparked an interest in fields focused on processing high-velocity data streams. One of these fields which have enjoyed increased popularity is Complex Event Recognition (CER) [15,22]. The main goal of a CER system is to detect interesting activity patterns occurring within a real-time 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 optimization issues. Less attention has been paid to forecasting event patterns [22], despite the fact that forecasting has attracted considerable attention in various related research areas, such as time-series forecasting [30], sequence prediction [10,11,39,14,44], temporal mining [42,26,45,12] and process mining [29]. The need for Complex Event Forecasting

(CEF) has been acknowledged though, as evidenced by several conceptual proposals [21,13].

Consider, for example, the domain of credit card fraud management [9], where the detection of suspicious activity patterns of credit cards must occur with minimal latency that is in the order of a few milliseconds. The decision margin is extremely narrow. Being able to forecast that a certain sequence of transactions is very likely to be a fraudulent pattern provides wider margins both for decision and for action. For example, a processing system might decide to devote more resources and higher priority to those suspicious patterns to ensure that the latency requirement will be satisfied. The field of monitoring of moving objects (for ships at sea, aircrafts in the air or vehicles on the ground) provides yet another example where CEF could be a crucial functionality [43]. 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 [32]. 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, predict which of them will turn out to be non-pertinent and decide to terminate them at an early state could thus save valuable computational resources and significantly speed-up the execution of such in-silico experiments. It is important to 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. Our framework allows a user to define a pattern for a Complex Event, e.g., a pattern for fraudulent credit card transactions or 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, while consuming a stream of events, if and when a complex event is expected to occur. We use the formalism of symbolic automata [17] to encode a pattern and that of prediction suffix trees [39, 38] 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 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 experimental results, on both synthetic and real-world datasets, show the advantage of being able to use high order models over previous proposed baseline methods for CEF and methods based on low order 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 processed). The training time is also increased, but still remains within the same order of magnitude (typically tens of seconds for training datasets composed of hundreds of thousands of input events). This fact allows us to be confident that training could also be performed online, but we will explore this line of work in the future.

Our contributions may be summarized as follows:

- We present a CEF framework that is both formal and easy to use. It is often the case that CER frameworks lack clear semantics, which in turn leads to confusion about how patterns should be written and which operators are allowed [22]. This problem is exacerbated in CEF where a formalism for defining the patterns to be forecast may be lacking completely. Our framework is formal, compositional and as easy to use as writing regular expressions. The only basic requirement is that the user declaratively define a pattern and provide a training dataset.
- Our framework can uncover deep probabilistic dependencies in a stream (if such dependencies do exist) by using a variable-order Markov model. By being able to look deeper into the past, we can achieve higher accuracy scores compared to other state-of-the-art solutions for CEF.
- Our framework can perform various types of forecasting and thus subsumes previous methods that restrict themselves to one type of forecasting. It can perform both simple event forecasting (i.e., predicting what the next input event might be) and Complex Event forecasting (events defined through a pattern). Due to lack of space, we do not present experimental results on simple event forecasting, but it must be stressed that moving from simple event to Complex Event forecasting is not trivial. Using simple event forecasting to project in the future the

most probable sequence of input events and then attempt to detect Complex Events on this future sequence yields sub-optimal results. A system that can perform simple event forecasting cannot thus be assumed to perform CEF as well.
– We also discuss the issue of how the forecasts of a CEF system may be evaluated with respect to their quality. Previous methods have used metrics borrowed from time-series forecasting (e.g., the root mean square error) or typical machine learning tasks (e.g., precision). We propose a more comprehensive set of metrics that takes into account the idiosyncrasies of CER and CEF. Besides accuracy itself, the usefulness of forecasts is also judged by their "earliness". We also discuss how the notion of earliness may be quantified.

1.1 Running Example

We now present the general idea behind CER systems, along with an example that we will use throughout the rest of the paper to make our presentation more accessible. The input to a CER system consists of two main components: a stream of events, also called simple derived events (SDEs); a set of patterns that define relations among the SDEs and must be detected upon the input stream. Instances of pattern satisfaction are called Complex Events (CEs). The output of the system is another stream, composed of the detected CEs. It is typically required that CEs must be detected with very low latency, which, in certain cases, may even be in the order of a few milliseconds [28,19,24].

Table 1: Example stream from the maritime domain.

status	vessel id	speed	timestamp
fishing	112	2	1
fishing	112	1	2
fishing	112	3	3
under way	112	22	4
under way	112	19	5
under way	112	27	6
...	...	...	...

As an example, 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, 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 identifier. Table 1 shows a possible stream of AIS messages, where we only include the *speed* attribute and the *timestamp* is shown as a sequence of increasing indices. An analyst 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 restricted (e.g., NATURA) areas, etc. The typical workflow consists of the analyst first writing these patterns in some (usually) declarative language, which are then used by a computational model applied on the stream of SDEs to detect CEs.

1.2 Structure of the Paper

We start by presenting in Section 2 the relevant literature on 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. We then discuss how we can quantify the quality of produced forecasts in Section 5. We finally demonstrate the efficacy of our framework in Section 6, by showing experimental results on two datasets. We conclude with Section 7, discussing some possible directions for future work. The paper assumes a basic familiarity with automata theory, logic and Markov chains.

2 Related Work

Forecasting has not received much attention in the field of CER, although some conceptual proposals have acknowledged the need for CEF [21,18,13]. The first concrete attempt at CEF was presented in [31]. A variant of regular expressions is used to define CE patterns, which are then compiled into automata. These automata are then translated to Markov chains through a direct mapping, where each automaton state is mapped to a Markov chain state. Frequency counters on the transitions are then used to estimate the Markov chain's transition matrix. This Markov chain is finally used to estimate if a CE is expected to occur within some future window. As we explain in Section 4.2, in the worst case, such an approach assumes that all SDEs are independent and is thus unable to encode higher order dependencies.

Another example of event forecasting is presented in [5]. Using Support Vector Regression, the proposed

method is able to predict the next input event(s) within some future window. This technique is more similar to time-series forecasting [30], as it mainly targets the prediction of the (numerical) values of the attributes of the input 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. The idea is put forward that these future SDEs may be used by a CER engine to detect future CEs. 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 fails to include the "paths" that lead to a CE detection.

In [33], Hidden Markov Models (HMM) are used to construct a probabilistic model for the behavior of a transition system. The observation 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. In principle, HMMs are more powerful than Markov chains. In practice, however, HMMs are hard to train and require elaborate domain knowledge, since mapping a CE pattern to a HMM is not straightforward (see Section 4.2 for more details). Our approach is able to seamlessly construct a probabilistic model from a given CE pattern (declaratively defined), without requiring extensive domain knowledge.

Automata and Markov chains are again used in [6, 7]. The main difference of these methods compared to [31] is that they can accommodate higher order dependencies by creating extra states for the automaton of a pattern and its Markov chain. As far as [6] is concerned, it has two important limitations: first, it works only on discrete sequences of finite alphabets; second, although theoretically possible, it is practically infeasible to increase the order of the Markov chain beyond a certain point, since 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. We show how this problem can be addressed by using variable-order Markov models.

A different approach is followed in [27], where knowledge graphs are used to encode events and their timing relationships. Stochastic gradient descent is employed in order to learn the weights of the graph's edges that determine how important an event is with respect to another target event. However, this approch falls in the category of SDE forecasting, as it does not target complex events. Instead, it attempts to forecast simple events. More precisely, it tries to forecast which predicates the next simple events will satisfy without taking into account relationships between the events themselves (e.g., through simple sequences).

3 Complex Event Recognition with Symbolic Automata

Before presenting our approach for CEF, we begin by first presenting a formal framework for CER. For surveys of CER, please see [15,8,22]. As can be deduced from these surveys, there is an abundance of CER systems and languages. One issue, however, is that 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 [22] and [23] which have established some core operators that are most often used. In a spirit similar to [23], we use automata as our computational model and define a CER language whose expressions can readily be converted to automata. Instead of choosing one of the automaton models already proposed in the CER literature, we employ symbolic regular expressions and automata [17, 16,41]. The rationale behind our choice is that, contrary to other automata-based CER models, symbolic expressions and automata have nice closure properties and clear, compositional semantics (see [23] for a similar line of work, based on symbolic transducers).

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 [17]. 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 for an effective Boolean algebra is the following:

Definition 1 (Effective Boolean algebra [17]) An effective Boolean algebra is a tuple ($\mathcal{D}$, Ψ, $[\![\_]\!]$, $\bot$, $\top$, $\vee$, $\wedge$, $\neg$) where $\mathcal{D}$ is a set of domain elements; Ψ is a set of predicates closed under the Boolean connectives; $\bot, \top \in \Psi$; the component $[\![\_]\!] : \Psi \rightarrow 2^{\mathcal{D}}$ is a denotation function such that

- $[\![\bot]\!] = \emptyset$

- $[\![\top]\!] = \mathcal{D}$
- 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 effectively computable.

Using our running example, such an algebra could be one consisting of two predicates about the speed level of a vessel, e.g., *speed* < 5 and *speed* > 20, along with their combinations constructed from the Boolean connectives, e.g., $\neg(speed < 5) \wedge \neg(speed > 20)$.

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 $\mathcal{D}$.

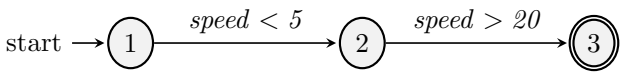
As with classical regular expressions [25], we can use symbolic regular expressions to represent a class of languages over $\mathcal{D}$.

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

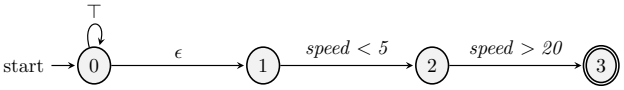
- The constants ϵ and $\emptyset$ are symbolic regular expressions with $\mathcal{L}(\epsilon) = \{\epsilon\}$ and $\mathcal{L}(\emptyset) = \{\emptyset\}$;
- If $\psi \in \Psi$, then $R := \psi$ is a symbolic regular expression, with $\mathcal{L}(\psi) = [\![\psi]\!]$, i.e., the language of ψ is the 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 $L(R^*) = (L(R))^*$, where $L^* = \bigcup_{i \geq 0} L^i$ and L^i is the concatenation of L with itself i times.

As an example, if we want to detect instances of a vessel accelerating suddenly, we could write the expression $R := (speed < 5) \cdot (speed > 20)$. The third and fourth events of the stream of Table 1 would then belong to the language of R.

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



(a) *SFA* corresponding to the *SRE* $R := (speed < 5) \cdot (speed > 20)$.



(b) Streaming *SFA* corresponding to $R := (speed < 5) \cdot (speed > 20)$.

Fig. 1: Examples of a symbolic automaton and streaming symbolic automaton.

Definition 3 (Symbolic finite automaton [17]) A symbolic finite automaton (*SFA*) is a tuple $M = (\mathcal{A}, Q, q^s, F, \Delta)$, 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 $q_k \in Q^f$. We refer to the set of strings accepted by M as the language of M, denoted by $\mathcal{L}(M)$ [17]. Figure 1a shows a *SFA* that can detect the expression of sudden acceleration for our running example.

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. The proof is essentially the same as that for classical expressions and automata [25]. See Appendix, Section A.1. □

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 we need to handle continuously updated streams of events and detect instances of *SRE* satisfaction as soon as they appear in a stream. For example, the automaton of the (classical) regular expression $a \cdot b$ would accept only the string a, b. In a streaming setting, we would like the automaton to report a match every time this string appears in a stream. For the stream a, b, c, a, b, c, we would thus expect two matches to be reported, one after the second symbol

and one after the fifth. In order to accommodate this scenario, slight modifications are required so that *SRE* and *SFA* may work in a streaming setting. First, we note that 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 [23]. 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*) corresponding to R). Using R_s we can detect CEs of R while consuming a stream S, since a stream segment $S_{m..k}$ is recognized by R iff the prefix $S_{1..k}$ is recognized by 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 $R' := \top^* \cdot (speed < 5) \cdot (speed > 20)$. Then, after consuming the fourth event of the stream of Table 1, $S_{1..4}$ would belong to the language of $\mathcal{L}(R)$ and $S_{3..4}$ to the language of $\mathcal{L}(R')$. 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* as well. Figure 1 shows how a *sSFA* can be constructed from $R := (speed < 5) \cdot (speed > 20)$.

The streaming behavior of a *sSFA* as it consumes a stream S can be formally described through 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 $c' = [i', q']$ a successor 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 $[1, q_1 = M.q^s] \xrightarrow{\delta_1} [2, q_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_k} [k+1, q_{k+1}]$. A run is called accepting iff $q_{k+1} \in M.Q^f$.

We can see that 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. Therefore, for a *sSFA* that consumes 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.

4 Complex Event Forecasting with Prediction Suffix Trees

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 and this *sSFA* could be used to perform recognition. In order to be able to perform event forecasting, we use this *sSFA* to construct 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. We can convert an automaton to a Markov chain if we first determinize it. 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 when the *DSFA* is expected to reach a final state or, equivalently, about when a CE defined by R is expected to occur. The issue that we address in this paper is how to build a model which retains only meaningful long-term dependencies.

4.1 Deterministic Symbolic Automata

The definition of *DSFA* is similar to that for 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, since two predicates may be different but still both evaluate to TRUE for the same tuple. Therefore, the formal definition for *DSFA* must take this into account:

Definition 6 (Deterministic SFA [17]) 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 [17]. 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 $N = Minterms(Predicates(M))$, and then use these minterms as guards for the *DSFA* [17].

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 for two reasons: 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 [17]. For example, if $Predicates(M) = \{\psi_1, \psi_2\}$, 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, which, in our case, is 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 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 $N = Minterms(Predicates(M))$. 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 prove the lemma, which states that for every *DSFA* there exists an equivalent classical deterministic automaton.

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 $N = Minterms(Predicates(M_s))$ to $\mathcal{L}(M_s)$.*

Proof. From an algebraic point of view, the set $N = Minterms(Predicates(M))$ may be treated as a generator of the monoid N^*, with concatenation as the operation. If the cardinality of N is k, then we can always find a set $\Sigma = \{a_1, \cdots, a_k\}$ of k distinct symbols and then a morphism (in fact, an isomorphism) $\phi : N^* \rightarrow \Sigma^*$ that maps each minterm to exactly one, unique a_i. A classical *DFA* M_c can then be constructed by relabeling the *DSFA* M_s under ϕ, i.e., by copying/renaming the states and transitions of the original *DSFA* M_s and by replacing the label of each transition of M_s by the image of this label under ϕ. Then, the behavior of M_c (the language it accepts) is the image under ϕ of the behavior of M_s [40]. Or, equivalently, the language of M_c is the set of strings induced by applying $N = Minterms(Predicates(M_s))$ to $\mathcal{L}(M_s)$. □

A direct consequence drawn from the proof of the above lemma is that, for every run $\varrho = [1, q_1] \xrightarrow{\delta_1} [2, q_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_k} [k+1, q_{k+1}]$ followed by a *DSFA* M_s by consuming a symbolic string (stream of tuples) S, the run that the equivalent *DFA* M_c follows by consuming the induced string S' is also $\varrho' = [1, q_1] \xrightarrow{\delta_1} [2, q_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_k} [k+1, q_{k+1}]$, i.e., M_c follows the same copied/renamed states and the same copied/relabeled transitions.

This direct relationship between *DSFA* and classical *DFA* allows us to transfer techniques developed for classical *DFA* to the study of *DSFA*. Moreover, we can simplify our notation by employing the terminology of symbols/characters and strings/words that is typical for classical automata. From now on, 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*. For example, the symbol a may actually refer to the minterm $\psi_1 \wedge \psi_2$, the symbol b to $\psi_1 \wedge \neg\psi_2$, etc.

4.2 Variable-order Markov Models

Assuming that we have a deterministic automaton, the next question is how we can build a Markov chain that describes its (probabilistic) behavior so that we can then make inferences about this behavior. One approach would be to map each state of the automaton to a state of a Markov chain, then feed a training stream of symbols to the automaton, count the number of transitions

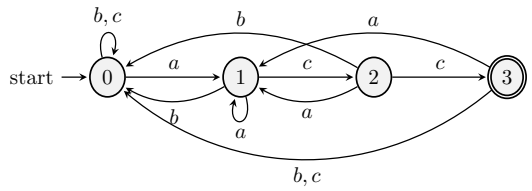


Fig. 2: A classical automaton for the expression $R := a \cdot c \cdot c$. State 1 can always remember the last symbol seen, since it can be reached only with a, whereas state 0 can be reached either with b or c.

from each state to every other target state and use these counts to calculate the transition probabilities. This is the approach followed in [31]. One important question with respect to this approach concerns the order of the states of the Markov chain, i.e., how deep into the past a state can look when calculating its transition probabilities. The answer is that this order essentially depends on the structure of the pattern R defining the automaton and not on the training data. As an example, consider a state of an automaton whose incoming transitions all have a as a symbol/guard. This state implicitly encodes the fact that the last symbol consumed is always a. The transition probabilities from this state are thus conditional on a and the order of the state can be said to be 1. In Figure 2, state 1 is such a state, since all its incoming transitions are labeled with a. If, however, there exists at least one other incoming transition whose symbol is not a, then the state can no longer implicitly remember the last seen symbol and its order collapses to 0. This is the case of state 0 in Figure 2, which we can reach by seeing either b or c. As a result, with this approach, there is no guarantee that dependencies may be captured. In the worst case, the order can be 0 for all states, thus essentially assuming 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.

Our proposed approach uses a variable-order Markov model (VMM) to mitigate the high cost of increasing the order m [10,11,39,38,14,44]. As a result, we can 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) [36], which are generally more expressive than bounded-order (either full or variable) Markov models, since they can encode the whole past of a sequence. However, HMMs ofter require large training datasets [10,4]. Another problem is that it is not always obvious how a domain can be modelled through HMMs and a deep understanding of the domain may be required [10]. Consider, for example, our case of automata-based CER. Even with the previous approaches mentioned above, there is a natural way to map an automaton to a Markov chain, whereas the relation between an automaton and the observed state of a HMM is not straightforward.

Various Markov models of variable order have been proposed in the past. For a nice comparative study, see [10]. Let Σ denote an alphabet, $\sigma \in \Sigma$ a symbol from that alphabet and $s \in \Sigma^m$ a string from that alphabet of length m. The general idea is to derive a predictor $\hat{P}$ from the training data such that the average log-loss with respect to a test sequence $S_{1..k}$, given by $l(\hat{P}, S_{1..k}) = -\frac{1}{T}\sum_{i=1}^{k} log\hat{P}(t_i \mid t_1 \cdots t_{i-1})$, is minimized. This 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 of the test sequence [10]. The mean 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 learning conditional distributions $\hat{P}(\sigma \mid s)$, where m is constant and equal to the assumed order of the Markov model. VMMs, on the other hand, learn such conditional distributions by relaxing the assumption of m being fixed. The length of the "context" 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 [39,38], as our VMM of choice. The reason is that, after a PST has been learned, it can be readily converted to a probabilistic automaton, which, as we will show, can then be combined with a symbolic automaton. More precisely, we learn a probabilistic suffix automaton (PSA), whose states correspond to contexts of variable length and the outgoing transitions from

each state encode the conditional distribution of seeing a symbol given the context of that state. This *PSA* is then embedded into each state of the *DSFA* of a pattern R, which then allows us to infer when the *DSFA* will reach one of its final states, taking into account the statistical properties of the stream, as encoded into the *PSA*.

The formal definition of a PST is the following:

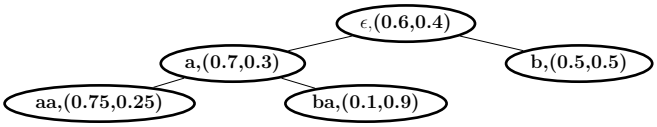
Definition 8 (Prediction Suffix Tree [39]) Let Σ be an alphabet. A PST T over Σ is a tree whose edges are labeled by symbols $\sigma \in \Sigma$ and each internal node has exactly one edge for every $\sigma \in \Sigma$ (hence, the degree is $\mid \Sigma \mid$). Each node is labeled by a pair (s, γ_s), where s is the string associated with the walk starting from that node and ending in 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, $\sum_{\sigma \in \Sigma} \gamma_s(\sigma) = 1$.

Figure 3a shows an example of a *PST* for $m = 2$. According to this tree, if the last symbol/minterm that we have encountered in a stream is a, 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, then we do not need to look deeper into the past. The next symbol probability distribution remains the same, $(0.5, 0.5)$.

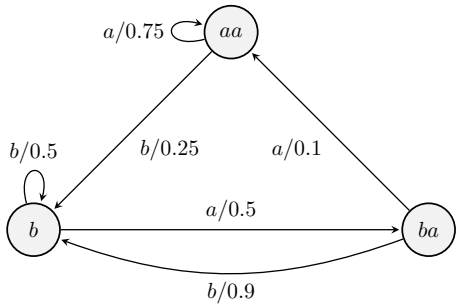
Our goal is to incrementally learn a *PST* $\hat{T}$ by adding new nodes only when it is necessary and then use $\hat{T}$ to construct a *PSA* $\hat{M}$ that would be close enough to the actual *PSA* M that has generated the training data. The learning algorithm in [39] starts with a tree having only 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 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) (*suffix*(s) is the longest suffix of s that is different from s).

The thresholds θ_1 and θ_2 depend, among others, on an approximation parameter α, measuring how close we want the estimated *PSA* $\hat{M}$ to be compared to the actual *PSA* M, on n, denoting the maximum number of



(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 tree T. 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.

states that we allow $\hat{M}$ to have and on 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 in learning where we have not yet added any of its children, aa and ba. We now want to check whether it is meaningful to add ba as a node. We can thus 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, which leads us to add node ba to the tree. For more details, see [39].

After 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 [39]) 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';

- 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 a 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 consume 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.

4.4 Embedding of a *PSA* in a *DSFA*

Our final goal is to use the statistical properties of a stream, as encoded in a *PSA*, in order to be able to infer when a CE of a given *SRE* R will be detected. Equivalently, we are interested in inferring when the *SFA* of R will have reached one of its final states. To achieve this goal, we work in the following way. We are initially given a *SRE* R along with 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 any probabilistic inferences. Our next step is to use the minterms of M_R (acting as "symbols", see Lemma 1) and the training stream S to learn a *PSA* M_S which encodes the statistical properties of S, but has no knowledge of the structure of R (it only knows its minterms), is not an acceptor and cannot be used for recognition. At this point, we have two different automata, M_R for recognition, and M_S, describing the properties of the training dataset. We can then combine M_R and M_S into a single automaton M that has the power of both M_R and M_S and can be used both for recognition and for inferring, according to the properties of S, when a CE of R will be detected. We call M the embedding of M_S in M_R. Its formal definition is given below, where, in order to simplify notation, we use Lemma 1 so that a *DSFA* is represented as a classical deterministic automaton.

Definition 10 (Embedding of a *PSA* in a *DSFA*) Let M_R be a *DSFA* (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 $M = (Q, Q^s, Q^f, \Sigma, \Delta, \Gamma)$, 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: 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).

The first condition ensures that all automata have the same alphabet. The second condition ensures that M is equivalent to M_R by having the same language. The third condition 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 learned with the minterms of M_R, there exists an embedding of M_S in M_R.*

Proof. Construct an embedding in the following straightforward manner: First let the product $M_R.Q \times M_S.Q$ be its states, i.e., for every $q \in Q$, $q = (r, s)$ and $r \in M_R.Q$, $s \in M_S.Q$. Set the initial states of M as follows: for every $q = (r, s)$ such that $r = M_R.q^s$, set $q \in Q^s$. Similarly, for the final states, for every $q = (r, s)$ such that $r \in M_R.Q^f$, set $q \in Q^f$. Then let the transitions of M be defined as follows: A transition $\delta((r, s), \sigma) = (r', s')$ is added to M if there exists a transition $\delta_R(r, \sigma) = r'$ in M_R and a transition $\tau(s, \sigma) = s'$ in M_S. Let also Γ

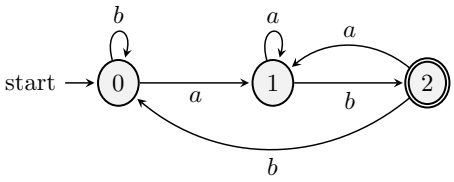
be defined as follows: $\Gamma((r,s),\sigma) = \gamma(s,\sigma)$. Finally, for the initial state distribution, we set:

$$\pi((r,s)) = \begin{cases} M_S.\pi(s) & \text{if } r = M_R.q^s \\ 0 & \text{otherwise} \end{cases}$$

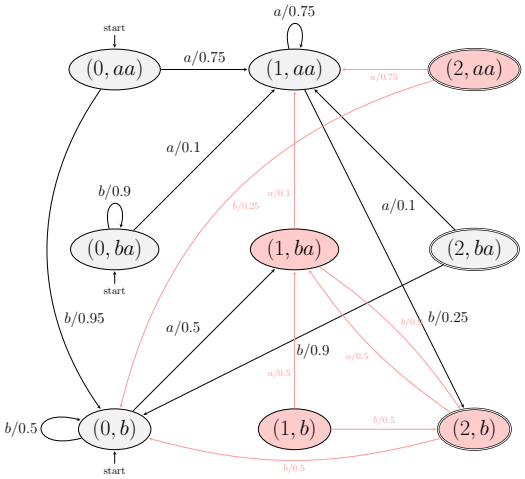
Proving that $\mathcal{L}(M) = \mathcal{L}(M_R)$ is done with induction on the length of strings. The inductive hypothesis is that, for strings $S_{1..k} = t_1 \cdots t_k$ of length k, if $q = (r,s)$ is the state reached by M and q_R the state reached by M_R, then $r = q_R$. Note that both M_R and M are deterministic and complete automata and thus only one state is reached for every string (only one run exists). If a new element t_{k+1} is read, M will move to a new state $q^{'} = (r^{'}, s^{'})$ and M_R to $q_R^{'}$. From the construction of the transitions of M, we see that $r^{'} = q_R^{'}$. Thus, the induction hypothesis holds for $S_{1..k+1}$ as well. It also holds for $k = 0$, since, for every $q = (r,s) \in Q^s$, $r = M_R.q^s$. Therefore, it holds for all k. As a result, if M reaches a final state (r,s), r is reached by M_R. Since $r \in M_R.Q^f$, M_R also reaches a final state.

For proving probabilistic equivalence, first note that the probability of a string given by a predictor P is $P(S_{1..k}) = \prod_{i=1}^{k} P(t_i \mid t_1 \dots t_{i-1})$. Assume now that a *PSA* M_S reads a string $S_{1..k}$ and follows a run $\varrho = [l, q_l] \xrightarrow{t_l} [l+1, q_{l+1}] \xrightarrow{t_{l+1}} \cdots \xrightarrow{t_k} [k+1, q_{k+1}]$. We define a run in a manner similar to that for runs of a *DSFA*. The difference is that a run of a *PSA* may begin at an index $l > 1$, since it may have to wait for l symbols before it can find a state q_l whose label is equal to $S_{1..l}$. We also treat the *PSA* as a reader (not a generator) of strings for which we need to calculate their probability. The probability of $S_{1..k}$ is then given by $P_{M_S}(S_{1..k}) = M_S.\pi(q_l) \cdot \prod_{i=l}^{k} M_S.\gamma(q_i, t_i)$. Similarly, for the embedding M, assume it follows the run $\varrho^{'} = [l, q_l^{'}] \xrightarrow{t_l} [l+1, q_{l+1}^{'}] \xrightarrow{t_{l+1}} \cdots \xrightarrow{t_k} [k+1, q_{k+1}^{'}]$. Then, $P_M(S_{1..k}) = M.\pi(q_l^{'}) \cdot \prod_{i=l}^{k} M.\Gamma(q_i^{'}, t_i)$. Now note that M has the same initial state distribution as M_S, i.e., the number of the initial states of M is equal to the number of states of M_S and they have the same distribution. With an inductive proof, as above, we can prove that whenever M reaches a state $q = (r,s)$ and M_S reaches q_S, $s = q_S$. As a result, for the initial states of M and M_S, $M.\pi(q_l^{'}) = M_S.\pi(q_l)$. From the construction of the embedding, we also know that $M_S.\gamma(s,\sigma) = M.\Gamma(q,\sigma)$ for every $\sigma \in \Sigma$. Therefore, $M_S.\gamma(q_i, t_i) = M.\Gamma(q_i^{'}, t_i)$ for every i and $P_M(S_{1..k}) = P_{M_S}(S_{1..k})$. □

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 correspond to minterms. Thus, Σ is the set of 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 that would be created, following the construction procedure of the proof of Theorem 1. Notice, however, that this embedding has some redundant states and transitions. The red states 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. Note that all incoming transitions to state 1 of M_R have a as their symbol. Similarly, state 2 has only one transition with b as its symbol. Therefore, there is no point in merging state 1 of M_R with all the states of M_S, but only with state b. If we follow a straightforward construction, as described above, the result will be the automaton depicted in Figure 4b, including the redundant red states. To avoid the inclusion of such states, we can merge M_R and M_S in an incremental fashion. The resulting automaton would then consist only in the black states and transitions of Figure 4b.



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



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

Fig. 4: Embedding example.

4.5 Emitting Forecasts

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 initial expression R as those of M_R. Our goal is to forecast, after every event, when M will reach one of its final states. More precisely, we want to estimate the number of transitions from any state M might be in 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):

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

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 in two sets, such as N and F, the following theorem can then 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*

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

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:

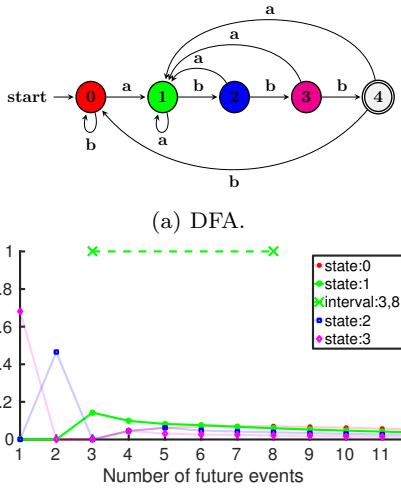
$$P(Y_n \in F, Y_{n-1} \notin F, \cdots, Y_1 \in F \mid \boldsymbol{\xi_{init}}) = \begin{cases} \boldsymbol{\xi_F}^T \boldsymbol{F}\boldsymbol{1} & \text{if } n = 2 \\ \boldsymbol{\xi_F}^T \boldsymbol{F_N} \boldsymbol{N}^{n-2}(\boldsymbol{I} - \boldsymbol{N})\boldsymbol{1} & \text{otherwise} \end{cases}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

Proof. The proof for Eq. 2 may be found in [20]. The proof for Eq. 3 may be found in the Appendix, Section A.2. □

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:

$$W_q = inf\{n : Y_0, Y_1, ..., Y_n, Y_0 = q, q \in Q \backslash F, Y_n \in F\}$$

Note, however, that Theorem 2 provides us with a way to calculate the probability of reaching a final state, given an initial state distribution $\boldsymbol{\xi_{init}}$. In our case, where we have an automaton moving through its various states, $\boldsymbol{\xi_{init}}$ has a special form. As the automaton consumes a stream, it always finds itself (with certainty) in a specific state q. As a result, for each state q that the automaton might find itself in, $\boldsymbol{\xi_{init}}$ is a vector with all of its elements being equal to 0, except for the element corresponding to the current state of the automaton, which is equal to 1.



(a) DFA.

(b) Waiting-time distributions and smallest interval exceeding $\theta_{fc} = 50\%$ for state 1.

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

Figure 5 shows an example of an automaton (its exact nature is not important, as long as it can also be described as a Markov chain), along with the waiting-time distributions for its non-final states. For this example, if the automaton is in state 2, then the probability of reaching the final state 4 for the first time in 2 transitions is $\approx 50\%$ but 0% for 3 transitions (the automaton has no path of length 3 from state 2 to state 4).

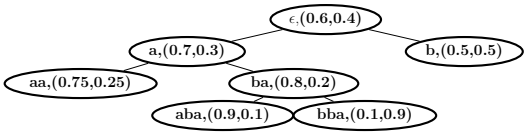
We can use the waiting-time distributions to produce various kinds of forecasts. In the simplest case, we can scan a distribution in order to find the point with the highest probability and return this point as a forecast. Alternatively, we may want to know whether a CE will occur within the next w input events. In this case, 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. These kinds of forecasts are easy to compute.

4.6 Avoiding the Construction of the Markov Chain

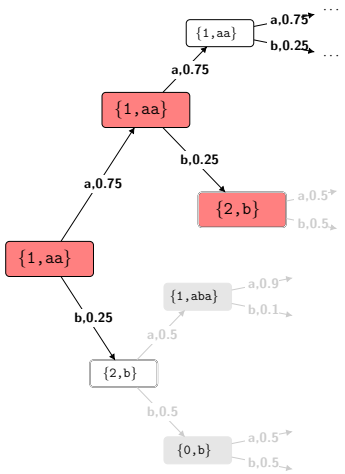
The reason for constructing an embedding, as described above, is that it is based on a variable-order model and the expectation is that it will thus consist of much fewer states than a full-order model. In practice, however, we have observed that the gains from creating an embedding are not as significant as we would expect. Although the number of states of an embedding may indeed be smaller (even up to 50%), it is often still in the same order of magnitude as that of a full-order model. A reduction in the number of states that can reach up to 50% might seem significant, but the fact that the order of magnitude remains the same means that such a reduction is of little use for our purposes. For example, if a full-order model requires 1 million states and is thus impossible to build, reducing this number to 500.000 states makes no difference, since we will still be unable to handle so many states.

Upon closer inspection, we identified one specific step in the process of creating an embedding that acts as the main bottleneck: the step of converting a *PST* to a *PSA*. The difference between the number of nodes of a *PST* and the number of states of the *PSA* constructed from that *PST* has a range that covers several orders of magnitude. 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*. 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, without using Theorem 2, in the following manner.

As we consume events from the input stream, besides feeding them to M_R, we also feed them to a buffer that always holds the last m events from the stream, where m is equal to the maximum order of T. After consuming an event and feeding it to this buffer, we traverse T according to its contents. This traversal leads us to a leaf l of T. We are thus in a position to estimate the probability of any future sequence of events. If $S_{1..k} = \cdots, t_{k-1}, t_k$ is the stream that we have seen, then the next symbol probability for t_{k+1} can be directly retrieved from the distribution of the leaf l ($P(t_{k+1} \mid t_{k-m+1}, \cdots, t_k)$). If we want to look further into the future, e.g., into t_{k+1}, t_{k+2}, we can iterate this process as many times as necessary. The probability for t_{k+2}, $P(t_{k+2} \mid t_{k-m+2}, \cdots, t_{k+1})$, can again be retrieved from T, by finding the leaf l' that is reached with $t_{k+1}, \cdots, t_{k-m+2}$. We can thus find the probability of any future sequence of events. As a result, we can also find 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. 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.



(a) Prediction suffix tree 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. Red nodes correspond to the only path of length $k = 2$ that leads to a final state.

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

Figure 6 shows an example of this process for the automaton M_R of Figure 4a. Figure 6a 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 were the only step, the resulting *PSA* would always have fewer states than the *PST*. As this example shows, this is not the case. Imagine that our states 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 over-

come this ambiguity regarding the correct next state, we would have to first 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 caused by such expansions far outweigh the number of the original leaves. As a result, the size of the *PSA* is far greater than that of the original, unexpanded *PST*.

Figure 6b shows how we can estimate the probability for any future sequence of states of M_R, using the distributions of T. We 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 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 initial node, 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. How can we use it to estimate the waiting-time distribution for our initial node $\{1, aa\}$? The estimation is actually simple. If we want to estimate the probability of reaching a final state for the first time in k transitions, we first find all the paths starting from the original node, having length k, ending in a final state and without another final state at a level below k. 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 red nodes show the path the leads to a final state after 2 transitions. Its probability is $0.75 * 0.25 = 0.1875$, since we just need to multiply the probabilities of the edges. If there were more such paths, we would have to add all their probabilities. Thus, $P(W_{\{1,aa\}} = 2) = 0.1875$.

It is obvious that this tree grows exponentially as we try to look deeper into the future. This cost can be significantly reduced by employing some simple optimizations. First, note in Figure 6b, that the paths starting from the $\{2, b\}$ nodes are grayed out. This indicates that these nodes need not be expanded, since they correspond to final states and any paths starting from them will not be used again. 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 in Figure 6b are not useful. With this optimization, we can still do an exact estimation of the waiting-time distribution. Another optimization that we have found to be very useful tries to prune paths that should normally be expanded (no final state is involved). The intuition is that a path with a very low probability will not contribute significantly to the probabilities of our waiting-time distribution, even if we do expand it. We can thus prune such paths, accepting the risk that we will have an approximate estimation of the waiting-time distribution. Although this is an ad hoc optimization, we have found it to be very efficient while having a negligible impact on the distribution for a wide range of cut-off thresholds. In the future, we intend to explore more rigorous ways for performing such an approximate estimation of our waiting-time distributions.

4.7 Complexity Analysis

We have thus far described how an embedding of a *PSA* M_S in a *DSFA* M_S can be constructed and how we can estimate the forecast intervals for this embedding. We have also presented an optimization that bypasses the construction of a *PSA* and can estimate forecasts directly via a *PST*. In this section, we give some results about the complexity of each of the steps involved. Before doing so, we need to describe one more step that is required. In [39], it is assumed that, before learning a *PST*, the empirical probabilities of symbols given various contexts are available. The suggestion in [39] is that these empirical probabilities can be calculated either by repeatedly scanning the training stream or by using a more time-efficient algorithm that keeps pointers to all occurrences of a given context in the stream. We opt for a variant of the latter choice. First, note that the empirical probabilities are given by the following formulas [39]:

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

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

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

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

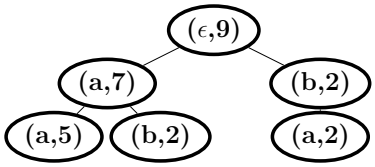


Fig. 7: Example of a Counter Suffix Tree with $m = 2$ and $S = aaabaabaaa$.

In other words, we need to count the number of occurrences of the various candidate strings s in $S_{1..k}$.

In order to estimate these counters, we can use a tree data structure which allows us 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. The counter of every node is equal to the sum of the counters of its children. 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 and σ_n to the symbol of the node that is reached. The property that we maintain as we build a *CST* from a stream $S_{1..k}$ is that the counter of the node 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 $S = aaabaabaaa$. If we want to retrieve the number of occurrences of the string $b \cdot a$ in S, we can first take the left child of the root node and then the right child of $(a, 7)$. 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 elements of S. The contents of the buffer are fed into the *CST* after every new element is read. The *CST* starts following a path according to the 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 set to 1. After the *CST* has read the training stream, it can be used to retrieve the necessary counters, as per Equation 6, and estimate the empirical probabilities of Equations 4 and 5.

With the addition of this step for constructing a *CST*, we have a total of four steps required for building an embedding from an initial *DSFA* M_R and a training stream S. We additionally have two more steps in order to derive the final forecast intervals. Figure 8 depicts these steps as a process, along with the input required for each of them. 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. In the next 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 in the next step. The third step converts the *PST* to a *PSA*, by using the leaves of the *PST* as states of the *PSA*. This *PSA* is then merged with the initial *DSFA* to create the embedding of the *PSA* in the *DSFA*. From the embedding we can calculate the waiting-time distributions and these can be used to derive the forecast intervals, using the confidence threshold θ_{fc} provided by the user. Figure 8 also shows the alternative option of bypassing the construction of the *PSA*, by estimating the waiting-time distributions directly from the *PST*.

The learning algorithm of the second step, as presented in [39], is polynomial in m, n, $\frac{1}{\alpha}$ and the size of the alphabet (number of minterms in our case). The complexity of estimating the waiting-time distributions from the transition matrix of a Markov chain depends highly on the library used for matrix calculations and especially on the complexity of raising a matrix to the powers of $n-1$ and $n-2$ in Equations 2 and 3. A straightforward way to multiply a matrix $\boldsymbol{N}$ by itself would require N multiplications and $N-1$ additions for each element of the final matrix, where $N \times N$ is the number of elements of $\boldsymbol{N}$. Thus, a total of $N^3(N-1)$ operations would be required for one matrix multiplication. For raising $\boldsymbol{N}$ to the power of $n-1$, a total of $(n-2)N^3(N-1)$ operations would be required, assuming that $\boldsymbol{N}^k$ is calculated by multiplying $\boldsymbol{N}$ by $\boldsymbol{N}^{k-1}$, estimated at a previous step. However, efficient libraries can reduce this cost significantly, especially for sparse matrices, as would be typical for the matrix of an embedding. Below, we give complexity results for the four remaining steps of the main route to estimating the forecasts (steps 1,3,4 and 6 in Figure 8). We also give complexity results for the alternative option that bypasses the *PSA* (step 3′).

Proposition 2 (Step 1 in Figure 8) *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 $O(m(k-m))$.*

Proof. See Appendix, Section A.3.1. □

Proposition 3 (Step 3 in Figure 8) *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 $O(t^{m+1} \cdot m)$.*

Proof. See Appendix, Section A.3.2. □

Proposition 4 (Step 4 in Figure 8) *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 $O(t \cdot |M_R.Q \times M_S.Q|)$.*

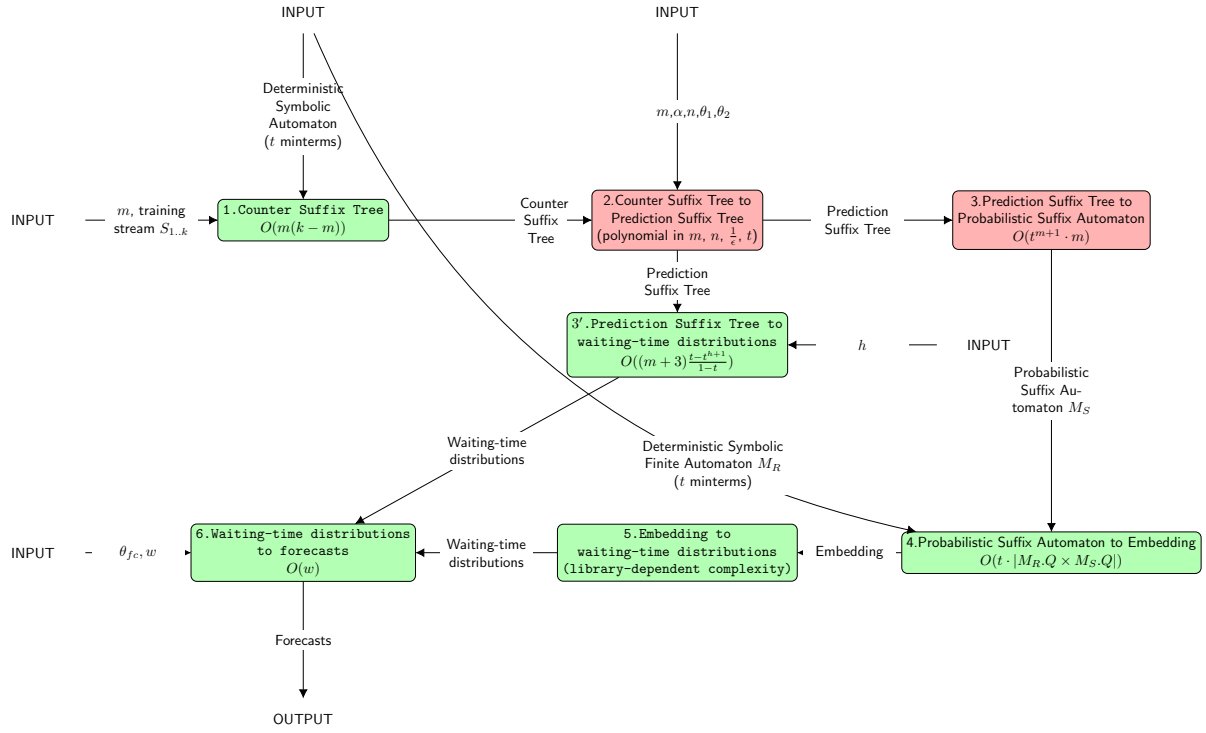


Fig. 8: Steps for calculating the forecast intervals of a *DSFA*. m is the maximum assumed order, α the approximation parameter, n the maximum number of states for the *PSA* and θ_{fc} the confidence threshold of the intervals. Red blocks indicate steps described in [39]. Green blocks indicate steps described in this paper.

Proof. See Appendix, Section A.3.3. □

Notice that the cost of learning a *PSA* might be exponential in m. In the worst case, all permutations of the t minterms of length m need to be added to the *PST* and the *PSA*. This may happen if the statistical properties of the training stream are such that all these permutations are deemed as important. In this case, the final embedding in the *DSFA* M_R might have up to $t^m \cdot |M_R.Q|$ states. This is also the upper bound of the number of states of the automaton the would be created using the method of [6], where every state of an initial automaton is split into at most t^m sub-states, regardless of the properties of the stream. Thus, in the worst case, our approach would create an automaton of size similar to an automaton encoding a full-order Markov chain. Our approach provides an advantage when the statistical properties of the training stream allow us to retain only some of the dependencies of order up to m.

Proposition 5 (Step 3′ in Figure 8) *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 $O((m+3)\frac{t-t^{h+1}}{1-t})$.*

Proof. See Appendix, Section A.3.4. □

The complexity of the last step (6) for producing forecasts from the distributions is simply $O(w)$, assuming that we simply sum the probabilities of the first w points of a distribution and compare this sum to the given threshold θ_{fc}.

5 Measuring the Quality of Forecasts

Our goal is to predict the occurrence of CEs within a short future window or provide a "negative" forecast if our model predicts that no CE is likely to occur within this window. This can then be viewed as a classification task, where, at specific points in time, we emit such "positive" or "negative" forecasts which let us know whether we should expect a CE within the next w points. Note that forecasting could also be viewed as a regression task where the goal would be to forecast when a CE will happen and the output would then be a number (or an interval within which the CE is expected to happen). Due to space limitations, in this paper we only discuss forecasting as classification.

In order to properly measure the quality of the produced forecasts, the first issue that needs to be resolved is how to establish *checkpoints* in the stream, i.e., points where it makes sense to emit forecasts. Eagerly emitting forecasts after every new SDE is feasible in principle, 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 vast 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 Barcelona. We can also try to use this pattern for forecasting, with the goal of predicting when the vessel will arrive at the port of Barcelona. 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 these, necessarily low-quality, distant forecasts.

Our proposed solution is to use a pattern's automaton in order 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, with the following reasoning. For an automaton that is in a final state, it can be said that the distance to a CE is 0. More conveniently, we can say that the "process" which it 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 take 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 take the transition to a final state (and we thus have an absence of a CE), this "expected" distance is not an actual distance, but 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 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. 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. Another option for measuring the distance of a state to a possible future CE would be to use the waiting-time distribution of the state and set its expectation as the distance. However, this assumes that we have gone through the training phase first and learned the distributions. For this reason, we avoid using this way to estimate distances.

The task itself consists in 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 have a binary classification task.

As far as the metrics are concerned, we use the standard ones used in classification tasks, 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. This means that, for each distance, we plot precision-recall and ROC curves, using the points we obtain from the values of θ_{fc}. For each distance, we can then estimate the area under curve (AUC) for both the precision-recall and ROC curves. The higher the AUC value, the better the model is assumed to be.

6 Empirical Evaluation

We now present experimental results on two datasets, a synthetic one (Section 6.3) and and a real-world one (Section 6.4). We first briefly discuss the models that we test in Section 6.1. and present our software and hardware settings in Section 6.2.

6.1 Models Tested

In the experiments that we present, we test five different models for event forecasting in total. One of them is the variable-order Markov model that we have presented in this paper: the one that employs the optimization of Section 4.6 and bypasses the construction of a Markov chain, using directly the PST learned from a stream. The remaining four models that we have implemented have been inspired by models previously proposed in the literature.

The first, described in [6, 7], is similar in its general outline to our proposed method. It is also based on automata and Markov chains, the main difference being that it attempts to construct full-order Markov models of order m, thus being typically restricted to low values for m.

The second model is presented in [31], 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 test a model that is based on Hidden Markov Models (HMM), similar to the work presented in [33]. This 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 pattern. The observation variable of the HMM corresponds to the states of the pattern automaton, i.e., an observation sequence of length l for the HMM consists of the sequence of states visited by the automaton after consuming l SDEs. We can train a HMM for an automaton 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 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 interval, 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" 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. The upside with using HMMs is that we can at least estimate the waiting-time distribution, even if this is possible only in an online manner.

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 complex event forecasting. 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 mean "waiting time".

6.2 Hardware and Software Settings

All experiments were run on a 64-bit Debian 10 (buster) 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[1]. No attempt at parallelization was made. All experiments were run in a centralized setting.

6.3 Credit Card Fraud Management

The first dataset against which we test our method 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 more easily controlled and that produce results more readily interpretable. 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 small amount 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. Most of them are genuine transactions, with $\approx 20\%$ being fraudulent. We inject seven different types of fraudulent patterns in the dataset, with the pattern for the increasing trend being one of them (a decreasing trend is another such pattern). Each fraudulent sequence for the increasing trend consists of eight consecutive transactions with increasing amounts, where the amount is increased each time by 100 or more units. We additionally inject similar sequences of transactions with increasing amounts, which, however, do not lead to a fraud and completion of the pattern. We randomly interrupt the sequence before it reaches the eighth transaction. In these cases of genuine 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.

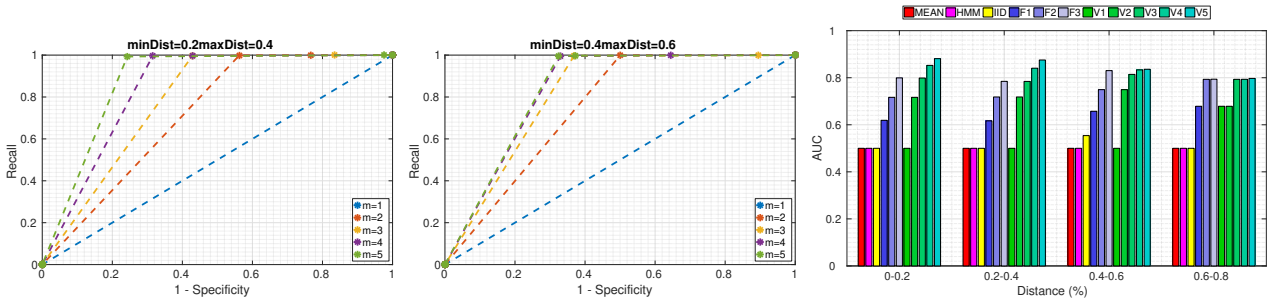
Formally, the symbolic regular expression that we use to capture the pattern of an increasing trend in the amount spent is the following:

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

amountDiff is an extra attribute 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 subexpressions, in order to capture eight consecutive events (the first terminal subexpression captures an increasing amount between the first two events in a fraudulent pattern). If we attempted to perform forecasting based solely on Expression 7, then the only minterms that would be created would be based only on the predicate $amountDiff > 0$: this predicate itself, along with its negation $\neg(amountDiff > 0)$. As expected, such an approach does not yield optimal results, since, even with high orders, the learned conditional probabilities would involve just these two minterms. In order to address this issue of expressions that, in their definitions, do not have 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). For Expression 7, we have decided to include the following extra predicate: $amountDiff > 100$. The expectation is that, by including this predicate, we will be able to differentiate more easily 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 9 shows the ROC curves that we obtain by running classification experiments with the variable-

[1] If the paper is accepted, the source code will be publicly released. A version with instructions on how to reproduce the presented results, released exclusively for the reviewers of VLDBJ, may be found here: `someurl`.

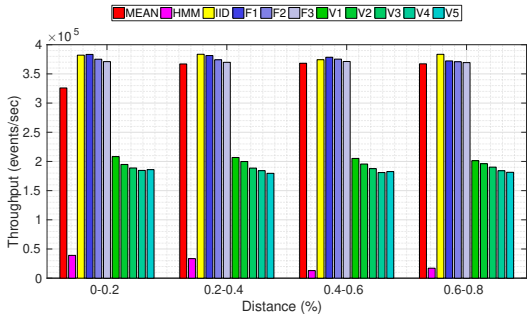


(a) ROC curves for the variable-order model using the prediction suffix tree for various values of the maximum order. $minDist = 0.2$ and $maxDist = 0.4$.

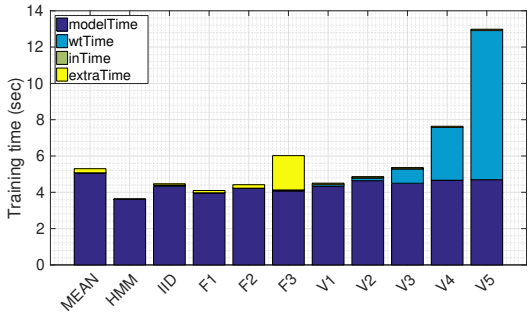
(b) ROC curves for the variable-order model using the prediction suffix tree for various values of the maximum order. $minDist = 0.4$ and $maxDist = 0.6$.

(c) AUC for ROC curves for all models.

Fig. 9: Results for CE forecasting from the domain of credit card fraud management. Fx stands for a Full-order Markov Model of order x. Vx stands for a Variable-order Markov Model of maximum order x. 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.



(a) Throughput.



(b) Training time.

Fig. 10: Throughput and training time results for classification CE forecasting from the domain of credit card fraud management. modelTime = time to construct the model. wtTime = time to estimate the waiting-time distributions for all states. inTime = time to estimate the forecast of all states from their waiting-time distributions. extraTime = time to determinize an automaton (+ disambiguation time for full-order models).

order model that directly uses a prediction suffix tree. We show results for two different "expected" distance ranges: for $minDist = 0.2$ and $maxDist = 0.4$ in Figure 9a and for $minDist = 0.4$ and $maxDist = 0.6$ in Figure 9b. The more area a ROC curve covers, the better the corresponding model is assumed to be. We see that increasing the maximum order does indeed lead to better results. Notice, however, that in Figure 9b, where the distance is greater, increasing the order from 4 to 5 yields only marginally better results. This implies that, the more the distance increases, the less important it is to increase the order. Figure 9c gathers results for ROC for all distances and models. The first observation is that the MEAN and HMM methods consistently underperform compared to the Markov models. With respect to the Markov models, as expected, the task becomes more challenging and both the ROC scores decrease, as the distance increases. We can also see that it is indeed important to be able to increase the order of the model. The advantage of increasing the order becomes less pronounced (or even non-existent for high orders) as the distance increases.

We also show throughput and training time results in Figure 10. Figure 10b, which shows training times, is 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, after the model has been constructed. *inTime* 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. These results demonstrate the trade-off between the better results we can obtain with high order models and the performance penalty that these models incur. The models based on prediction suffix trees have a throughput figure that is almost half that for the full-order models, while their training time is also significantly higher than the other models. This is due to the fact that these tree models, in order to emit a forecast, need to traverse a tree after every new event arrives at the system, as described in Section 4.6. The automata-based full-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. By far the worst, however, are the HMM models. The reason is that the waiting-time distributions and forecasts are always estimated online, as explained in Section 6.1 (this is why they have low training times). It would be possible to improve these figures by using caching techniques, so that we can reuse some of the previously estimated forecasts, but we reserve such optimizations for future work.

6.4 Maritime Monitoring

The second dataset that we used for 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 messages that relay information about their position, heading, speed, etc., as described in the running example of Section 1.1. These trajectories can be analyzed, using the techniques of CER, in order to detect interesting patterns in the behavior of vessels [34]. 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 [37]. Due to the fact that AIS messages are noisy, we used a derivative dataset that contains clean and compressed trajectories, consisting only of critical points [35]. Critical points are the important points of a trajectory that indicate a significant change in the behavior of a vessel but allow for an accurate reconstruction of the original trajectory [34]. 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 preprocessing step is that AIS messages typically arrive at unspecified time intervals. These intervals can exhibit a very wide variation, depending on a lot of factors (e.g., human operators may turn on/off the AIS equipment),

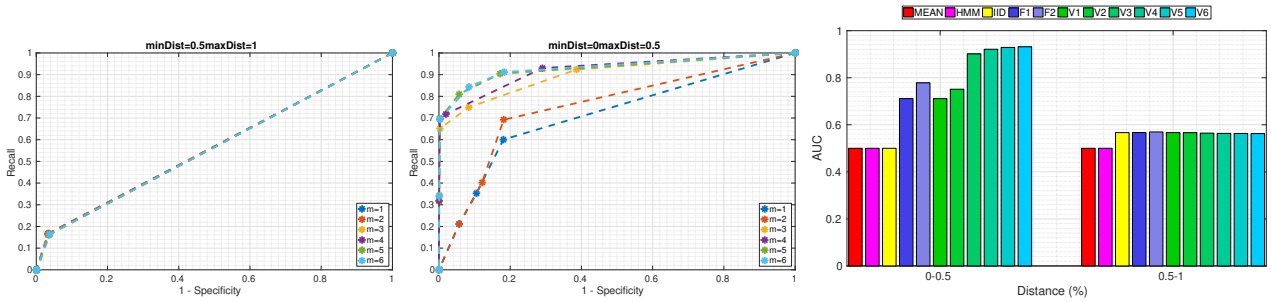
Fig. 11: Trajectories of the vessel with the most matches for Pattern 8 around the port of Brest. Green points denote ports. Red, shaded circles show the areas covered by each port. They are centered around each port and have a radius of 5 km.

without any clear pattern that could be encoded by our probabilistic model. Consequently, running our experiments directly on the raw or compressed dataset may yield sub-optimal results.

The pattern that we used 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. The symbolic regular expression for this pattern is the following:

$$R := (\neg InsidePort(Brest))^* \cdot (\neg InsidePort(Brest)) \cdot (\neg InsidePort(Brest)) \cdot (InsidePort(Brest)) \tag{8}$$

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. The predicate is generic and any port can be passed to it as an argument. In this case, we use the Brest port and this is the reason why Brest is the argument to *InsidePort*. In fact, we pass the longitude and latitude of a point as attributes, but we show here a simplified version for readability reasons. The pattern then defines the entrance to the port as a sequence of at least 3 consecutive events. The last event must always indicate that the vessel is inside the port. It could be argued that we could use just this last event as our pattern (i.e., a single event with the $InsidePort(Brest)$ predicate), but doing so would prevent us from differentiating between entrances and exits. 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

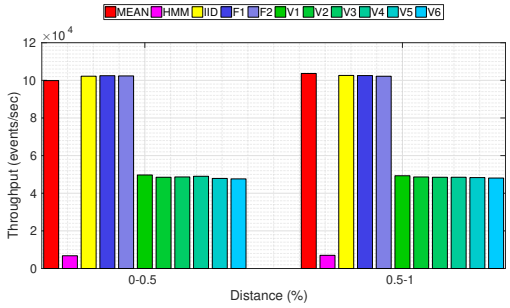


(a) ROC curves for the variable-order model using the prediction suffix tree for various values of the maximum order. $minDist = 0.0$ and $maxDist = 0.5$.

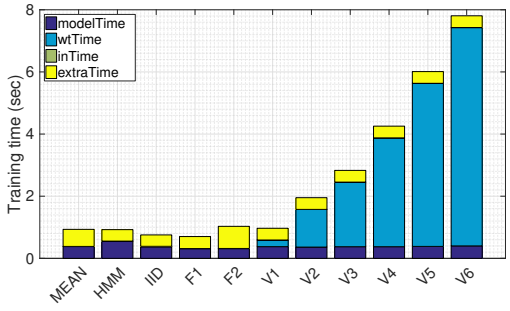
(b) ROC curves for the variable-order model using the prediction suffix tree for various values of the maximum order.$minDist = 0.5$ and $maxDist = 1.0$.

(c) AUC for ROC curves for all models.

Fig. 12: Results for CE forecasting from the domain of maritime monitoring.



(a) Throughput.



(b) Training time.

Fig. 13: Throughput and training time results for classification CE forecasting from the domain of maritime monitoring.

the port. We use the Boolean operator of negation ($\neg$), applied to the predicate $InsidePort(Brest)$, in order to determine whether a vessel is outside the port. We require 2 or more such events to have occurred (instead of just one) in order to avoid detecting any "noisy" entrances, by making sure that the vessel was consistently outside the port before finally entering it.

As was the case with the experiments on credit card data, we also try to incorporate some extra predicates during the construction of our probabilistic models for Pattern 8. For our initial experiments, we included 5 extra predicates 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 similarly cover the remaining "rings" with a 1 km width until 10 km. We will later investigate the sensitivity of our models to the presence or absence of various extra predicates.

For all experimental results that follow, we always employ 4-fold cross validation. We first show results by analyzing the trajectories of a single vessel. Results with multiple, selected vessels will be shown later. There are two issues that we try to address by separating our experiments into single-vessel and multiple-vessel experiments and by excluding some vessels. First, we need to make sure that we have enough data for training. For this reason, we only retain vessels for which we can detect a significant number of matches for Pattern 8. Second, we have experimentally observed that building a global model from multiple vessels tends to produce sub-optimal results (this was not the case with credit card data because the dataset was synthetic and all cards exhibited the same behavior). This is why we first focus on a single vessel. 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 events corresponding to it is ≈ 30.000. Figure 11 shows the isolated trajectories for this vessel.

We now show results for classification CE forecasting in Figures 12a, 12b, 12c and 13. We can observe here 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 true positives and true negatives. Their forecasts are either all positive, where we have *Recall* = 100% and *Specificity* = 0%, or all negative, where we have *Recall* = 0% and *Specificity* = 100% (see Figure 12b). Achieving higher scores with higher-order models comes at a cost though. The training time for variable-order models tends to increase as we increase the order. This increase may be tolerated, given that the total training time is always less than 8 seconds and that training is an offline process. The effect on throughput is also significant for the tree-based variable-order models. The reason behind this reduced performance is the fact that, upon each new event, these models need to traverse the tree structure that they maintain.

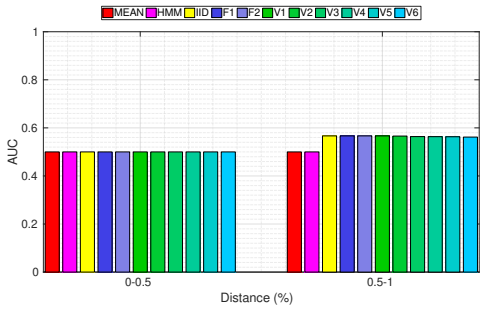
Finally, we test our proposed method in two more scenarios. First, we want to intvestigate how our approach behaves with different extra features (or none at all). Figures 14a, 14b and 14c show the relevant results for classification CE forecasting. Figure 14a shows results when no extra features are included in the construction of the probabilistic model, i.e., when only the predicate *InsidePort*(*Brest*) and its negation, present in the pattern itself, are taken into account. Without any extra predicates, the model cannot produce any meaningful forecasts. Figure 14b shows results when the extra predicates referring to the distance of a vessel from the port are modified so that each "ring" around the port has a width of 3 km. With these extra features, increasing the order does indeed make an important difference, but the best score achieved is still lower than the best score achieved with "rings" of 1 km (Figure 12c). "Rings" of 1 km are thus more appropriate as predictive features. Figure 14c shows results from yet another variation. We keep the features of 1 km "rings" but we also add one more feature which checks whether the heading of a vessel points towards the port (more precisely, we use the vessel's speed and heading to project its location 1 hour ahead in the future and then check whether this projected segment and the circle around the port intersect). The intuition for adding this feature is that the knowledge of whether a vessel is heading towards the port predictive value. As observed, the best score we can achieve with this set of features is not in fact higher than the best score without the feature of heading towards the port (Figure 12c). However, it is interesting to notice that this set of features allows us to achieve high scores even with low orders, reached even with full-order models. The heading feature is thus indeed important. On the other hand, the high-order models without this feature seem to be able to "infer" whether the vessel is heading towards the port, even without this feature being explicitly included.

In the second scenario that we tested, we used more than one vessel. 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. The resulting dataset has $\approx$ 222.000 events. Out of the 9 retained vessels, we construct a global probabilistic model and produce forecasts according to this model. Note that the another option would be to build a single model for each vessel (as we previously did with the vessel having the most matches), but in this scenario we wanted to test the robustness of our aprroach when a global model is built from multiple entities. Figure 14d shows the relevant classification results. The scores are very similar to the scores of the single-vessel model (Figure 12c). The main difference is that, as expected, the scores for the multi-vessel models are slightly lower, due to the different behavioral patterns that different vessels follow.

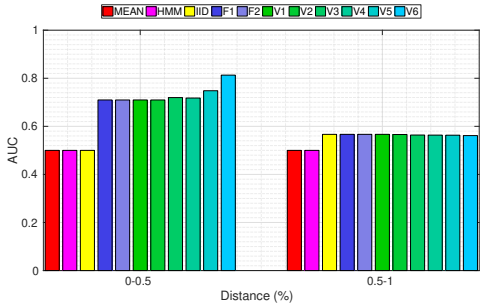
7 Summary & Future Work

We have presented a framework for Complex Event Forecasting, based on a variable-order Markov model, which allows us to delve deeper into the past and capture long-term dependencies, not feasible with full-order models. Our empirical evaluation on two different datasets has shown the advantage of being able to use high-order models. Another important feature of our proposed framework is that it requires minimal intervention by the user. A given CE pattern is declaratively defined and it is subsequently automatically translated to an automaton and then to a Markov model, without requiring extensive domain knowledge that should guide the modeling process.

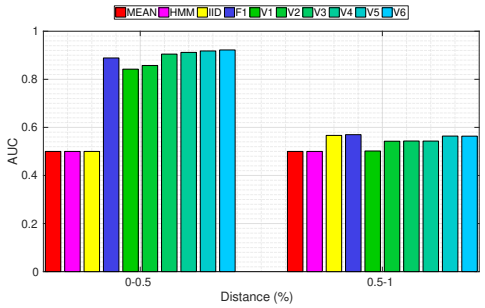
The weak point of our framework, with respect to user involvement, is its inability to automatically select the features that are the most informative. Currently, the user has to manually provide these features, but we intend to explore ways to automate this process in the future. The user also has to manually set the maximum order allowed by the probabilistic model. Automatically estimating the optimal order could thus be another possible direction for future work. We have also started investigating ways to handle concept drift by continuously training and updating the probabilistic model of a pattern.



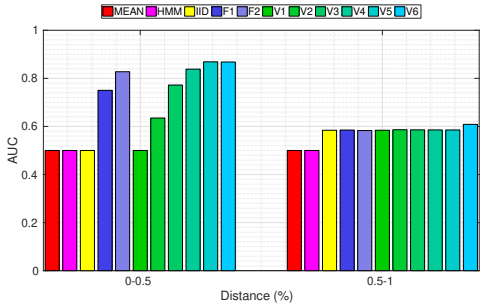
(a) AUC for ROC curves. Extra features included: none. Single vessel.



(b) AUC for ROC curves. Extra features included: concentric rings around the port every 3 km. Single vessel.



(c) AUC for ROC curves. Extra features included: concentric rings around the port every 1 km and heading. Single vessel.



(d) 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.

Fig. 14: Results for classification CE forecasting from the domain of maritime monitoring for various sets of extra features and for multiple vessels.

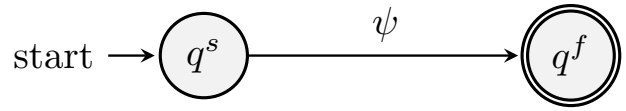
It is often the case that, in addition to detecting when a CE occurs, we also need to know which of the input events take part into a detected CE. This functionality would require the use of symbolic transducers in order to mark the input events according to whether they belong to a match or not [23]. We currently do not pay particular attention to this requirement, since our main goal is to forecast the occurrence of CEs, without needing to report the input events that lead to a CE occurrence. We aim to address this requirement in the future.

Finally, our framework could conceivably be used for a task that is not directly related to Complex Event Forecasting. Since forecasting/prediction and compression are the two sides of the same coin, our framework could be used for pattern-driven lossless compression in order to minimize the communication cost, which could be a severe bottleneck for geo-distributed CER. The probabilistic model that we construct with our approach could be pushed down to the event sources in order to compress each individual stream and then these compressed stream could be transmitted to a centralized CER engine to perform recognition.

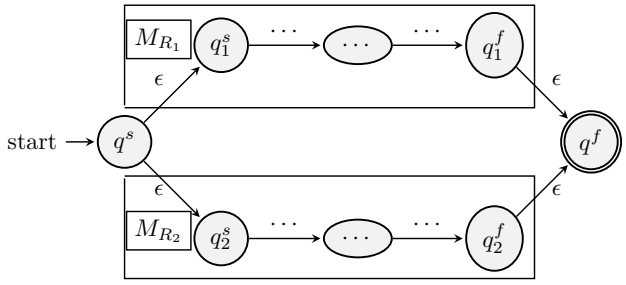
References

1. Esper. `http://www.espertech.com/esper`. [Online; accessed 16-January-2020]
2. Smile - statistical machine intelligence and learning engine. `http://haifengl.github.io/`. [Online; accessed 16-January-2020]
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. Bühlmann, P., Wyner, A.J., et al.: Variable length markov chains. The Annals of Statistics **27**(2), 480–513 (1999)

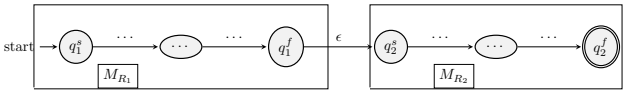
12. Cho, C., Wu, Y., Yen, S., Zheng, Y., Chen, A.L.P.: Online rule matching for event prediction. VLDB J. **20**(3), 303–334 (2011)
13. Christ, M., Krumeich, J., Kempa-Liehr, A.W.: Integrating predictive analytics into complex event processing by using conditional density estimations. In: EDOC Workshops, 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.: Extended symbolic finite automata and transducers. Formal Methods in System Design **47**(1), 93–119 (2015)
17. 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)
18. Engel, Y., Etzion, O.: Towards proactive event-driven computing. In: DEBS, pp. 125–136. ACM (2011)
19. Etzion, O., Niblett, P.: Event Processing in Action. Manning Publications Company (2010)
20. Fu, J.C., Lou, W.W.: Distribution theory of runs and patterns and its applications: a finite Markov chain imbedding approach. World Scientific (2003)
21. 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)
22. 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)
23. 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)
24. Hedtstück, U.: Complex event processing: Verarbeitung von Ereignismustern in Datenströmen. Springer Vieweg, Berlin (2017)
25. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation, 3rd Edition. Pearson international edition. Addison-Wesley (2007)
26. 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)
27. 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)
28. Luckham, D.C.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley (2001)
29. 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)
30. Montgomery, D.C., Jennings, C.L., Kulahci, M.: Introduction to time series analysis and forecasting. John Wiley & Sons (2015)
31. Muthusamy, V., Liu, H., Jacobsen, H.: Predictive publish/subscribe matching. In: DEBS, pp. 14–25. ACM (2010)
32. 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)
33. 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)
34. 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)
35. 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`
36. Rabiner, L.R.: A tutorial on hidden markov models and selected applications in speech recognition. Proceedings of the IEEE **77**(2), 257–286 (1989)
37. 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`
38. Ron, D., Singer, Y., Tishby, N.: The power of amnesia. In: NIPS, pp. 176–183. Morgan Kaufmann (1993)
39. 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)
40. Sakarovitch, J.: Elements of Automata Theory. Cambridge University Press (2009)
41. Veanes, M., de Halleux, P., Tillmann, N.: Rex: Symbolic regular expression explorer. In: ICST, pp. 498–507. IEEE Computer Society (2010)
42. Vilalta, R., Ma, S.: Predicting rare events in temporal domains. In: ICDM, pp. 474–481. IEEE Computer Society (2002)
43. 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)
44. 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)
45. Zhou, C., Cule, B., Goethals, B.: A pattern based predictor for event streams. Expert Syst. Appl. **42**(23), 9294–9306 (2015)



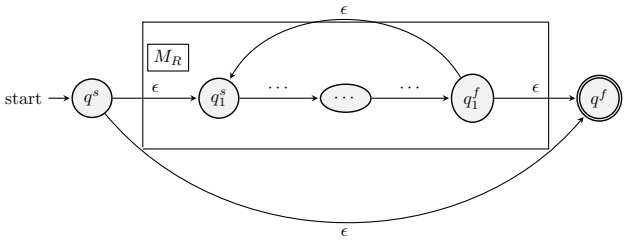
(a) Base case of a single predicate. $R = \psi$.



(b) OR. $R = R_1 + R_2$.



(c) Concatenation. $R = R_1 \cdot R_2$.



(d) Iteration. $R^{'} = R^*$.

Fig. 15: The four cases for constructing a SFA from a SRE

A Appendix

A.1 Complete proof of Proposition 1

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

Proof. Except for the first case, for the other three cases the induction hypothesis is that the theorem holds for the sub-expressions of the initial expression.

Case where $R = \psi, \psi \in \Psi$.
We construct a SFA as in Figure 15a. If $w \in \mathcal{L}(R)$, then w is a single character and $w \in [\![\psi]\!]$, i.e., ψ evaluates to TRUE for w. Thus, upon seeing w, the SFA of Figure 15a moves from q^s to q^f and since q^f is a final state, then w is accepted by this SFA. Conversely, if a string w is accepted by this SFA then it must again be a single character and ψ must evaluate to TRUE since the SFA moved to its final state through the transition equipped with ψ. Thus, $w \in [\![\psi]\!]$ and $w \in \mathcal{L}(R)$.

Case where $R = R_1 + R_2$.
We construct a SFA as in Figure 15b. If $w \in \mathcal{L}(R)$, then either $w \in \mathcal{L}(R_1)$ or $w \in \mathcal{L}(R_2)$ (or both). Without loss of generality, assume $w \in \mathcal{L}(R_1)$. From the induction hypothesis, it also holds that $w \in \mathcal{L}(M_{R_1})$. Thus, from Figure 15b, upon reading w, M_{R_1} will have reached q_1^f. Therefore, M_R will have reached q^f throught the ϵ-transition connecting q_1^f to q^f and thus w is accepted by M_R. Conversely, if $w \in \mathcal{L}(M_R)$, then the SFA M_R of Figure 15b must have reached q^f and therefore also q_1^f or q_2^f (or both). Assume it has reached q_1^f. Then $w \in \mathcal{L}(M_{R_1})$ and, from the induction hypothesis $w \in \mathcal{L}(R_1)$. Similarly, if its has reached q_2^f, then $w \in \mathcal{L}(R_2)$. Therefore, $w \in \mathcal{L}(R_1) \cup \mathcal{L}(R_2) = \mathcal{L}(R)$.

Case where $R = R_1 \cdot R_2$.
We construct a SFA as in Figure 15c. If $w \in \mathcal{L}(R)$, then $w \in \mathcal{L}(R_1) \cdot \mathcal{L}(R_2)$ or $w = w_1 \cdot w_2$ such that $w_1 \in \mathcal{L}(R_1)$ and $w_2 \in \mathcal{L}(R_2)$. Therefore, from the induction hypothesis, upon reading w_1, M_R will have reached q_1^f and q_2^s. Upon reading the rest of w (w_2), again from the induction hypothesis, M_R will have reached q_2^f. As a result, $w \in \mathcal{L}(M_R)$. Conversely, if $w \in \mathcal{L}(M_R)$, M_R will have reached q_2^f upon reading w and therefore will have also passed through q_1^f upon reading a prefix w_1 of w. Thus, $w = w_1 \cdot w_2$ with $w_1 \in \mathcal{L}(M_{R_1})$ and $w_2 \in \mathcal{L}(M_{R_2})$. From the induction hypothesis, it also holds that $w_1 \in \mathcal{L}(R_1)$ and $w_2 \in \mathcal{L}(R_2)$ and therefore that $w \in \mathcal{L}(R)$.

Case where $R^{'} = R^*$.
We construct a SFA as in Figure 15d. If $w \in \mathcal{L}(R^{'})$, then $w \in (\mathcal{L}(R))^*$ or, equivalently, $w = w_1 \cdot w_2 \cdot \cdots \cdot w_k$ such that $w_i \in \mathcal{L}(R)$ for all w_i. From the induction hypothesis and Figure 15d, upon reading w_1, $M_{R^{'}}$ will have reached q_1^f and q_1^s. Therefore, the same will be true after reading w_2 and all other w_i, including w_k. Thus, $w \in \mathcal{L}(M_{R^{'}})$. Note that if $w = \epsilon$, the ϵ-transition from q^s to q^f ensures that $w \in \mathcal{L}(M_{R^{'}})$. Conversely, assume $w \in \mathcal{L}(M_{R^{'}})$. If $w = \epsilon$, then by the definition of the * operator, $w \in (\mathcal{L}(\mathcal{R}))^*$. In every other case, $M_{R^{'}}$ must have reached q_1^f and must have passed through q_1^s. Therefore, w may be written as $w = w_1 \cdot w_2$ where $w_2 \in \mathcal{M}_{\mathcal{R}}$ and, for w_1, upon reading it, $M_{R^{'}}$ must have reached q_1^s. There are two cases then: either $w_1 = \epsilon$ and q_1^s was reached from q^s or $w_1 \neq \epsilon$ and q_1^s was reached from q_1^f. In the former case, $w = \epsilon \cdot w_2 = w_2$ and thus $w \in (\mathcal{L}(R))^*$. In the latter case, we can apply a similar reasoning recursively to w_1 in order to split it to sub-strings w_i such that $w_i \in \mathcal{L}(R)$. Therefore, $w \in (\mathcal{L}(R))^*$ and $w \in \mathcal{L}(R^{'})$.

□

A.2 Proof of Theorem 2

Theorem. *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 F, can be obtained from*

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

where ξ_F is the vector consisting of the elements of ξ_{init} corresponding to the states of F.

Proof. **Case where $n = 2$.**
In this case, we are in a state $i \in F$ and we take a transition that leads us back to F again. Therefore, $P(Y_2 \in F, Y_1 = i \in F \mid \boldsymbol{\xi_{init}}) = \boldsymbol{\xi}(i) \sum_{j \in F} \pi_{ij}$, i.e., we first take the probability of starting in i and multiply it by the sum of all transitions from i that lead us back to F. This result folds for a certain state $i \in F$. If we start in any state of F, $P(Y_2 \in F, Y_1 \in$

$F \mid \boldsymbol{\xi_{init}}) = \sum_{i \in F} \boldsymbol{\xi}(i) \sum_{j \in F} \pi_{ij}$. In matrix notation, this is equivalent to $P(Y_2 \in F, Y_1 \in F \mid \boldsymbol{\xi_{init}}) = \boldsymbol{\xi_F}^T \boldsymbol{F1}$.

Case where $n > 2$.

In this case, we must necessarily first take a transition from $i \in F$ to $j \in N$, then, for multiple transitions we remain in N and we finally take a last transition from N to F. We can write

$$P(Y_n \in F, Y_{n-1} \notin F, ..., Y_1 \in F \mid \boldsymbol{\xi_{init}}) = P(Y_n \in F, Y_{n-1} \notin F, ..., Y_2 \notin F \mid \boldsymbol{\xi'_N}) = P(Y_{n-1} \in F, Y_{n-2} \notin F, ..., Y_1 \notin F \mid \boldsymbol{\xi'_N}) \quad (9)$$

where $\boldsymbol{\xi'_N}$ is the state distribution (on states of N) after having taken the first transition from F to N. This is given by $\boldsymbol{\xi'_N} = \boldsymbol{\xi_F}^T \boldsymbol{F_N}$. By using this as an initial state distribution in Eq. 2 and running the index n from 1 to $n-1$, as in Eq. 9, we get

$$P(Y_n \in F, Y_{n-1} \notin F, ..., Y_1 \in F \mid \boldsymbol{\xi_{init}}) = \boldsymbol{\xi_F}^T \boldsymbol{F_N} \boldsymbol{N}^{n-2} (\boldsymbol{I} - \boldsymbol{N}) \boldsymbol{1}$$

□

A.3 Proofs of Complexity Results

A.3.1 Proof of Proposition 2

Proposition (Step 1 in Figure 8)**.** *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 $O(m(k-m))$.*

Proof. There are three operations that affect the cost: incrementing the counter of a node by 1, with constant cost i; inserting a new node, with constant cost n; visiting an existing node with constant cost v; We assume that $n > v$. For every $S_{l-m+1..l}$, $m \leq l \leq k$ of length m, there will be m increment operations and m nodes will be "touched", i.e., either visited if already existing or created. Therefore, the total number of increment operations is $(k-m+1)m = km - m^2 + m = m(k-m)+m$. The same result applies for the number of node "touches". It is always true that $m < k$ and typically $m \ll k$. Therefore, the cost of increments is $O(m(k-m))$ and the cost of visits/creations is also $O(m(k-m))$. Thus, the total cost is $O(m(k-m))+O(m(k-m)) = O(m(k-m))$. In fact, the worst case is when all $S_{l-m+1..l}$ are different and have no common suffixes. In this case, there are no visits to existing nodes, but only insertions, which are more expensive than visits. Their cost would again be $O(nm(k-m)) = O(m(k-m))$, ignoring the constant n. □

A.3.2 Proof of Proposition 3

Proposition (Step 3 in Figure 8)**.** *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 $O(t^{m+1} \cdot m)$.*

Proof. We assume that the cost of creating new states and transitions for M_S is constant. In the worst case, all possible suffixes of length m have to be added to T as leaves. T will thus have t^m leaves. The main idea of the algorithm for converting a *PST* T to a *PSA* M_S is to use the leaves of T as states of M_S and for every symbol (minterm) σ find the next state/leaf and set the transition probability to be equal to the probability of σ from the source leaf. If we assume that the cost of accessing a leaf is constant (e.g., by keeping separate pointers to the leaves), the cost for constructing M_S is dominated by the cost of constructing the k^m states of M_S and the t transitions from each such state. For each transition, finding the next state requires traversing a path of length m in T. The total cost is thus $O(t^m \cdot t \cdot m) = O(t^{m+1} \cdot m)$. □

A.3.3 Proof of Proposition 4

Proposition (Step 4 in Figure 8)**.** *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 $O(t \cdot |M_R.Q \times M_S.Q|)$.*

Proof. We assume that the cost of constructing new states and transitions for M is constant. We also assume that the cost of finding a given state in both M_R and M_S is constant, e.g., by using a linked data structure for representing the automaton with a hash table on its states (or an array), and the cost of finding the next state from a given state is also constant. In the worst case, even with an incremental algorithm, we would need to create the full Cartesian product $M_R.Q \times M_S.Q$ to get the states of M. For each of these states, we would need to find the states of M_R and M_S from which it will be composed and to create t outgoing transitions. Therefore, the complexity of creating M would be $O(t \cdot |M_R.Q \times M_S.Q|)$. □

A.3.4 Proof of Proposition 5

Proposition (Step 3′ in Figure 8)**.** *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 $O((m+3)\frac{t-t^{h+1}}{1-t})$.*

Proof. After every new event arrival, we first have to construct the tree of future states, as shown in Figure 6b. In the worst case, no paths can be pruned and the tree has to be expanded until level h. The total number of nodes that have to be created is thus a geometric progress: $t + t^2 + \cdots + t^h = \sum_{i=1}^{h} t^i = \frac{t-t^{h+1}}{1-t}$. Assuming that it takes constant time to create a new node, this formula gives the cost of creating the nodes of the trees. Another cost that is involved concerns the time required to find the proper leaf of the *PST* T before the creation of each new node. In the worst case, all leaves will be at level m. The cost of each search will thus be m. The total search cost for all nodes will be $mt + mt^2 + \cdots + mt^h = \sum_{i=1}^{h} mt^i = m\frac{t-t^{h+1}}{1-t}$. The total cost (node creation and search) for constructing the tree is $\frac{t-t^{h+1}}{1-t} + m\frac{t-t^{h+1}}{1-t} = (m+1)\frac{t-t^{h+1}}{1-t}$. With the tree of future states at hand, we can now estimate the waiting-time distribution. In the worst case, the tree will be fully expanded and we will have to access all its paths until level h. We will first have to visit the t nodes of level 1, then the t^2 nodes of level 2, etc. The access cost will thus be $t + t^2 + \cdots + t^h = \sum_{i=1}^{h} t^i = \frac{t-t^{h+1}}{1-t}$. We also need to take into account the cost of estimating the probability of each node. For each node, one multiplication is required, assuming that we store partial products and do not have to traverse the whole path to a node

to estimate its probability. As a result, the number of multiplications will also be $\frac{t-t^{h+1}}{1-t}$. The total cost (path traversal and multiplications) will thus be $2\frac{t-t^{h+1}}{1-t}$, where we ignore the cost of summing the probabilities of final states, assuming it is constant. By adding the cost of constructing the tree ($(m+1)\frac{t-t^{h+1}}{1-t}$) and the cost of estimating the distribution ($2\frac{t-t^{h+1}}{1-t}$), we get a complexity of $O((m+3)\frac{t-t^{h+1}}{1-t})$. □