Interactive Learning and Complex Event Forecasting Technology version 1
Work Package 6 Tasks 6.2, 6.3, Deliverable 6.2

Elias Alevizos, Alexander Artikis, Alexia Atsidakou, Ioannis Fikioris, Nikolaos Katzouris, Vissarion Konidaris, Periklis Mantenonglou, Evangelos Michelioudakis, Emmanouil Ntoulias, Georgios Paliouras, Vasilis Samoladas, Alexandros Troupiotis

Distribution list:

Group:	Others:
WP Leader: NCSR	INFORE Management Team
Task Leader: NCSR	INFORE Project Officer

Document history:

Revision	Date	Section	Page(s)	Modification
0.1	06/04/2020	All	All	First draft
0.3	10/04/2020	All	All	OMLDM by ATHENA
0.5	15/04/2020	All	All	More experimental results
0.7	19/04/2020	All	All	Cleaning up
0.8	20/04/2020	All	All	Ready for internal review
1.0	29/04/2020	All	All	Final version after integration of review comments

Approvals:

First Author:	Elias Alevizos (NCSR)	Date:	06/04/2020
Internal Reviewer:	Antonios Deligiannakis (Athena)	Date:	25/04/2020
Coordinator:	Antonios Deligiannakis (Athena)	Date:	29/04/2020

Contents

1 Introduction

This document presents the progress of the INFORE project with respect to Interactive, Online Learning and Data Mining and Distributed Complex Event Forecasting.

1.1 Purpose and Scope

The reader is expected to be familiar with Complex Event Processing, Artificial Intelligence and Distributed processing techniques, as well as the general intent and concept of the INFORE project. The target readership is:

- INFORE researchers
- INFORE audit

INFORE focuses on scalable event recognition, forecasting and machine learning of event definitions for extreme scale analytics. This document presents the current advancements and discusses the scientific and technological issues that are being investigated in Work-Package 6, with respect to Interactive, Online Learning and Data Mining and Distributed Complex Event Forecasting.

1.2 Relation to other Deliverables

This document is related to the following project deliverables:

- D1.1 : Requirements and Scenario Definitions (Life Sciences Use Case);
- D2.1 : Requirements and Scenario Definitions (Financial Use Case);
- D3.1 : Requirements and Scenario Definitions (Maritime Use Case);
- D3.1 : Architecture Definition.

1.3 Structure of the Deliverable

This document has the following structure:

- Section 2 presents an extension of an interval-based complex event recognition system, called PIEC, which operates on top of a probabilistic Event Calculus implementation. The improvements we propose concern handling data streams, contrary to the original batch implementation of PIEC. Towards this goal, we introduce oPIEC, an on-line event recognition system designed to tackle the uncertainty in data streams of probabilistic instantaneous event indications.

- Section 3 presents an Answer Set Programming (ASP) based approach to learning complex event patterns in the form of weighted temporal logical rules. Our proposed approach is capable of noise/uncertainty-resilient, probabilistic complex event recognition over an event stream, while using the labeled portions of the stream to update the underlying event pattern set's structure and weights. It builds on existing Statistical Relational Learning techniques, which it significantly improves via allowing to take advantage of the grounding, solving, optimization and uncertainty modeling abilities of modern answer set solvers.

- Section 4 presents an approach to online semi-supervised learning of rules for complex event recognition. Our method is built on top of SPLICE, an existing approach based on graph-based semi-supervised learning. We propose an hybrid distance measure that combines the structural measure of SPLICE with a mass-based dissimilarity. The structural measure is further enhanced by feature selection optimized for kNN classification, by adapting a state-of-the-art approach to metric learning. Finally a graph connection strategy is employed that favors temporal connections and provides guarantees about the labeling of unlabeled examples. Our empirical analysis suggests that our improved method outperforms its predecessor in terms of inferring the missing labels, at the price of a tolerable increase in processing time.

- Section 5 presents the Online Machine Learning and Data Mining (OMLDM) component of the INFORE platform. This component implements the state-of-the-art in distributed, online ML adopting a parameter server paradigm for incremental training of models. At the very same time, previously extracted models are deployed for analysis and inference purposes. This is in contrast to other frameworks such as SAMOA which lack a parameter server implementation and fills the gap of existing APIs such as MLlib in Spark or FlinkML which are mainly focused on batch data processing. The OMLDM Component currently supports (i) classification algorithms Passive Aggressive Classifier, Online Support Vector Machines and Vertical Hoeffding Trees, (ii) clustering algorithms BIRCH, Online k-means and StreamKM++ and (iii) regression techniques Passive Aggressive Regressor, Online Ridge and Polynomial Regression, Autoregressive Integrated Moving Average (ARIMA).

- Section 6 presents two methods for exploring the parameter space of cancer simulations. The first one is based on genetic algorithms, while the second employs random forests as a surrogate model for modeling the interesting areas of the space. In addition, a time-series discretization technique, called SAX, is used for transforming the generated simulations into symbolic example sequences that will eventually be used in order to learn patterns for Complex Event Forecasting (CEF).

- Section 7 presents a formal framework that attempts to address the issue of Complex Event Forecasting (CEF). Our framework is based on symbolic automata and a variable-order Markov model and has the ability to capture long-term dependencies in a stream. We also present experimental results with a prototype implementation, showing how our approach achieves better accuracy scores compared to previous, state-of-the-art CEF solutions.

- Section 8 takes the prototype implementation of the previous section and presents a new version enhanced in two major ways. First, it presents online training, as the models so far were trained in an offline manner. Second, it presents the distribution of both training (parameter estimation) and forecasting with the help of Flink. This distributed implementation is also backed with Kafka for input/output and request handling. Finally, an evaluation step is made to showcase the efficiency of the distribution.

- Section 9 summarizes our progress towards achieving INFORE objectives.

2 Probabilistic Complex Event Recognition

2.1 Introduction

Complex Event Recognition systems employ pattern-based algorithms to infer composite events from an input stream of time-stamped ‘simple, derived events’. This input is derived from the processing of indications and measurements of various sensors, depending on the domain under examination. The automatic recognition of complex activities has been deployed in many real-world applications ranging from human activity recognition [134], city transport management and maritime surveillance [119] to recognition of attacks in computer network nodes [47] and credit card fraud detection. A salient issue in most such applications is the uncertainty in the input data which often causes erroneous event detection. Acknowledging and resolving that inherent uncertainty leads to reliable complex event recognition systems. For that purpose, simple events have probabilities attached to them, serving as confidence estimates. Various approaches have been proposed for handling uncertainty in complex event recognition – see [11] for a recent survey.

Prob-EC [134] is a system based on a probabilistic logic programming implementation of the Event Calculus [86, 84], designed to handle data uncertainty and compute the probability of a complex event at each time-point. The Probabilistic Interval-Based Event Calculus (PIEC) is an extension of Prob-EC that computes, in linear-time, all maximal intervals during which a complex event is said to take place, with a probability above a given threshold [16]. By supporting interval-based recognition, PIEC has proven robust to noisy instantaneous probability fluctuations, and performs better in the common case of non-abrupt probability change.

We present an extension of PIEC, called oPIEC^b, which is capable of online recognition, as opposed to the batch processing of PIEC. This way, oPIEC^b may handle data streams. More precisely, our contributions are the following. First, we propose a technique for identifying the minimal set of data points that need to be cached in memory, in order to guarantee correct complex event recognition in a streaming setting. Furthermore, we present a way to further reduce the cached data points, supporting highly efficient recognition, while at the same time minimising the effects on correctness.

2.2 Related Work

Systems for complex event recognition accept as input a stream of time-stamped sensor events and identify composite events of interest—combinations of events that satisfy some pattern. See [47, 25, 14, 13, 119] for a few applications. The event streams that provide the input data to a complex event recognition system exhibit various types of uncertainty [11, 56]. Consequently, the input events are often accompanied by a probability value.

A recent survey [11] identified the following classes of methods for handling uncertainty in complex event recognition: automata-based methods, probabilistic graphical models, probabilistic/stochastic Petri Nets, and approaches based on stochastic (context-free) grammars. In automata-based methods (e.g. [4, 151]), the representation of time is implicit, and hierarchical knowledge, i.e., defining a complex event in terms of some other complex event, is not typically supported. The approaches that use Petri Nets and stochastic grammars do not support relations between attributes of simple events and complex events [11].

Regarding probabilistic graphical models, Markov Logic Networks (MLNs) [125] have been used for complex event recognition. As an example, Morariu and Davis [106] employed Allen’s Interval Algebra [12] to determine the most consistent sequence of complex events, based on the observations of low-level classifiers. A bottom-up process discards the unlikely event hypotheses, thus avoiding the combinatorial explosion of all possible intervals. This elimination process, however, can only be applied to domain-dependent axioms, as it is guided by the observations. Sadilek and Kautz [128] employed hybrid-MLNs [147] in order to detect human interactions using location data from GPS devices. 'Hybrid formulas', i.e., formulas with weights associated with a real-valued function, such as the distance between two persons, de-noise the location data. In contrast to the above, a *domain-independent* probabilistic activity recognition framework via MLNs was presented in [136]. This framework is based on the Event Calculus [86, 107, 108] and handles complex event definition uncertainty by modelling imperfect rules expressing complex events.

There are also logic-based approaches to complex event recognition that do not (directly) employ graphical models. The Probabilistic Event Logic [25, 131] has been used to define a log-linear model from a set of weighted formulas expressing complex events [133]. Recognition is performed using 'spanning intervals' that allow for a compact representation of event occurrences satisfying a formula. In [7], complex events are defined in a first-order logic, the input simple events may be deterministic or probabilistic, while their dependencies are modelled by triangular norms [52]. Shet et al. [132] handled uncertainty by expressing the Bilattice framework in logic programming [63]. Each complex and simple event is associated with two uncertainty values, indicating, respectively, a degree of information and confidence. The more confident information is provided, the stronger the belief about the corresponding complex event becomes.

Skarlatidis et al. [134] presented an activity recognition system based on Prob-EC, a probabilistic logic programming implementation of the Event Calculus [86, 84]. Similar to [136], Prob-EC computes the probability of a complex event at each time-point. Unlike [136], Prob-EC is designed to handle data uncertainty. The use of the Event Calculus, as in, e.g., [40], allows the development of domain-independent, expressive event recognition frameworks, supporting the succinct, intuitive specification of complex events, by taking advantage of the built-in representation of inertia. Consequently, the interaction between event definition developer and domain expert is facilitated, and code maintenance is supported.

Recently, Artikis et al. [16] proposed the Probabilistic Interval-based Event Calculus (PIEC), a method for computing in linear-time all maximal intervals during which a complex event is said to take place, with a probability above a given threshold. PIEC was proposed as an extension of Prob-EC, but may operate on top of any Event Calculus dialect for point-based probability calculation (such as [136, 40]). By supporting interval-based recognition, PIEC is robust to noisy instantaneous probability fluctuations, and outperforms point-based recognition in the common case of non-abrupt probability change.

Note that various Event Calculus dialects allow for event duration by means of durative events or by explicitly representing 'fluent' intervals. These dialects, however, cannot handle uncertainty.

PIEC was designed to operate in a batch mode, requiring all available data for correct interval computation. We present an extension of PIEC for online recognition where data arrive in a streaming fashion.

An area related to complex event recognition is that of 'Run-time Verification', i.e., the online monitoring of a system's correctness with regard to a set of desired behaviours (specifications). For instance, in [73], the run-time monitoring of IoT systems is performed by means of an event-oriented temporal logic. The methods of run-time verification need to handle uncertainty, originating, e.g., from network issues or event sampling [74, 22]. As an example, [22] handles lossy traces via monitors robust to a transient loss of events (short intervals of missing indications). Similarly, PIEC features robustness to transient noise in the input complex event probabilities, i.e., brief probability fluctuations.

2.3 Background

The Event Calculus is a formalism for representing and reasoning about events and their effects [86]. Since its original proposal, many dialects have been put forward, including formulations in (variants of) first-order logic and as logic programs. As an example, in Prob-EC [134] a simple version of the Event Calculus was presented, with a linear time model including integer time-points. The ontology of most such dialects comprises time-points, events and 'fluents', i.e. properties that are allowed to have different values at different points in time. Event occurrences may change the value of fluents. Hence, the Event Calculus represents the effects of events via fluents. Given a fluent F, the term $F=V$ denotes that F has value V. A key feature of the Event Calculus is the built-in representation of the common-sense law of inertia, according to which $F=V$ holds at a particular time-point, if $F=V$ has been 'initiated' by an event at some earlier time-point, and not 'terminated' by another event in the meantime. A set of complex event definitions may be expressed as an Event Calculus *event description*, i.e., axioms expressing the simple events occurrences as Event Calculus events, the values of fluents expressing complex vents, and the effects of simple events i.e., the way simple events define complex events.

The Event Calculus has been expressed in frameworks handling uncertainty, such as ProbLog [84], in the case of Prob-EC [134], and Markov Logic Networks in [136], in order to perform probabilistic, time-point-based complex event recognition recognition. The Probabilistic Interval-based Event Calculus (PIEC) [16] consumes the output of such a point-based recognition, in order to compute the 'probabilistic maximal intervals' of complex events, i.e., the maximal intervals during which a complex event is said to take place, with a probability above a given threshold. Below, we define 'probabilistic maximal intervals'; then, we present the way PIEC detects such intervals in linear time.

Definition 1. *The **probability of interval** $I_{CE}=[i,j]$ of a complex event with $length(I_{CE})=j-i+1$ time-points, is defined as*

$$P(I_{CE}) = \frac{\sum_{k=i}^{j} P(\mathsf{holdsAt}(CE,\ k))}{length(I_{CE})},$$

where $\mathsf{holdsAt}(CE,k)$ is an Event Calculus predicate which signifies the occurrence of the complex event at time-point k.

In other words, the probability of an interval of some complex event is equal to the average of the event's probabilities at the time-points that it contains. A key concept of PIEC is that of probabilistic maximal interval:

Definition 2. *A **probabilistic maximal interval** $I_{CE}=[i,j]$ of a complex event is an interval such that, given some threshold $\mathscr{T} \in [0,1]$, $P(I_{CE}) \geq \mathscr{T}$, and there is no other interval I'_{CE} such that $P(I'_{CE}) \geq \mathscr{T}$ and I_{CE} is a sub-interval of I'_{CE}.*

Probabilistic maximal intervals (PMIs) may be overlapping. To choose an interval among overlapping PMIs of the same complex event, PIEC computes the 'credibility' of each such interval – see [16].

Given a dataset of n instantaneous complex event probabilities $In[1..n]$ and a threshold $\mathscr{T}$, PIEC infers all PMIs of that complex event in linear-time. To achieve this, PIEC constructs:

- The $L[1..n]$ list containing each element of In subtracted by the given threshold $\mathscr{T}$, i.e., $\forall\, i \in [1,n]$, $L[i]=In[i]-\mathscr{T}$. Note that an interval I_{CE} satisfies the condition $P(I_{CE}) \geq \mathscr{T}$ iff the sum of the corresponding elements of list L is non-negative.
- The $prefix[1..n]$ list containing the cumulative or prefix sums of list L, i.e., $\forall\, i \in [1,n]$, $prefix[i]=\sum_{j=1}^{i} L[j]$.

Table 1: PIEC with threshold $\mathscr{T} = 0.5$.

Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
In	0	0.5	0.7	0.9	0.4	0.1	0	0	0.5	1
L	-0.5	0	0.2	0.4	-0.1	-0.4	-0.5	-0.5	0	0.5
prefix	-0.5	-0.5	-0.3	0.1	0	-0.4	-0.9	-1.4	-1.4	-0.9
dp	0.1	0.1	0.1	0.1	0	-0.4	-0.9	-0.9	-0.9	-0.9

- The $dp[1..n]$ list, where $\forall\ i \in [1,n]$ we have that $dp[i] = \max_{i \leq j \leq n}(prefix[j])$. The elements of the *dp* list are calculated by traversing the *prefix* list in reverse order.

Table 1 presents an example dataset $In[1..10]$, along with the lists calculated by PIEC for $\mathscr{T} = 0.5$. In this example, there are three PMIs: $[1,5]$, $[2,6]$ and $[8,10]$.

PIEC processes a dataset sequentially using two pointers, s and e, indicating, respectively, the starting point and ending point of a potential PMI. Furthermore, PIEC uses the following variable:

$$dprange[s,e] = \begin{cases} dp[e] - prefix[s-1] & \text{if } s > 1 \\ dp[e] & \text{if } s = 1 \end{cases} \tag{1}$$

Substituting in the above formulation *prefix* and *dp* with their respective definitions, we derive that $dprange[s,e]$ expresses the maximum sum that may be achieved by adding the elements of list L from s to some $e^* \geq e$, i.e.:

$$dprange[s,e] = \max_{e \leq e^* \leq n}(L[s] + \cdots + L[e^*]). \tag{2}$$

The following entailment is a corollary of equation (2):

$$dprange[s,e] \geq 0 \Rightarrow \exists e^* : e^* \geq e \ \text{ and } \sum_{s \leq i \leq e^*} L[i] \geq 0. \tag{3}$$

Consequently, $[s,e^*]$ is a potential PMI. In this case, PIEC increments the e pointer until *dprange* becomes negative. When *dprange* becomes negative, PIEC produces the following PMI: $[s,e-1]$. Once a PMI is computed, PIEC increments the s pointer and re-calculates *dprange*. By repeating this process, PIEC computes all PMIs of a given dataset.

Example 1. Consider the dataset presented in Table 1 and a threshold $\mathscr{T} = 0.5$. Initially, $s = e = 1$ and PIEC calculates that $dprange[1,1] = 0.1 \geq 0$. Then, PIEC increments e as long as *dprange* remains non-negative. This holds until $e = 6$ when $dprange[1,6] = -0.4$. At that point, PIEC produces the PMI $[1,5]$ and increments s. Subsequently, PIEC calculates that $dprange[2,6] = 0.1$ and thus increments e, i.e., e becomes 7 while s remains equal to 2. $dprange[2,7] = -0.4 < 0$ and, accordingly, PIEC produces the PMI $[2,6]$ and increments s, i.e., $s = 3$. $dprange[s,7] < 0$ holds $\forall s \in [3,7]$. Hence, PIEC increments s until $s = 8$ when $dprange[8,7] = 0$. Note that for all time-points t we have $dprange[t+1,t] = dp[t] - prefix[t] \geq 0$ — see the definition of *dp*. Hence, PIEC avoids such erroneous pointer values, i.e., $s > e$, by incrementing e. Here, e increases as long as $dprange[8,e] \geq 0$. This holds for every subsequent time-point of the dataset. Finally, PIEC produces the PMI $[8,10]$ as $P([8,10]) \geq \mathscr{T}$ and there is no subsequent time-point to add. Summarising, PIEC computes all PMIs of $In[1..10]$. □

Table 2: PIEC operating on data batches.

Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
prefix	-0.5	-0.5	-0.3	0.1	-0.1	-0.5	-1	-1.5	0	0.5
dp	0.1	0.1	0.1	0.1	-0.1	-0.5	-1	-1.5	0.5	0.5

By computing PMIs, PIEC improves upon point-based recognition in the presence of noisy instantaneous probability fluctuations, and in the common case of non-abrupt probability change. See [16] for an empirical analysis supporting these claims. On the other hand, PIEC was designed to operate in a batch mode, as opposed to an online setting where data stream into the recognition system. Consider the example below.

Example 2. Assume that the dataset shown in Table 1 arrives in three batches: $In[1..4]$, $In[5..8]$ and $In[9,10]$. Table 2 shows the elements of the *prefix* and *dp* lists in this case. $prefix[5]$ e.g. is now equal to $L[5]=-0.1$, since the values of the $L[1..4]$ list are not available at the second data batch. Note that the elements of list L do not change (and are presented in Table 1). Given the first data batch $In[1..4]$, PIEC, starting from time-point 1, increments pointer e as long as $dprange[1,e] \geq 0$. This condition holds for every time-point in the first batch. Hence, PIEC computes the interval $[1,4]$. Considering the second data batch $In[5..8]$, PIEC does not compute any interval, as every probability in $In[5..8]$ is lower than $\mathscr{T}$. For the third batch $In[9,10]$, PIEC initiates with $s=e=9$ and subsequently computes the interval $[9,10]$, as $dprange[9,10] \geq 0$. □

Neither of the intervals $[1,4]$ and $[9,10]$ computed in the above example is a PMI. The former could have been extended to the right by one time-point, if the next data batch was foreseen. The latter could have started from time-point 8, if that time-point had been stored for future use. Additionally, the PMI $[2,6]$ was ignored entirely. One way to address these issues would be to re-iterate over all data received so far upon the receipt of each new data batch. The computational cost of such a strategy, however, is not acceptable in streaming environments (as will be shown in our empirical analysis).

2.4 Online PIEC

We present an extension of PIEC, called *online Probabilistic Interval-based Event Calculus* (oPIEC), which operates on data batches $In[i..j]$, with $i \leq j$, i.e., oPIEC processes each incoming data batch and then discards it. oPIEC identifies the minimal set of time-points that need to be cached in memory in order to guarantee correct LTA recognition. These time-points are cached in the '*support set*' and express the starting points of potential PMIs, i.e., PMIs that may end in the future. For instance, after processing the first data batch $In[1..4]$ in Example 2, oPIEC would have cached time-points $t=1$ and $t=2$ in the support set, thus allowing for the computation of PMIs $[1,5]$ and $[2,6]$ in the future. On the contrary, oPIEC would not have cached time-points $t=3$ and $t=4$, because, given that $\mathscr{T}=0.5$, a PMI cannot start from any of these points, irrespective of the data that may arrive after the first batch.

Upon the arrival of a data batch $In[i..j]$, oPIEC computes the values of the $L[i..j]$, $prefix[i..j]$, and $dp[i..j]$ lists. To allow for correct reasoning, the last *prefix* value of a batch is transferred to the next one. Consequently, the *prefix* value of the first time-point of a batch, $prefix[i]$, is set to $prefix[i-1]+L[i]$. (For the first batch, we have, as in PIEC, $prefix[1]=L[1]$.) This way, the computation of the values of $prefix[i..j]$ and $dp[i..j]$ is not affected by the absence of the data prior to i. Subsequently, oPIEC performs the following steps:

1. It computes intervals starting from a time-point in the support set.

2. It computes intervals starting from a time-point in the current batch.

3. It identifies the elements of the current batch that should be cached in the support set.

Step 2 is performed by means of PIEC (see Section 3.3). In what follows, we present first Step 3 and then move to Step 1.

Support Set

The support set comprises a set of tuples of the form $(t, \mathit{prev\_prefix}[t])$, where t is a time-point and $\mathit{prev\_prefix}[t]$ expresses t's previous *prefix* value, which is defined as follows:

$$\mathit{prev\_prefix}[t] = \begin{cases} \mathit{prefix}[t-1] & \text{if } t > 1 \\ 0 & \text{if } t = 1 \end{cases} \tag{4}$$

With the use of $\mathit{prev\_prefix}[t]$, oPIEC is able to compute $\mathit{dprange}[t,t']$ for any future time-point t', and thus determine whether t is the starting point of a PMI. For example, the arrival of a time-point $t' > t$ for which $dp[t'] \geq \mathit{prev\_prefix}[t]$ implies that $\mathit{dprange}[t,t'] \geq 0$ (see equations (1) and (4)), and hence indicates that t is the starting point of a PMI that may end either at t' or at a later time-point (see corollary (3)).

ALGORITHM 1: support_set_update$(\mathit{ignore_value},\ \mathit{support_set})$

```
for $t \in In[i..j]$ do
    if $\mathit{prev\_prefix}[t] < \mathit{ignore\_value}$ then
        $\mathit{support\_set} \xLeftarrow{add} (t, \mathit{prev\_prefix}[t])$
        $\mathit{ignore\_value} \leftarrow \mathit{prev\_prefix}[t]$
    end if
end for
return $(\mathit{ignore\_value}, \mathit{support\_set})$
```

Algorithm 1 identifies the time-points of a data batch $In[i..j]$ that should be cached in the support set. For each time-point $t \in In[i..j]$, we check whether $\mathit{prev\_prefix}[t]$ is less than the *ignore_value*—this variable expresses the lowest *prev_prefix* value found so far. If $\mathit{prev\_prefix}[t]$ is less than the *ignore_value*, then we append $(t, \mathit{prev\_prefix}[t])$ to the support set, and set the *ignore_value* to $\mathit{prev\_prefix}[t]$. A formal justification for this behaviour is given after the following example.

Example 3. Consider the dataset of the previous examples, arriving in three batches, $In[1..4]$, $In[5..8]$, and $In[9,10]$, as in Example 2. The values of the *prefix* list are shown in Table 1—recall that operating on data batches, as opposed to all data received so far, does not affect oPIEC's computation of the *prefix* list. Algorithm 1 processes every time-point of each batch sequentially. For $t = 1$, we have $\mathit{prev\_prefix}[1] = 0 < \mathit{ignore\_value}$, since, initially, $\mathit{ignore\_value} = +\infty$. Thus, the tuple $(1, \mathit{prev\_prefix}[1] = 0)$ is added to the support set and the *ignore_value* is set to 0. Next, $t = 2$ and $\mathit{prev\_prefix}[2] = -0.5 < \mathit{ignore\_value}$. Therefore, Algorithm 1 caches the tuple $(2, -0.5)$ and updates the *ignore_value*. The remaining time-points of the first batch are not added to the support set as their *prev_prefix* value does not satisfy the condition $\mathit{prev\_prefix}[t] < \mathit{ignore\_value}$. By processing the remaining batches in a similar way, we get the following support sets:

- $[(1,0),(2,-0.5)]$; computed after processing batch $In[1..4]$.

- $[(1,0),(2,-0.5),(8,-0.9)]$; computed after processing batch $In[5..8]$.
- $[(1,0),(2,-0.5),(8,-0.9),(9,-1.4)]$; computed after processing batch $In[9..10]$.

□

This example illustrates that oPIEC caches the time-points with the currently minimal *prev_prefix* value, and no other time-points. A time-point t may be the starting point of a PMI iff:

$$\forall t_{prev} \in [1,t),\ prev\_prefix[t_{prev}] > prev\_prefix[t] \tag{5}$$

Theorem 1. *If $[t_s,t_e]$ is a PMI, then t_s satisfies condition* (5).

Proof. Suppose that t_s does not satisfy condition (5). Then,

$$\exists t'_s : t'_s < t_s \text{ and } prev\_prefix[t'_s] \leq prev\_prefix[t_s] \tag{6}$$

We have that:

$$\begin{aligned} dprange[t'_s,t_e] &= dp[t_e] - prefix[t'_s - 1] \\ &= dp[t_e] - prev\_prefix[t'_s] \\ &\overset{\text{from ineq. (6)}}{\geq} dp[t_e] - prev\_prefix[t_s] \\ &= dprange[t_s,t_e] \geq 0 \end{aligned}$$

Note that $dprange[t_s,t_e] \geq 0$ because $[t_s,t_e]$ is a PMI.

The fact that $dprange[t'_s,t_e] \geq 0$, as shown above, indicates that $\exists t'_e : t'_e \geq t_e$ and $[t'_s,t'_e]$ is a PMI — see corollary (3). Additionally, $[t_s,t_e]$ is a sub-interval of $[t'_s,t'_e]$ since $t'_s < t_s$. Therefore, by Definition 2, $[t_s,t_e]$ is not a PMI. By contradiction, t_s must satisfy condition (5). ■

Note that a time-point t may satisfy condition (5) and not be the starting point of a PMI in a given dataset. See e.g. time-point *9* in Example 3. These time-points must also be cached in the support set because they may become the starting point of a PMI in the future. Consider again Example 3 and assume that a fourth batch arrives with $In[11]=0$. In this case, we have a new PMI: $[9,11]$.

Theorem 2. *oPIEC adds a time-point t to the support set iff t satisfies condition* (5).

Proof. oPIEC adds a time-point t to the support set iff the condition of the *if* statement shown in line 2 of Algorithm 1 is satisfied. When this condition, $prev\_prefix[t] < ignore\_value$, is satisfied, the *ignore_value* is set to $prev\_prefix[t]$. Since Algorithm 1 handles every time-point in chronological order, in its i^{th} iteration, *ignore_value* will be equal to the minimum *prev_prefix* value of the first i time-points. So, a time-point's *prev_prefix* value is less than the *ignore_value* iff it satisfies condition (5). ■

According to Theorem 2, therefore, oPIEC caches in the support set the *minimal* set of time-points that guarantees correct event recognition, irrespective of the data that may arrive in the future.

ALGORITHM 2: interval_computation(dp, $support\_set$, $intervals$)

```
1:  $s \leftarrow 1, e \leftarrow 1, flag \leftarrow false$
2:  while $s \le support\_set.length$ and $e \le dp.length$ do
3:    if $dp[e] \ge support\_set[s].prev\_prefix$ then
4:      $flag \leftarrow true$
5:      $e \mathrel{+}= 1$
6:    else
7:      if $flag == true$ then
8:        $intervals \xLeftarrow{add} (support\_set[s].timepoint, e-1)$
9:      end if
10:     $flag \leftarrow false$
11:     $s \mathrel{+}= 1$
12:   end if
13: end while
14: if $flag == true$ then
15:   $intervals \xLeftarrow{add} (support\_set[s].timepoint, e-1)$
16: end if
17: return $intervals$
```

Table 3: oPIEC operating on data batches.

Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
prev_prefix	0	-0.5	-0.5	-0.3	0.1	0	-0.4	-0.9	-1.4	-1.4
dp	0.1	0.1	0.1	0.1	0	-0.4	-0.9	-1.4	-0.9	-0.9

Interval Computation

We now describe how oPIEC computes PMIs using the elements of the support set. Algorithm 2 shows this process. oPIEC uses a pointer s to traverse the support set, and a pointer e to traverse the dp list of the current data batch; the elements of the support set and all lists maintained by oPIEC are temporally sorted. Algorithm 2 starts from the first element s of the support set and the first time-point e of the current data batch, and checks if the interval $[support\_set[s].timepoint, e]$ is or may be extended to a PMI. The condition in line 3 of Algorithm 2 essentially checks whether

$$dprange[support\_set[s].timepoint, e] \tag{7}$$

is non-negative. If it is non-negative, then a PMI starts at $support\_set[s].timepoint$. The Boolean variable *flag* is set to *true* and, subsequently, e is incremented as an attempt to find the ending point of the PMI starting at $support\_set[s].timepoint$.

If the value of expression (7) is negative, then the interval $[support\_set[s].timepoint, e]$ is not a PMI, and there is no point extending it to the right. Consequently, oPIEC checks whether the s, e pointers were pointing to a PMI in the previous iteration. If they were, oPIEC adds the interval of the previous iteration, i.e., $[support\_set[s].timepoint, e-1]$, to the list of PMIs (lines 7-8). Then, it sets *flag* to *false*, and increments s, as no other PMI may be found that starts at the current element of the support set.

Example 4. We complete Example 3 by presenting the interval computation process for the same dataset arriving in batches $In[1..4]$, $In[5..8]$, and $In[9..10]$. Table 3 displays the values of the dp list as computed by oPIEC, as well as the *prev_prefix* values, aiding the presentation of the example. Upon the arrival of the first batch $In[1..4]$, the support set is empty, and thus Algorithm 2 does not compute any interval. When the second batch $In[5..8]$ arrives, the support set is $[(1,0),(2,-0.5)]$ (see Example 3). Hence, Algorithm 2 initializes pointer s to 1 and pointer e to 5. Since $dp[5] \geq prev\_prefix[1]$, the *flag* becomes *true* and e is incremented (see lines 4–5 of Algorithm 2). In the following iteration, $dp[6] < prev\_prefix[1]$ and thus Algorithm 2 produces the PMI $[1,5]$. Next, s is set to 2. Because $dp[6] > prev\_prefix[2]$, Algorithm 2 decides that there is a PMI starting from $t=2$. However, it fails to extend it in the following iteration. Therefore, Algorithm 2 produces the PMI $[2,6]$ and terminates for this data batch. When the third batch $In[9..10]$ arrives, the support set is $[(1,0),(2,-0.5),(8,-0.9)]$. Since $dp[9]=-0.9$ is less than the *prev_prefix* values of the first two elements of the support set, Algorithm 2 skips these elements, and sets s to the third element, i.e., $support\_set[s].timepoint=8$. Following similar reasoning, Algorithm 2 increments e and eventually produces the PMI $[8,10]$. □

In this example dataset, oPIEC computes all PMIs. This is in contrast to PIEC that does not compute any of the PMIs, given the partitioned stream of Example 2.

2.5 Bounded Support Set

The key difference between the complexity of oPIEC and PIEC is that the former takes into consideration the support set in interval computation. The size of the support set depends on the data stream of instantaneous probabilities and the value of the threshold $\mathscr{T}$. In brief, high threshold values increase the size of the support set, and vice versa. In any case, to allow for efficient reasoning in streaming environments, the support set needs to be bound. To address this issue, we present oPIEC^{b}, which introduces an algorithm to decide which elements of the support set should be deleted, in order to make room for new ones. Consequently, when compared to PIEC and oPIEC, oPIEC^{b} may detect shorter intervals and/or fewer intervals.

When a time-point t satisfying condition (5) arrives, i.e., t may be the starting point of a PMI, oPIEC^{b} will attempt to cache it in the support set, provided that the designated support set limit is not exceeded. If it is exceeded, then oPIEC^{b} decides whether to cache t, replacing some older time-point in the support set, by computing the ‘score range’, an interval of real numbers defined as:

$$score\_range[t] = [prev\_prefix[t], prev\_prefix[prev_s[t]]) \tag{8}$$

The *score_range* is computed for the time-points in set S, i.e., the time-points already in the support set, and the time-points that are candidates for the support set. All the time-points in S satisfy condition (5) and thus are in descending *prev_prefix* order. $prev_s[t]$ is the time-point before t in S.

With the use of $score\_range[t]$, oPIEC^{b} computes the likelihood that a time-point t, satisfying condition (5), will indeed become the starting point of a PMI. Suppose, e.g., that a time-point $t_e > t$ arrives later in the stream, with $dp[t_e] \in score\_range[t]$, i.e., $prev\_prefix[t] \leq dp[t_e] < prev\_prefix[prev_s[t]]$ (see eq. (8)). In this case, we have $dprange[t,t_e] \geq 0$ and $dprange[prev_s[t],t_e] < 0$ (see eq. (1) and (4)). Hence, a PMI will start from t. The longer the $score\_range[t]$, i.e., the longer the distance between $prev\_prefix[t]$ and $prev\_prefix[prev_s[t]]$, the more likely it is, intuitively, that a future time-point t_e will arrive with $dp[t_e] \in score\_range[t]$, and thus that t will be the starting point of a PMI. Consequently, oPIEC^{b} stores in the support set the elements with the longer score range.

Algorithm 3 presents the support set maintenance algorithm for a support set of size m and k candidate elements. oPIEC^{b} gathers every element of the support set and the candidate tuples in set S. The goal is to compute the elements of S with the shortest score ranges, in order to cache the remaining elements in the new support set.

ALGORITHM 3: support_set_maintenance(*support_set*, *new_tuples*)

```
$m \leftarrow support\_set.length,\ \ k \leftarrow new\_tuples.length$
$S \leftarrow \emptyset,\ \ S \overset{add}{\Longleftarrow} support\_set,\ \ S \overset{add}{\Longleftarrow} new\_tuples$
$counter \leftarrow 1,\ temp\_array \leftarrow \emptyset$
for $(t, prev\_prefix[t]) \in S$ do
    $score\_range\_size \leftarrow prev\_prefix[prev_s[t]] - prev\_prefix[t]$
    if $counter \leq k$ then
        $temp\_array \overset{add}{\Longleftarrow} (t, prev\_prefix[t], score\_range\_size)$
        if $counter == k$ then
            $longest\_elem \leftarrow find\_longest\_range(temp\_array)$
        end if
    else
        if $score\_range\_size < longest\_elem.score\_range$ then
            $temp\_array.delete(longest\_elem)$
            $temp\_array \overset{add}{\Longleftarrow} (t, prev\_prefix[t], score\_range\_size)$
            $longest\_elem \leftarrow find\_longest\_range(temp\_array)$
        end if
    end if
    $counter \mathrel{+}= 1$
end for
for $elem \in temp\_array$ do
    $S.delete(elem)$
end for
$support\_set \leftarrow S$
return $support\_set$
```

Example 5. Consider the dataset $In[1..5]$ presented in Table 4. With a threshold value $\mathscr{T}$ of 0.5, this dataset has a single PMI: $[2,5]$. Assume that the data arrive in two batches: $In[1..4]$ and $In[5]$. Therefore, given an unbounded support set, oPIEC would have cached time-points *1*, *2*, *3* and *4* into the support set.

Assume now that the limit of the support set is set to two elements. oPIEC^b processes $In[1..4]$ to detect the time-points that may be used as starting points of PMIs, i.e., those satisfying condition (5). These are time-points *1*, *2*, *3* and *4*. In order to respect the support set limit, oPIEC^b has to compute the score ranges:

- $score\_range[1]$ is set to $[0,+\infty)$ since $t=1$ has no predecessor in the support set.
- $score\_range[2] = [prev\_prefix[2], prev\_prefix[1]) = [-0.5, 0)$.
- $score\_range[3] = [prev\_prefix[3], prev\_prefix[2]) = [-0.7, -0.5)$.
- $score\_range[4] = [prev\_prefix[4], prev\_prefix[3]) = [-0.9, -0.7)$.

Given these *score_range* values, oPIEC^b caches the tuples $(1,0)$ and $(2,-0.5)$ in the support set, since these are the elements with the longest score range. oPIEC^b chooses time-point $t=2$, e.g., over time-points $t=3$ and $t=4$ for the

Table 4: oPIEC$^{\mathrm{b}}$ in action.

Time	**1**	**2**	**3**	**4**	**5**
In	0	0.3	0.3	0.6	0.9
L	-0.5	-0.2	-0.2	0.1	0.4
prefix	-0.5	-0.7	-0.9	-0.8	-0.4
prev_prefix	0	-0.5	-0.7	-0.9	-0.8
dp	-0.5	-0.7	-0.8	-0.8	-0.4

support set, because it is more likely that a future time-point t' will have a $dp[t']$ value within $\mathit{score\_range}[2]$ than within $\mathit{score\_range}[3]$ or $\mathit{score\_range}[4]$.

With such a support set, oPIEC$^{\mathrm{b}}$ is able to perform correct event recognition, i.e., compute PMI $[2,5]$, upon the arrival of the second data batch $\mathit{In}[5]$. Note that $\mathit{dprange}[2,5]=0.1 \geq 0$ and $t=5$ is the last time-point of the data stream so far. Also, for the other time-point of the support set, $t=1$, we have $\mathit{dprange}[1,5]=-0.4 < 0$, and hence a PMI cannot start from $t=1$.

Following a somewhat naive maintenance strategy, i.e., deleting the oldest element of the support set to make room for a new one, as opposed to the strategy of oPIEC$^{\mathrm{b}}$ based on *score_range*, would have generated, after processing $\mathit{In}[1..4]$, the following support set: $[(3,-0.7),(4,-0.9)]$. Consequently, upon the arrival of $\mathit{In}[5]$, the interval $[3,5]$ would have been computed, which is not a PMI. □

2.6 Experimental Evaluation on a Benchmark Activity Recognition Dataset

Datasets

To evaluate oPIEC$^{\mathrm{b}}$ we used CAVIAR[1], a benchmark activity recognition dataset. CAVIAR includes 28 videos with 26,419 video frames in total. The videos are staged, i.e., actors walk around, sit down, meet one another, fight, etc. Each video has been manually annotated by the CAVIAR team in order to provide the ground truth for both simple events, taking place on individual video frames, as well as composite events. The input to the activity recognition system consists of the simple events 'inactive', i.e., standing still, 'active', i.e., non-abrupt body movement in the same position, 'walking', and 'running', together with their time-stamps, i.e., the video frame in which the event took place. The dataset also includes the coordinates of the tracked people and objects as pixel positions at each time-point, as well as their orientation. Given such an input, the task is to recognise complex events such as two people having a meeting or fighting.

The CAVIAR dataset includes inconsistencies, as the members of the CAVIAR team that provided the annotation did not always agree with each other [95, 134]. To allow for a more demanding evaluation of activity recognition systems, Skarlatidis et al. [134] injected additional types of noise into CAVIAR, producing the following datasets:

- *Smooth noise*: a subset of the simple events have probabilities attached, generated by a Gamma distribution with a varying mean. All other simple events have no probabilities attached, as in the original dataset.

[1] Section "Clips From INRIA" of `http://groups.inf.ed.ac.uk/vision/CAVIAR/CAVIARDATA1/`

- *Strong noise*: probabilities have been additionally attached to contextual information (coordinates and orientation) using the same Gamma distribution, and spurious simple events that do not belong to the original dataset have been added using a uniform distribution.

Note that CAVIAR is a surrogate dataset we used as a proof-of-concept of the applicability of our system. We plan on employing oPIEC^{b} for experiments on maritime datasets.

In the analysis that follows, we use the original CAVIAR dataset as well as the 'smooth noise' and 'strong noise' versions. The target complex events are 'meeting together' and 'fighting'. All versions of CAVIAR, along with the definitions of these events in the Event Calculus, are publicly available[2].

Predictive Accuracy

The aim of the first set of experiments was to compare the predictive accuracy of oPIEC^{b} against that of PIEC. We focused our comparison on four cases in which PIEC has a noticeably better performance than the underlying system performing point-based event recognition. In these experiments, we use two such systems, Prob-EC [134] and OSLα [104, 102]. Prob-EC is an implementation of the Event Calculus in ProbLog [84], designed to handle data uncertainty. OSLα is a supervised learning framework employing the Event Calculus in Markov Logic [136], to guide the search for weighted LTA definitions. Our comparison concerns the following cases:

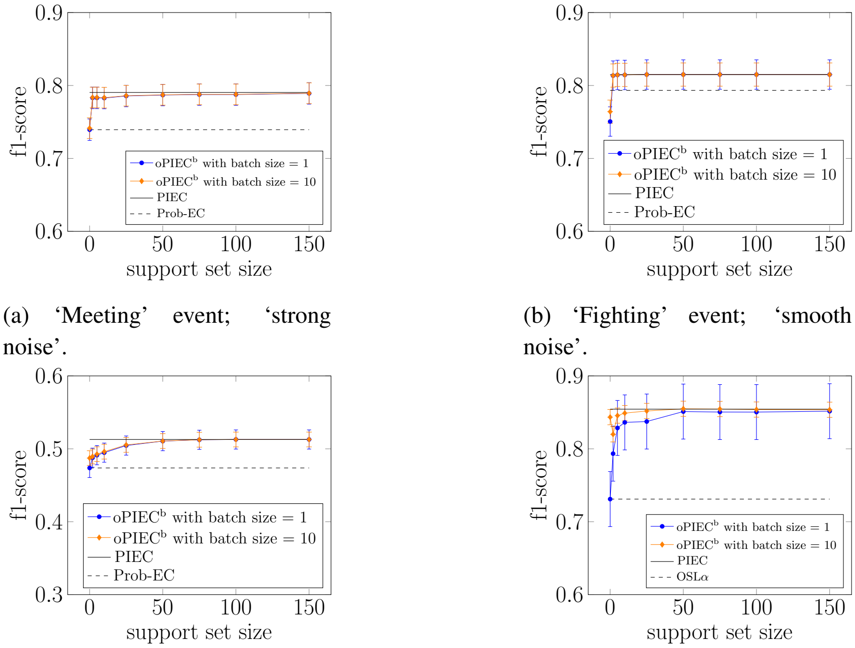
- Prob-EC recognising the 'meeting' event when operating on the 'strong noise' dataset.
- Prob-EC recognising the 'fighting' event when operating on the 'smooth noise' and the 'strong noise' datasets. The complex event definitions used by Prob-EC were manually constructed and do not have weights attached [134].
- OSLα recognising the 'meeting' event when operating on the original CAVIAR dataset. Prior to recognition, OSLα was trained to construct the complex event definition in the form of weighted Event Calculus rules, given the annotation provided by the CAVIAR team (see [104] for the setup of the training process).

In each case, PIEC and oPIEC^{b} consumed the output of point-based event recognition.

Figure 1 shows the experimental results in terms of the f1-score, which was calculated using the ground truth of CAVIAR. Each of the diagrams of Figure 1 shows the performance of point-based event recognition (Prob-EC in Figures 1(a)–(c) and OSLα in Figure 1(d)), PIEC, oPIEC^{b} operating on data batches of 1 time-point, i.e., performing reasoning at each time-point and then discarding it, unless cached in the support set (see 'batch size=1' in Figure 1), and oPIEC^{b} operating on batches of 10 time-points. We used the threshold value leading to the best performance for each system. For recognising 'meeting' and 'fighting' under 'strong noise', we set $\mathscr{T}=0.5$, while in the recognition of 'meeting' with the weighted definitions of OSLα, we set $\mathscr{T}=0.7$. In these cases, where the same threshold value is used for point-based and interval-based recognition, the performance of oPIEC^{b}, when operating on batches of 1 time-point with an empty support set, amounts to the performance of point-based recognition. See Figures 1(a), (c) and (d). In the case of 'fighting' under 'smooth noise', the best performance for Prob-EC is achieved for a different threshold value than that leading to the best performance for PIEC. Thus, Prob-EC operated with $\mathscr{T}=0.5$, while PIEC and oPIEC^{b} operated with $\mathscr{T}=0.9$.

Figure 1 shows that oPIEC^{b} reaches the performance of PIEC, even with a small support set ($\leq$ *50*), and with the

[2] https://anskarl.github.io/publications/TPLP15/



(a) ‘Meeting’ event; ‘strong noise’.

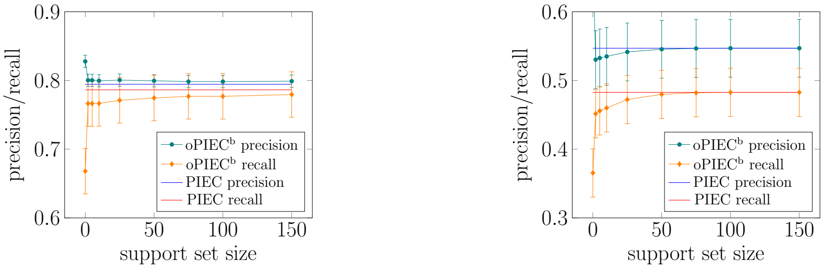
(b) ‘Fighting’ event; ‘smooth noise’.

(c) ‘Fighting’ event; ‘strong noise’.

(d) ‘Meeting’ event; weighted pattern.

Figure 1: Predictive accuracy.

common option for streaming applications of the smallest batch size (= *1*). In other words, oPIEC^b outperforms point-based recognition, requiring only a small subset of the data.

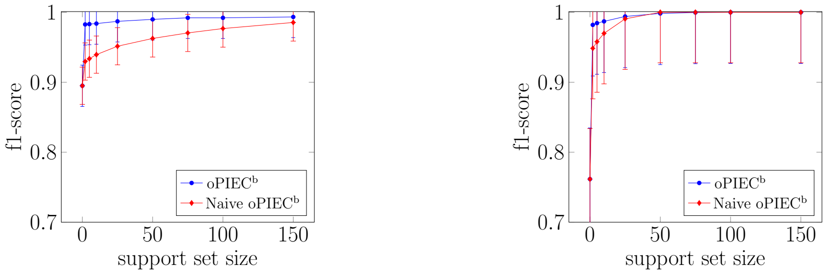


(a) ‘Meeting’ event; ‘strong noise’.

(b) ‘Fighting’ event; ‘strong noise’.

Figure 2: Precision and recall. For oPIEC^b, batch size = 1.

Figure 2 shows the precision and recall scores of oPIEC^b (operating on batches of 1 time-point) and PIEC, in the task of recognising the ‘meeting’ and ‘fighting’ events in the ‘strong’ noise dataset. In both cases, Prob-EC provides the instantaneous probabilities. In the case of ‘fighting’, both precision and recall increase for oPIEC^b as the support set increases, eventually reaching the precision and recall of PIEC. In the case of ‘meeting’, the precision of oPIEC^b is initially higher than that of PIEC, and drops, as the support set increases, approaching the precision of PIEC. Recall that, due to the bounded support set, oPIEC^b may detect shorter intervals and/or fewer intervals than PIEC. In this particular case, the limit on the size of the support set leads, also, to correcting some of PIEC’s errors, i.e. reducing its false positives.

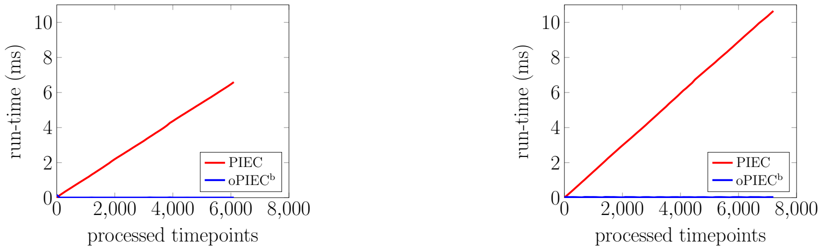


(a) ‘Meeting’ event; ‘strong noise’.

(b) ‘Fighting’ event; ‘strong noise’.

Figure 3: Support set maintenance. Batch size = 1.

The aim of the next set of experiments was to evaluate the effects of the support set maintenance strategy of oPIEC^{b}. Figure 3 compares the performance of oPIEC^{b} against that of a naive maintenance strategy, according to which the oldest element of the support set is deleted to make room for a new one. In these experiments, the ground truth was the output of PIEC. As shown in Figure 3, in most cases the strategy of oPIEC^{b}, based on the *score_range*, leads to a better approximation of the performance of PIEC.



(a) ‘Smooth noise’.

(b) ‘Strong noise’.

Figure 4: Recognition times for ‘fighting’: PIEC vs oPIEC^{b} with a support set limit of 60 elements.

Recognition Times

In these experiments, the aim was to compare the recognition times of oPIEC^{b} and PIEC in a streaming setting, i.e., when the recognition system must respond as soon as a data batch arrives. To achieve this, we instructed Prob-EC to recognise the ‘fighting’ event in the videos of the ‘smooth’ and ‘strong noise’ datasets with instances of this complex event. Then, we provided the output of Prob-EC in batches of 1 time-point to oPIEC^{b} and PIEC, for interval-based recognition. Upon the arrival of a data batch, PIEC was instructed to reason over all data collected so far, as this is the only way to guarantee correct PMI computation. oPIEC^{b} was instructed to operate on a support set limited to 60 elements, as this is sufficient for reaching the accuracy of PIEC. Figure 4 shows the experimental results. Note that the ‘strong noise’ dataset is larger due to the injection of spurious simple events. As shown in Figure 4, oPIEC^{b} has a constant (low) cost, in contrast to the cost of PIEC, which increases as data stream into the recognition system. The comparison of recognition times under different configurations (complex events, datasets, underlying point-based recognition system) yields similar results, and is not shown here to save space.

2.7 Summary and Further Research

We presented oPIEC^b, an algorithm for online complex event recognition under uncertainty. oPIEC^b identifies the minimal set of data points that need to be cached in memory, in order to guarantee correct activity recognition in a streaming setting. Moreover, oPIEC^b adopts a method for further reducing the cached data points, according to memory limits. This leads to highly efficient recognition, while at the same time minimising any effect on correctness. Our empirical evaluation on a surrogate activity recognition dataset showed that oPIEC^b achieves a higher predictive accuracy than point-based recognition models that have been manually constructed (Prob-EC) or optimised by means of relational learning (OSLα). Moreover, oPIEC^b reaches the predictive accuracy of batch interval-based event recognition (PIEC), with a small support set, thus supporting streaming applications.

For future work, we aim at developing new support set maintenance techniques that reduce further the errors of PIEC. Moreover, we plan to compare oPIEC^b with additional machine learning frameworks, such as [79] and test our approach on maritime datasets.

3 Online Structure & Weight Learning of Complex Event Patterns

3.1 Introduction

Complex Event Recognition (CER) systems [36] detect occurrences of *complex events* (CEs) in streaming input, defined as spatio-temporal combinations of *simple events* (e.g. sensor data), using a set of CE patterns. Since such patterns are not always known beforehand, while existing ones often need to be updated, machine learning algorithms for the automatic construction/revision of CE patterns are highly useful. Such algorithms should ideally operate in an online fashion, by using the current CE pattern set for inference (CER) in the incoming data stream, and the labeled portions of the stream for updating the CE pattern set. Moreover, such algorithms should be resilient to noise & uncertainty, which are ubiquitous in temporal data streams [11], and support reasoning with existing domain knowledge, while taking into account commonsense phenomena [109], which often characterize dynamic application domains, such as CER.

Logic-based CER systems [19] stand up to the aforementioned challenges. They combine reasoning under uncertainty with machine learning, via Statistical Relational AI techniques [42], while they are capable of reasoning with time and change, and incorporating commonsense principles via action formalisms, such as the Event Calculus [17].

A number of online learning algorithms, capable of temporal reasoning with a set of CE patterns, while continuously updating these patterns in the face of new data, have already been proposed [79, 101]. We advance the state of the art by proposing a novel implementation of one such algorithm, called WOLED (Online Learning of Weighted Event Definitions) [80], originally based on Markov Logic Networks (MLNs), which learns CE patterns in the form of weighted rules in the Event Calculus. Our new implementation, which is based entirely on Answer Set Programming (ASP), allows to take advantage of the grounding, solving, optimization and uncertainty modeling abilities of modern answer set solvers, while employing structure learning techniques from non-monotonic Inductive Logic Programming (ILP) [43], which are easily implemented in ASP, towards more robust learning.

We compare our novel ASP-based implementation to an MLN-based one, and to crisp version of the algorithm that learns unweighted rules, on three CE datasets for *activity recognition*, *maritime surveillance* and *vehicle fleet management*. Our results demonstrate the superiority of our novel implementation, both in terms of efficiency and predictive performance.

3.2 Related Work

Event Calculus-based CER [17] was combined with MLNs in [137], in order to deal with the uncertainty of CER applications. An inherent limitation of this approach is the fact that the non-monotonic semantics of the Event Calculus is incompatible with the open-world semantics of MLNs. Therefore, performing inference with Event Calculus-based MLN theories calls for extra, costly operations, such as computing the completion of a theory [109], in order to endow the first-order logic representations on which MLNs rely with a non-monotonic semantics. We bridge this gap via translating probabilistic inference with MLNs into an optimization task in ASP, which naturally supports non-monotonic and commonsense reasoning. This also allows to delegate probabilistic temporal reasoning and machine learning tasks to sophisticated, off-the-self answer set solvers.

Translating MLN inference in ASP has been put forth in [91, 90]. This line of work is mostly concerned with theoretical aspects of the translation, limiting applications to simple, proof-of-concept examples. Although we do rely on the theoretical foundation of this work, we take a more application-oriented stand-point and investigate the usefulness of

(a)	
Predicate	**Meaning**
$\mathsf{happensAt}(E,T)$	Event E occurs at time T.
$\mathsf{initiatedAt}(F,T)$	At time T, a period of time for which fluent F holds is initiated.
$\mathsf{terminatedAt}(F,T)$	At time T, a period of time for which fluent F holds is terminated.
$\mathsf{holdsAt}(F,T)$	Fluent F holds at time T.
(b)	
The axioms of the Event Calculus	
$\mathsf{holdsAt}(F,T+1) \leftarrow$ (1) $\mathsf{initiatedAt}(F,T)$	$\mathsf{holdsAt}(F,T+1) \leftarrow$ (2) $\mathsf{holdsAt}(F,T)$, $\mathsf{not}\ \mathsf{terminatedAt}(F,T)$
(c)	**(d)**
Observations I_1 at time 1:	**Weighted CE patterns:**
$\mathsf{happensAt}(\mathit{walk}(\mathit{id_1}),\mathit{1})$. $\mathsf{happensAt}(\mathit{walk}(\mathit{id_2}),\mathit{1})$. $\mathit{coords}(\mathit{id_1},\mathit{201},\mathit{454},\mathit{1})$. $\mathit{coords}(\mathit{id_2},\mathit{230},\mathit{440},\mathit{1})$ $\mathit{direction}(\mathit{id_1},\mathit{270},\mathit{1}$ $\mathit{direction}(\mathit{id_2},\mathit{270},\mathit{1}$	$1.234\ \mathsf{initiatedAt}(\mathit{move}(X,Y),T) \leftarrow$ $\mathsf{happensAt}(\mathit{walk}(X),T)$, $\mathsf{happensAt}(\mathit{walk}(Y),T)$, $\mathit{close}(X,Y,\mathit{25},T)$, $\mathit{orientation}(X,Y,\mathit{45},T)$
Target CE instances at time 1: $\mathsf{holdsAt}(\mathit{move}(\mathit{id}_1,\mathit{id}_2),2)$ $\mathsf{holdsAt}(\mathit{move}(\mathit{id}_2,\mathit{id}_1),2)$	$0.923\ \mathsf{terminatedAt}(\mathit{move}(X,Y),T) \leftarrow$ $\mathsf{happensAt}(\mathit{inactive}(X),T)$, $\mathsf{not}\ \mathit{close}(X,Y,\mathit{30},T)$

Table 5: **(a), (b)** The basic predicates and the EC axioms. **(c)** Example CAVIAR data. For example, at time point 1 person with $\mathit{id_1}$ is *walking*, her (X,Y) coordinates are $(201,454)$ and her direction is 270°. The query atoms for time point 1 ask whether persons id_1 and id_2 are moving together at the next time point. **(d)** An example of two domain-specific axioms in the EC. E.g. the first rule dictates that *moving together* between two persons X and Y is initiated at time T if both X and Y are walking at time T, their euclidean distance is less than 25 pixel positions and their difference in direction is less than 45°. The second rule dictates that *moving together* between X and Y is terminated at time T if one of them is standing still at time T and their euclidean distance at T is greater that 30.

these ideas in challenging domains, such as CER. Another important difference from previous work on combining MLNs with ASP is that while the latter does not touch upon machine learning, we propose a methodology for learning both the structure and the weights of rules representing CE patterns, in an online fashion, using ASP tools.

Regarding machine learning, a number of algorithms in the non-monotonic branch of Inductive Logic Programming (ILP), such as XHAIL [124], TAL [21] and ILASP [88] are capable of learning Event Calculus theories. However, these algorithms are batch learners, they are thus poor matches to the online nature of CER applications. Moreover, they learn crisp logical theories, thus their ability to cope with noise and uncertainty is limited. Existing online learning algorithms [79, 101] rely on MLNs, so they suffer from the same limitations discussed earlier in this section, while a recent online learner based on probabilistic theory revision [66] is limited to Horn logic and cannot handle Event Calculus reasoning.

3.3 Background

We assume a first-order language where atoms, literals (possibly negated atoms), rules and logic programs are defined as in [59] and not denotes negation as failure. Rules, atoms, literals and programs are ground if they contain no variables. Rules are denoted by $\alpha \leftarrow \delta_1,\ldots,\delta_n$, where α is an atom and $\delta_1,\ldots,\delta_n$ a conjunction of literals. An interpretation I is a set of true ground atoms. I satisfies a ground literal a (resp. $\mathsf{not}\ a$) iff $a \in I$ (resp. $a \notin I$) and it satisfies a ground rule iff it satisfies the head, or does not satisfy the body. I is a minimal (Herbrand) model of a logic program Π iff it satisfies every ground rule in Π and none of its strict subsets has this property. I is an answer set of Π iff it is a minimal model of the program that results from the ground instances of Π, after removing all rules with a negated literal not satisfied by I, and all negative literals from the remaining rules. A choice rule is an expression of the form $\{\alpha\} \leftarrow \delta_1, \ldots, \delta_n$. with the intuitive meaning that whenever the body $\delta_1, \ldots, \delta_n$ is satisfied by an answer set I of a program that includes the choice rule, instances of the head α are arbitrarily included in I (satisfied) as well. A weak constraint is an expression of the form $: \sim \delta_1, \ldots, \delta_n.[w]$, where δ_i's are literals and w is an integer. The intuitive meaning of a weak constraint c is that the satisfaction of the conjunction $\delta_1, \ldots, \delta_n$ by an answer set I of a program that includes c incurs a cost of w for I. Inclusion of weak constraints in a program triggers an optimization process that yields answer sets of minimum cost. We refer to [59] for a formal account of choice rules and weak constraints' semantics. In what follows we use the Clingo[3] syntax for representing these constructs.

The Event Calculus is a temporal logic for reasoning about events and their effects. Its ontology comprises time points (integers), fluents, i.e. properties which have certain values in time, and events, i.e. occurrences in time that may affect fluents and alter their value. Its axioms incorporate the commonsense law of inertia, according to which fluents persist over time, unless they are affected by an event. Its basic predicates and axioms are presented in Table 5(a), (b). Axiom (1) states that a fluent F holds at time T if it has been initiated at the previous time point, while Axiom (2) states that F continues to hold unless it is terminated. Definitions of initiatedAt/2 and terminatedAt/2 predicates are provided in a application-specific manner.

Using the Event Calculus in a CER context allows to reason with CEs that have duration in time and are subject to commonsense phenomena, via associating CEs to fluents. In this case, a set of CE patterns is a set of conditions that initiate/terminate a target CE, i.e., a set of initiatedAt/2 and terminatedAt/2 rules.

As an example we use the task of activity recognition, as defined in the CAVIAR project[4]. The CAVIAR dataset consists of videos of a public space, where actors perform some activities. These videos have been manually annotated by the CAVIAR team to provide the ground truth for two types of activity. The first type, corresponding to simple events, consists of knowledge about a person's activities at a certain video frame/time point (e.g. *walking, standing still* and so on). The second type, corresponding to CEs/fluents, consists of activities that involve more than one person, for instance two people *moving together, meeting each other* and so on. The aim is to detect CEs as of combinations of simple events and additional domain knowledge, such as a person's position and direction.

Table 5(c) presents an example of CAVIAR data, consisting of *observations* for a particular time point, in the form of an interpretation I_1. A stream of interpretations is matched against a set of CE patterns (initiation/termination rules – see Table 5(d)), to infer the truth values of CE instances in time, using the Event Calculus axioms as a reasoning engine. We henceforth call the atoms corresponding to CE instances whose truth values are to be inferred/predicted, *target CE instances*. Table 2(c) presents the target CE instances corresponding to the observations in I_1. Note that at time t the corresponding target CE instances refer to $t+1$, in accordance to the Event Calculus axioms, which infer the truth value of a CE instance at a time point, base on what happens at the previous time point.

In WOLED, the CE patterns included in a logic program Π are associated with real-valued weights, defining a probability distribution over answer sets of Π. Similarly to Markov Logic, where a possible world may satisfy a subset of the formulae in an MLN, and the weights of the formulae in a unique, maximal such subset determine the probability of the possible world, an answer set of a program with weighted rules may satisfy subsets of these rules, and these rules' weights determine the answer set's probability. Based on this observation, [91] propose to assign probabilities to answer sets of a program Π with weighted rules as follows: For each interpretation I, first find the maximal subset R_I of the

[3] https://potassco.org/

[4] http://homepages.inf.ed.ac.uk/rbf/CAVIARDATA1/

weighted rules in Π that are satisfied by I. Then, assign to I a weight $W_{\Pi}(I)$ proportional to the sum of weights of the rules in R_I, if I is an answer set of R_I, else assign zero weight. Finally, define a probability distribution over answer sets of Π by normalizing these weights.

Formally, let w_r be the weight of rule r and $\mathsf{ans}(\Pi)$ the set of all interpretations I which are answer sets of R_I and which, moreover, satisfy all hard-constrained rules in Π (rules without weights). Then

$$W_{\Pi}(I) = \begin{cases} \exp\left(\sum\limits_{r \in R_I} w_r\right) & \text{if } I \in \mathsf{ans}(\Pi) \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

$$P_{\Pi}(I) = \frac{W_{\Pi}(I)}{\sum\limits_{J \in \mathsf{ans}(\Pi)} W_{\Pi}(J)} \tag{10}$$

3.4 Structure & Weight Learning in ASP

The task that WOLED addresses is to online learn the structure and weights of CE patterns, while using their current version at each point in time to perform CER in the streaming input. We adopt a standard online learning approach consisting of the following steps: at time t the learner maintains a theory H_t (weighted CE pattern set, as in Table 5(c)), has access to some static background knowledge (e.g. the axioms of the Event Calculus – Table 5(a)) and receives an interpretation I_t, consisting of a data mini-batch (as in Table 5(b)). Then (i) the learner performs inference (CER) with $B \cup H_t$ on I_t and generates a "predicted state", consisting of inferred holdsAt/2 instances of the target predicate. Via closed-world assumption, all such instances not present in the predicted state are false; (ii) if available, the true state, consisting of the actual truth values of the predicted atoms is revealed; (iii) the learner identifies erroneous predictions via comparing the predicted state to the true one, and uses these mistakes to update the structure and the weights of the CE patterns in H_t, yielding a new theory H_{t+1}.

We next discuss each of these steps and their implementation using ASP tools.

3.4.1 Generating the Inferred State

To make predictions with the weighted CE patterns in the incoming data interpretations, WOLED uses MAP (Maximum A Posteriori) probabilistic inference[5], which amounts to computing a most probable answer set $\mathscr{A}$ of $\Pi = B \cup H_t \cup I_t$. From Equations (9), (10) it follows that

$$\mathscr{A} = \underset{I \in \mathsf{ans}(\Pi)}{\operatorname{argmax}} P_{\Pi}(I) = \underset{I \in \mathsf{ans}(\Pi)}{\operatorname{argmax}} W_{\Pi}(I) = \underset{I \in \mathsf{ans}(\Pi)}{\operatorname{argmax}} \sum_{r \in R_I} w_r \tag{11}$$

[5] Marginal inference, i.e. computing the probability of each target CE instance is also possible, but it is computationally expensive since it requires a full enumeration of a program's answer sets, of utilizing techniques for sampling from such answer sets. We are not concerned with marginal inference in this work.

ALGORITHM 4: $\mathsf{MAPInference}(B, H_t, I_t)$

```
 1: $T(H_t) := \emptyset$
 2: for each CE pattern $r_i = \alpha \leftarrow \delta_1, \ldots \delta_n$ in $H_t$ with integer weight $w_i$ do
 3:     let $\mathsf{vars}(\alpha)$ be a term wrapping the variables of $\alpha$.
 4:     Add to $T(H_t)$ the following rules:
 5:     $\alpha \leftarrow \mathsf{satisfied}(\mathsf{vars}(\alpha), i)$.
 6:     $\{\mathsf{satisfied}(\mathsf{vars}(\alpha), i)\} \leftarrow \delta_1, \ldots \delta_n$.
 7:     $: \sim \mathsf{satisfied}(\mathsf{vars}(\alpha), i).\ [-w_i, \mathsf{vars}(\alpha), i]$
 8: end for
 9: Find an optimal answer set $\mathscr{A}_{opt}$ of $B \cup T(H_t) \cup I_t$.
10: return the target CE instances in $\mathscr{A}_{opt}$.
```

that is, a most probable answer set is one that maximizes the sum of weights of satisfied rules, similarly to the MLN case, for possible worlds. This is a weighted MaxSat problem that may be delegated to an answer set solver using built-in optimization tools. Since answer set solvers only optimize integer-valued objective functions, a first step is to convert the real-valued CE pattern weights to integers. We do so by scaling the weights, via multiplying them by a positive factor, while preserving their relative differences, and rounding the result to the closest integer. Note that as it may be seen from Equation (11), weight scaling by a positive factor does not alter the set of most probable answer sets, therefore, the inference result remains unaffected.

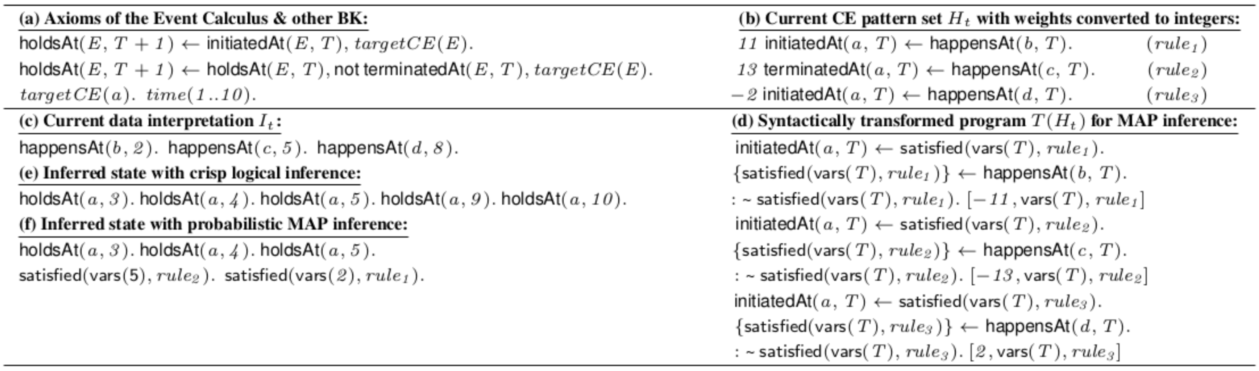
To make sure that small relative differences between weights are maintained once the weights are converted to integers, we set the scaling factor to K/d_{min}, where $d_{min} = min_{i \neq j}|w_i - w_j|$ is the smallest distance between any pair of weights and K is a large positive constant, which reduces precision loss when rounding the scaled weights to integer values.

The MAP inference/weighted MaxSat computation is realized via a standard generate-and-test ASP approach, presented in Algorithm 4, whose input is the background knowledge B, the current CE pattern set H_t and the current interpretation I_t. First, H_t is transformed into a new program, $T(H_t)$, as follows: each CE pattern r_i in H_t of the form $r_i = head_i \leftarrow body_i$ is "decomposed", so as to associate $head_i$ with a fresh predicate, $\mathsf{satisfied}/2$, wrapping $head_i$'s variables and its unique id, i (line 5, Algorithm 4). The choice rule in line 6, the "generate" part of the process, generates instances of $\mathsf{satisfied}/2$ that correspond to groundings of $body_i$. The weak constraint in line 7, the "test" part of the process, decides which of the generated $\mathsf{satisfied}/2$ instances will be included in an answer set, indicating groundings of the initial CE pattern r_i, that will be true in the inferred state.

As it may be seen from line 7, the violation of a weak constraint by an answer set $\mathscr{A}$ of $\Pi = B \cup T(H_t) \cup I_t$, i.e. the satisfaction of a ground instance of r_i by $\mathscr{A}$, incurs a cost of $-w_i$ on $\mathscr{A}$, where w_i is r_i's integer-valued weight. The optimization process triggered by the inclusion of these weak constraints in a program generates answer sets of minimum cost. During the cost minimization process, costs of $-w_i$ are actually rewards for rules with a positive w_i, whose satisfaction by an answer set, via the violation of the corresponding weak constraint, reduces the answer set's total cost. The situation is reversed for rules with a negative weight, whose corresponding weak constraint is associated with a positive cost.

Obtaining the inferred state amounts to "reading-off" target CE instances from an optimal (minimum-cost) answer set of the program $B \cup T(H_t) \cup I_t$.

Example 6. *We illustrate the inference process via the example in Figure 5. In (a) the Event Calculus axioms are presented, with an extra predicate, targetCE/1, indicating the target CE whose occurrences we wish to detect and which is subject to the effects of inertia; (b) presents a CE pattern set H_t, where we assume that the actual real-valued*



(a) Axioms of the Event Calculus & other BK:

holdsAt$(E, T+1) \leftarrow$ initiatedAt(E, T), $targetCE(E)$.
holdsAt$(E, T+1) \leftarrow$ holdsAt(E, T), not terminatedAt(E, T), $targetCE(E)$.
$targetCE(a)$. $time(1..10)$.

(b) Current CE pattern set H_t with weights converted to integers:

11 initiatedAt$(a, T) \leftarrow$ happensAt(b, T). $(rule_1)$
13 terminatedAt$(a, T) \leftarrow$ happensAt(c, T). $(rule_2)$
-2 initiatedAt$(a, T) \leftarrow$ happensAt(d, T). $(rule_3)$

(c) Current data interpretation I_t:

happensAt$(b, 2)$. happensAt$(c, 5)$. happensAt$(d, 8)$.

(e) Inferred state with crisp logical inference:

holdsAt$(a, 3)$. holdsAt$(a, 4)$. holdsAt$(a, 5)$. holdsAt$(a, 9)$. holdsAt$(a, 10)$.

(f) Inferred state with probabilistic MAP inference:

holdsAt$(a, 3)$. holdsAt$(a, 4)$. holdsAt$(a, 5)$.
satisfied(vars(5), $rule_2$). satisfied(vars(2), $rule_1$).

(d) Syntactically transformed program $T(H_t)$ for MAP inference:

initiatedAt$(a, T) \leftarrow$ satisfied(vars(T), $rule_1$).
{satisfied(vars(T), $rule_1$)} $\leftarrow$ happensAt(b, T).
: ~ satisfied(vars(T), $rule_1$). $[-11$, vars(T), $rule_1]$
initiatedAt$(a, T) \leftarrow$ satisfied(vars(T), $rule_2$).
{satisfied(vars(T), $rule_2$)} $\leftarrow$ happensAt(c, T).
: ~ satisfied(vars(T), $rule_2$). $[-13$, vars(T), $rule_2]$
initiatedAt$(a, T) \leftarrow$ satisfied(vars(T), $rule_3$).
{satisfied(vars(T), $rule_3$)} $\leftarrow$ happensAt(d, T).
: ~ satisfied(vars(T), $rule_3$). $[2$, vars(T), $rule_3]$

Figure 5: ASP-based MAP inference with the Event Calculus.

weights of the patterns have been converted to integers, as described earlier; (c) presents the current data interpretation I_t; (d) presents the program $T(H_t)$ obtained from H_t, via the transformation in Algorithm 4, to allow for MAP inference; (e) presents the inferred state obtained with crisp logical inference, i.e. the target CE instances included in the unique answer set of the program $BK \cup H_t \cup I_t$, where the CE patterns' weights have been disregarded. Note that the occurrence of happensAt$(b, 2) \in I_t$ *initiates the target CE a via rule*$_1 \in H_t$*, so a holds at the next time point, 3, and it also holds at time points 4 & 5 via inertia. Then, the occurrence of* happensAt$(c, 5) \in I_t$ *terminates a, via rule*$_2 \in H_t$*, so a does not holds at times 6,7,8, while the occurrence of* happensAt$(d, 8) \in I_t$ *re-initiates a, via rule*$_3 \in H_t$*, so a holds at times 9 & 10. Finally, (f) presents the MAP-inferred state, i.e. the target predicate instances included in an optimal (minimum-cost) answer set of the program $BK \cup T(H_t) \cup I_t$ (for illustrative purposes the* satisfied$/2$ *instances in the optimal answer set are also presented). Note that the set of target CE inferences is reduced, as compared to the crisp case, since the negative-weight, rule*$_3 \in H_t$ *is not satisfied by the optimal answer set. The* satisfied$/2$ *instances in the MAP-inferred state correspond to the ground atoms* terminatedAt$(a, 5)$, initiatedAt$(a, 2)$*, which, along with inertia, are responsible for the target CE inferences.*

3.4.2 Weight Learning

Once the learner makes a prediction on the incoming interpretation I_t and generates the inferred state, the true state is revealed, if available (i.e., if I_t is labeled), and the CE patterns' weights are updated by comparing their true groundings in the inferred and the true state. For a target CE α and an initiatedAt$/2$ (resp. terminatedAt$/2$) CE pattern r_i, a true grounding, either in the inferred, or in the true state, is a grounding of r_i at time t, such that holdsAt$(\alpha, t+1)$ is true (resp. false). CE patterns that contribute towards correct predictions (target CE inferences) are promoted, while those that make erroneous predictions are down-weighted.

As in [80], we use the AdaGrad algorithm [49] for weight updates, a version of Gradient Descent that dynamically adapts the learning rate, i.e. the magnitude of weight promotion/demotion, for each CE pattern individually, by taking into account the pattern's performance on the past data. AdaGrad updates a weight vector, whose coordinates correspond to a set of features (the CE patterns in out case), based on the subgradient of a convex loss function of these features. Our loss function is a simple variant of the hinge loss for structured prediction, originally used in [72] for MLNs, whose subgradient is the vector with Δg_i in its i-th coordinate, the difference in the i-th CE pattern's true groundings in the true and the inferred state respectively. The weight update rule for the i-th CE pattern is then:

$$w_i^{t+1} = sign(w_i^t - \frac{\eta}{C_i^t} g_i^t)\; max\{0, |w_i^t - \frac{\eta}{C_t^i} g_i^t| - \lambda \frac{\eta}{C_t^i}\}$$

where $t/t+1$-superscripts in terms denote respectively the previous and the updated values, η is a learning rate parameter, λ is a regularization parameter and $C_i^t = \delta + \sqrt{\sum_{j=1}^{t}(g_i^j)^2}$ is a term that expresses the CE pattern's quality so far, as reflected by the accumulated sum of Δg_i's, amounting to its past mistakes (plus a $\delta \geq 0$ to avoid division by zero in η/C_i^t). The C_i^t term is the adaptive factor, since the magnitude of a weight update via the term $|w_i^t - \frac{\eta}{C_t^i} g_i^t|$ is affected by the CE pattern's previous history, in addition to its current mistakes, expressed by g_i^t. The regularization term in Equation (1), $\lambda \frac{\eta}{C_t^i}$, is the amount by which the i-th CE pattern's weight is discounted when $g_i^t = 0$. This is to eventually push to zero the weights of irrelevant rules, which have very few, or even no groundings in the data.

3.4.3 Updating CE patterns' Structure

Similarly to OLED [79], WOLED learns CE patterns via a classical in ILP, hill-climbing search process, generating a *bottom rule* [43] $\perp_\alpha$ from a CE instance α and then searching for a high-quality CE pattern into the *subsumption lattice* defined by $\perp_\alpha$. It does so by progressively specializing an initially empty-bodied rule with the addition of one literal at a time from $\perp_\alpha$. To make this process online, the data in the incoming interpretations are used once, to evaluate a CE pattern and its current specializations. A Hoeffding test [45] allows to identify, with high probability, the best specialization from a small subset of the input interpretations. Once the test succeeds, the parent rule is replaced by its best specialization and the process continues for as long as new specializations improve the current rule's performance. New bottom rules are generated over time from "missed" CE instances (not entailed by none of the existing CE patterns). Each such bottom rule instantiates a new subsumption lattice, which is searched for new CE patterns.

In particular, at each point in time WOLED evaluates a parent rule and its specializations on incoming data, via an information gain scoring function, assessing the cumulative merit of a specialization over the parent rule, across the portion of the stream seen so far:

$$G(r,r') = P_r \cdot (log\frac{P_r}{P_r+N_r} - log\frac{P_{r'}}{P_{r'}+N_{r'}})$$

where r' is r's parent rule and for each rule r, P_r (resp. N_r) denotes the sum of true (resp. false) groundings of r in the MAP-inferred states generated so far. The information gain function is normalized in $[0,1]$ by taking 0 as the minimum (as we are interested in positive gain only) and dividing a G-value by its maximum, $G_{max}(r,r') = P_{r'} \cdot (-log\frac{P_{r'}}{P_{r'}+N_{r'}})$. When the range of G is $[0,1]$, a Hoeffding test succeeds, allowing to select r_1 as the best of a parent rule r's specializations, when $G(r_1,r) - G(r_2,r) > \varepsilon = \sqrt{\frac{log1/\delta}{2N}}$, where r_1, r_2 are respectively r's best and second-best specializations, δ is a confidence parameter and N is the number of observations seen so far, we refer to [79] for further details.

A successful Hoeffding test results in replacing the parent rule r with its best specialization r_1 and moving one level down in the subsumption lattice, via generating r_1's specializations and subsequently evaluating them on new data.

Figure 6 illustrates the process for an initiation CE pattern. The rules at each level of the lattice represent the specializations of a corresponding rule at the preceding level. The greyed-out part of the search space in Figure 6 represents the portion that has already been searched, while the non greyed-out rule at the third level represents the best-so-far rule that has resulted from a sequence of Hoeffding tests.

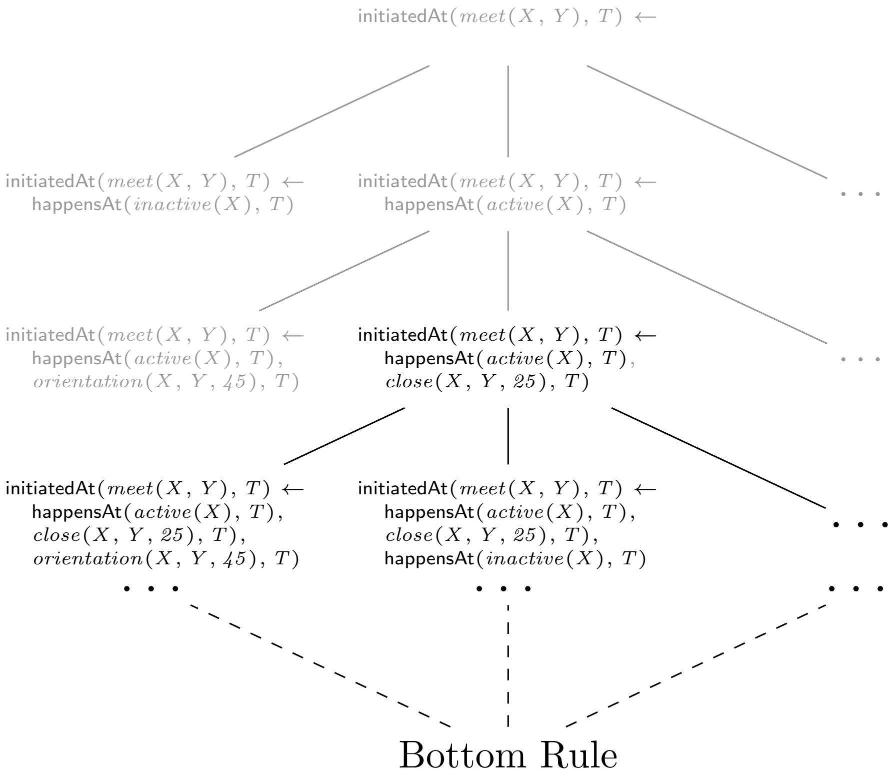

initiatedAt(meet(X, Y), T) ←
initiatedAt(meet(X, Y), T) ← happensAt(inactive(X), T)
initiatedAt(meet(X, Y), T) ← happensAt(active(X), T)
. . .
initiatedAt(meet(X, Y), T) ← happensAt(active(X), T), orientation(X, Y, 45), T)
initiatedAt(meet(X, Y), T) ← happensAt(active(X), T), close(X, Y, 25), T)
. . .
initiatedAt(meet(X, Y), T) ← happensAt(active(X), T), close(X, Y, 25), T), orientation(X, Y, 45), T)
initiatedAt(meet(X, Y), T) ← happensAt(active(X), T), close(X, Y, 25), T), happensAt(inactive(X), T)
. . .
. . .
. . .
. . .
Bottom Rule

Figure 6: A subsumption lattice.

The specializations' weights are learnt simultaneously to those of their parent rules as described in Section 3.4.2, by comparing the specializations' true groundings over time in the MAP-inferred states (generated from "top theories", consisting of parent rules only) and the true states respectively.

3.5 Learning New CE patterns

If necessary, the existing CE pattern set H_t is expanded with the addition of new CE patterns, generated in response to erroneous predictions. New initiatedAt/2 (resp. terminatedAt/2) patterns, generated from false negative (*FN*) (resp. false positive (*FP*)) mistakes, have the potential to prevent similar mistakes in the future. For instance, an *FN* mistake at time t, i.e. a target CE instance predicted as false, while actually being true at t, could have been prevented via a pattern that initiates the target CE at some time prior to t.

Generating new CE patterns from the entirety of mistakes may result in a very large number of rules, most of which are redundant. To avoid that, WOLED uses the following strategy for new CE pattern generation, presened in Algorithm 5: First, a set of bottom rules (BRs) is generated (line 4), using the constants in the erroneously predicted atoms to generate ground initiatedAt/2 and terminatedAt/2 atoms, which are placed in the head of a set of initially empty-bodied rules. The bodies of these rules are then populated with literals, grounded with constants that appear in the head, that are true in the current data interpretation I_t. The signatures of allowed body literals are specified via *mode declarations* [43].

Next, constants in the BRs are replaced by variables and the BR set is "compressed" (line 6) to a bottom theory $H_{\perp}$,

ALGORITHM 5: $\mathsf{LearnNewCEPatterns}(B, M, H_t, I_t, I_t^{\mathsf{MAP}}, I_t^{\mathsf{true}})$

```
 1: $\Pi := \emptyset$, $H_{new} := \emptyset$, $H_\perp := \emptyset$, $T(H_\perp) := \emptyset$
 2: $Mistakes := I_t^{\mathsf{true}} \setminus I_t^{\mathsf{MAP}}$.
 3: for each $m \in Mistakes$ do
 4:     $H_\perp \leftarrow \mathsf{generateBottomRule}(m, I_t, M)$
 5: end for
 6: $H_\perp \leftarrow \mathsf{compressBottomRules}(H_\perp)$
 7: for each bottom rule $r_i = \alpha_i \leftarrow \delta_i^1, \dots \delta_i^n$ in $H_\perp$ do
 8:     Add to $T(H_\perp)$ the following rules:
        $\alpha_i \leftarrow \mathsf{use}(i,0), \mathsf{try}(i,1,\mathsf{v}(\delta_i^1)), \dots, \mathsf{try}(i,n,\mathsf{v}(\delta_i^n))$.
        $\mathsf{try}(i,1,\mathsf{v}(\delta_i^1)) \leftarrow \mathsf{use}(i,1), \delta_i^1$.
        $\mathsf{try}(i,1,\mathsf{v}(\delta_i^1)) \leftarrow \mathsf{not}\ \mathsf{use}(i,1)$.
        ...
        $\mathsf{try}(i,n,\mathsf{v}(\delta_i^n)) \leftarrow \mathsf{use}(i,n), \delta_i^n$.
        $\mathsf{try}(i,n,\mathsf{v}(\delta_i^n)) \leftarrow \mathsf{not}\ \mathsf{use}(i,n)$.
 9: end for
10: $\Pi \leftarrow B \cup I_t \cup T(H_t) \cup T(H_\perp)$,
    where $T(H_t)$ is the MAP inference-related transformation
    of Algorithm 4 applied to the current CE pattern set $H_t$.
11: Add to $\Pi$ the following rules:
    $\{\mathsf{use}(I,J)\} \leftarrow \mathsf{ruleId}(I), \mathsf{literalId}(J)$.
    $:\sim \mathsf{use}(I,J).\ [1,I,J]$
12: Add to $\Pi$ one weak constraint of the form $:\sim \mathsf{not}\ \alpha.\ [1]$
    (resp. $:\sim \alpha.\ [1]$) for each target CE instance $\alpha$ included
    (resp. not included – closed world assumption) in $I_t^{\mathsf{true}}$.
13: Find an optimal answer set $\mathscr{A}_{opt}$ of $\Pi$.
14: Remove from $H_\perp$ every body literal $\delta_i^j$ for which $\mathsf{use}(i,j) \notin \mathscr{A}_{opt}$ and each rule $r_i$ for which $\mathsf{use}(i,0) \notin \mathscr{A}_{opt}$.
15: $H_{new} \leftarrow H_\perp$.
16: return $H_{new}$.
```

which consists of unique, w.r.t. θ-subsumption, variabilized BRs. The new CE patterns are chosen among those that θ-subsume $H_\perp$. To this end, the generalization technique of [124, 78], which allows to search into the space of theories that θ-subsume $H_\perp$, is combined with inference with the existing weighted CE pattern set H_t, yielding a concise set of CE patterns H_{new}, such that an optimal answer set of $B \cup H_t \cup H_{new} \cup I_t$ best-approximates the true state associated with I_t.

To this end, each BR $r_i \in H_\perp$ is "decomposed" in the way shown in line 8 of Algorithm 5, where the head of r_i corresponds to an atom $\mathsf{use}(i,0)$ and each of its body literals, δ_i^j, to a $\mathsf{try}/3$ atom, which, via the $\mathsf{try}/3$ definitions provided, may be satisfied either by satisfying δ_i^j and an additional $\mathsf{use}(i,j)$ atom, or by "assuming" not $\mathsf{use}(i,j)$. Choosing between these two options is done via ASP optimization in line 11 of Algorithm 5, where the choice rule generates $\mathsf{use}/2$ atoms that correspond to head atoms/body literals for $H_\perp$, and the subsequent weak constraint minimizes the generated instances to those necessary to approximate the true state, as encoded via the additional weak constraints in line 12. New rules are "assembled" from the bottom rules in $H_\perp$, by following the prescriptions encoded in the $\mathsf{use}/2$ atoms of an optimal answer set of the resulting program, as in line 14.

This is essentially the XHAIL algorithm [124] in an ASP context. The difference of our approach from usages of this technique in previous works [124, 78], is that here the search into the space of $H_\perp' s$ subsumers is combined with MAP

ALGORITHM 6: WOLED$(B, M, \mathscr{I})$

```
$H_t := \emptyset$.
for each interpretation $I_t \in \mathscr{I}$ do
    $I_t^{\mathsf{MAP}} := \mathsf{MAPInference}(B, H_t, I_t)$.
    Receive $I_t^{\mathsf{true}}$.
    $Mistakes := I_t^{\mathsf{true}} \setminus I_t^{\mathsf{MAP}}$.
    $H_t \leftarrow \mathsf{SpecializeCEPatterns}(H_t)$.
    $H_{\mathsf{new}} := \mathsf{LearnNewCEPatterns}(B, M, I_t, I_t^{\mathsf{MAP}}, I_t^{\mathsf{true}})$.
    $H_{\mathsf{new}} \leftarrow \mathsf{UpdateWeights}(H_t \cup H_{\mathsf{new}}, mistakes)$.
    $H_t \leftarrow H_t \cup H_{\mathsf{new}}$.
end for
```

inference with the existing set of weighted CE patterns (line 10, Algorithm 5). Therefore, new patterns are generated only insofar they indeed help to better approximate the true state. This technique allows to generalize from the data in the current interpretation via avoiding to over-fit that data, which may be potentially corrupted by noise.

Once the new CE patterns are generated, their weights (initially set to a near-zero value) are updated based on their groundings in I_t and the true state. Moreover, each new pattern r is associated with the bottom rules from $H_{\perp}$, which are θ-subsumed by r_i. These bottom rules are used as a pool of literals for further specializing r over time, as described in Section 3.4.3.

WOLED's learning strategy, mentioned at the beginning of Section 3.4, is summarized in Algorithm 6.

3.6 Experimental Evaluation

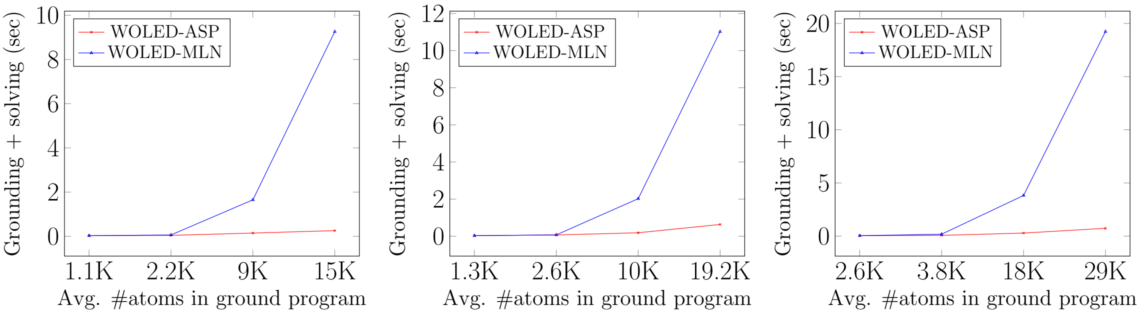
We present an experimental evaluation of our approach on two CER data sets from the domains of *activity recognition* and maritime monitoring.

3.6.1 Datasets Used

CAVIAR is a benchmark dataset for activity recognition, described in Section 3.3, consisting of 28 videos with 26,419 video frames in total. We experimented with learning CE patterns for two CEs from CAVIAR, related to two people *meeting each other* and *moving together*, which we henceforth denote by *meeting* and *moving* respectively. There are 6,272 video frames In CAVIAR where *moving* occurs and 3,722 frames where *meeting* occurs. A fragment of a CE definition for *moving* is presented in Table 5(d).

Our second dataset is a publicly available dataset from the field of maritime monitoring[6]. It consists of Automatic Identification System (AIS) position signals collected from vessels sailing in the area of Brest, France, for a period of six months, between October and March 2015. The data have been pre-processed using trajectory compression techniques [115], whereby major changes along each vessels movement are tracked. This process allows to identify critical points along each trajectory, such as a vessel stop, a turn, or a slow motion movement. Using the retained movement features, i.e. the critical points, the trajectory of a vessel may be reconstructed with small deviations from

[6] https://zenodo.org/record/1167595#.WzOOGJ99LJ9



(a) *Meeting* (b) *Moving* (c) *Rendezvous*

Figure 7: Scalability of MAP inference.

the original one. The maritime dataset has been additionally pre-processed, in order to extract spatial relations between vessels (e.g. vessels being close to each other) and areas of interest, such as protected areas, areas near coast, open-sea areas etc. There 16,152,631 critical points in the maritime dataset, involving 4,961 vessels and 6,894 areas, for a total size of approximately 1,3GB.

The maritime dataset is not labeled in terms of occurring CE instances, we therefore used hand-crafted CE patterns to perform CER on the critical points, thus generating the annotation, and the purpose of learning was to reconstruct the hand-crafted CE patterns. We experimented with learning CE patterns for a CE related to vessels involved in potentially suspicious rendezvous (henceforth denoted by *rendezVous*), which holds when two vessels are stopped, or move with very low speed in proximity to each other in the open sea.

All experiments were carried-out on a 3.6GHz processor (4 cores, 8 threads) and 16GB of RAM. The code for all algorithms used in these experiments in available online[7].

3.6.2 Scalability of Inference

The purpose of our first experiment was to assess the scalability of the ASP-based MAP inference process, which lies at WOLED's core. In this respect, we compare the ASP-based version of WOLED, which we henceforth denote by WOLED-ASP, with the version of [80], which relies on MLN libraries, and which we henceforth denote by WOLED-MLN.

Contrary to WOLED-ASP, which is based entirely on the Clingo[8] answer set solver, WOLED-MLN is based on a number of different software tools. It uses the LoMRF library for Markov Logic Networks [135], for grounding MLN theories and performing circumscription via predicate completion [137], in order to convert them into a form that supports the non-monotonic semantics of the Event Calculus for reasoning, something that WOLED-ASP has out of the box. MAP inference in WOLED-MLN is performed via a state-of-the-art in MLNs, Integer Linear Programming-based approach, which is introduced in [71] and is implemented using the lpsolve[9] solver.

To compare the MAP inference scalability of the two implementations, we used the task of online weight learning with hand-crafted CE patterns, where the learner is required to first perform MAP inference on the incoming interpretations

[7] https://github.com/nkatzz/ORL
[8] https://potassco.org/
[9] https://sourceforge.net/projects/lpsolve/

	Method	Prequential Loss	F_1-score (test set)	Theory size	Inference Time (sec)	Pred. Compl. Time (sec)	Total Time (sec)
Moving	WOLED-ASP	**1.723**	**0.821**	26	15	–	112
	WOLED-MLN	2.817	0.801	47	187	28	478
	OLED	3.755	0.730	**24**	**13**	–	74
	HandCrafted	6.342	0.637	28	–	–	–
	HandCrafted-WL	4.343	0.702	28	16	–	**52**
Meeting	WOLED-ASP	**1.212**	**0.887**	34	**12**	–	82
	WOLED-MLN	2.554	0.841	56	134	12	145
	OLED	3.224	0.782	42	10	–	36
	HandCrafted	5.734	0.735	**23**	–	–	–
	HandCrafted-WL	4.024	0.753	**23**	13	–	**31**
Rendezvous	WOLED-ASP	**0.023**	0.98	18	647	–	4,856
	WOLED-MLN	0.088	0.98	18	2,923	434	6,218
	OLED	0.092	0.98	18	**623**	–	**4,688**

Table 6: Online structure & weight learning results.

with a fixed-structure CE pattern set, and then update the CE patterns' weights based on their contribution to erroneous inferences in the MAP-inferred state. Given that the weight update cost is negligible and the CE pattern set is fixed, the MAP inference cost is the dominant one in this task, and in turn, it depends on the cost of grounding the current CE pattern set, plus the cost of solving the corresponding weighted MaxSat problem for each incoming interpretation. Note that since the CE pattern sets for each CE are fixed in this experiment, predicate completion in WOLED-MLN is performed only once at the beginning of a run, therefore its cost is negligible.

The data were consumed by the learners in mini-batches, where each mini-batch is an interpretation consisting of data in a particular time interval. We performed weight learning with different mini-batch sizes of 50, 100, 500 & 1000 time points. The size, in total number of literals in a CE pattern set, of the hand-crafted theories used in this experiment was as follows: *meeting* 23 literals, *moving* 28 literals, *rendezVous* 18 literals.

We measured the average MAP inference time (grounding plus solving time) for WOLED-ASP and WOLED-MLN respectively, throughout a single-pass over the data, for different mini-batch sizes. Note that as the mini-batch size grows, so does the size of the corresponding ground program from which the MAP-inferred state is extracted.

Figure 7 presents the results, which indicate that the growth in the size of the ground program, as the mini-batch size increases, entails an exponential growth to the MAP inference cost for WOLED-MLN. In contrast, thanks to Clingo's highly optimized grounding and solving abilities, MAP inference with WOLED-ASP takes near-constant time.

3.6.3 Online Structure & Weight Learning Performance

In our next experiment we assess WOLED-ASP's predictive performance and efficiency in the task of online structure & weight learning and we compare it to (i) WOLED-MLN; (ii) OLED [79], the crisp version of the algorithm that learns unweighted CE patterns; (iii) HandCrafted, a set of predefined rules for each CE and (iv) HandCrafted-WL, the rules in HandCrafted with weights learnt by WOLED-ASP.

To assess the predictive performance of the systems compared we used two methods: *Prequential* (predictive sequential) evaluation, or *interleaved test-then-train* [24], where each incoming data interpretation is first used to evaluate the current CE pattern set and then to update its structure and weights, and standard cross-validation. In prequential evaluation we typically measure the average prediction loss over time, which is an indication of a learner's ability to incorporate new knowledge that arrives over time into the current model. With cross-validation we assess a learner's generalization abilities, by evaluating the predictive performance of a learnt model on a test set.

The results are presented in Table 6, where the following statistics are reported for each one of the systems being compared: (i) the average prequential loss, which, for the n-th mini-batch in the learning process is defined as S/n, where S is the cumulative sum of false positive and false negative predictions up to that time. The value reported in 6 is the final value of S/n in a prequential run; (ii) F_1-score on a test set. For CAVIAR we used tenfold cross-validation and the reported F_1-scores are micro-averages obtained from ten different test sets. For the maritime and the fleet management datasets, whose size makes tenfold cross-validation impractical, we used half the dataset for training and half for testing, so the reported F_1-scores are obtained for the latter half; (iii) CE pattern set sizes (total number of literals) at the end of a prequential run (i.e., after a single-pass over a dataset); (iv) Total inference time at the end of a prequential run (MAP inference for WOLED-ASP, WOLED-MLN & HandCrafted-WL, crisp logical inference for OLED); (v) For WOLED-MLN, total time spent on predicate completion at the end of a prequential run; (vi) Total training time at the end of a prequential run, which includes time spent on CE pattern generation, computing θ-subsumption etc, i.e. the dominant costs involved in learning CE patterns structure. Note that we report on (iv), (v), (vi) only for approaches that require training (i.e., not for HandCrafted). Also, we did not experiment with hand-crafted CE patterns in the maritime and the fleet management datasets, since in these datasets hand-crafted CE patterns were used to generate the ground truth in the first place.

In addition to the different implementations of probabilistic inference, an important difference between WOLED-ASP and WOLED-MLN from an algorithmic perspective, lies in the new CE pattern generation process of Section 3.5. Thanks to its ASP-based implementation and the underlying optimization tools, WOLED-ASP is able to perform the search for new CE patterns, while taking into account the contribution of the weights of existing ones in approximating the true state of an interpretation I_t. As a result, it generates new patterns only when this does indeed result in a better approximation of the true state, given the existing weighted patterns. In contrast, WOLED-MLN, lacks this ability. It generates a bottom theory $H_\perp$ from the erroneously predicted atoms, and then attempts to gradually learn a high-quality CE pattern from the rules therein, regardless of their quality. In comparison, WOLED-ASP's strategy may lead, in principle, to simpler theories of more meaningful rules and lower online error (i.e. better prequential performance). The results in Table 6 seem to validate this claim. WOLED-ASP achieves the best prequential performance among all compared approaches. It also achieves superior cross-validation performance, as compared to WOLED-MLN (test set F_1-scores), which indicates that its new CE pattern generation strategy affects its ability to generalize. Moreover, WOLED-ASP learns simpler CE patterns sets, as shown by the theory size statistic. OLED lacks the ability for weight learning, while HandCrafted-WL does not update the CE patterns' structure, which explains their inferior prequential and cross-validation performance. The trade-off is their lower training times.

Regarding efficiency, it may be seen by comparing inference times to total training times, that the dominant cost is related to structure learning tasks (recall that total training times factor-in such costs). Yet, in comparison to WOLED-MLN, WOLED-ASP achieves significantly lower costs for MAP inference, which approximate the cost of OLED's crisp logical inference. In addition to its more sophisticated CE pattern creation strategy, which tends to generate fewer CE patterns of highly quality, this results in WOLED-ASP being significantly more efficient than WOLED-MLN. Note also, that an additional, not negligible cost for WOLED-MLN stems from the necessity of predicate completion.

4 Online Semi-supervised Learning of Event Rules

Symbolic *complex event recognition* (CER) systems [37] consume input sequences of *simple events* (SEs), match them against a knowledge base of first-order rules [20, 11], and recognise *complex events* (CEs) of interest. CEs are usually defined as multi-relational structures over actors and objects involved in an event, and thus, manual derivation of such rules can be cumbersome and error-prone. In addition, event recognition applications typically operate in noisy data streams of significant volume and velocity [62], which further renders the synthesis of such relational dependencies unrealistic. To that end, methods for learning the structure of the CEs in a single pass over the data stream are essential [138, 48, 57].

Several online relational learners have been proposed for the automated discovery of CE structures under uncertainty [79, 81, 100], stemming from Inductive Logic Programming [122] and Statistical Relational Learning [60]. Nonetheless, all of them presume that a fully labelled training sequence arrives for processing, which of course is an unrealistic assumption.

SPLICE [103] is a recent method that aids semi-supervised learning of CE rules by completing the missing labels using a graph label propagation technique [154]. Since the training instances are represented as sets of logical atoms, it employs a structural distance, adapted from [111], to compute the distance of unlabelled data to their labelled counterparts. The labelling is achieved online (single-pass), by storing previously seen labels for future usage. Although SPLICE facilitates the learning of CE rules in the presence of missing labels, its distance measure may be compromised by irrelevant features or imbalanced supervision. Moreover, its approach to online label propagation does not provide any guarantee about the labelling inferred from the local graphs built as the data stream-in.

We propose an improved hybrid distance measure that combines the structural measure of SPLICE with a mass-based dissimilarity [141], that employs mass estimation theory [140] to quantify the distance between examples. We further enhance the structural distance by performing feature selection optimised for *k*NN classification. To that end, we adapt LMNN [148], a state-of-the-art approach to metric learning. Finally, in order to provide guarantees about the online labelling, we use a technique proposed by [146] that retains a synopsis of the graph. Similar to SPLICE, the completed training data can be subsequently used by any supervised structure learner.

The proposed method (SPLICE$^+$) is compared to its predecessor (SPLICE) on the task of maritime monitoring, where the goal is to recognise vessel activities, by exploiting information such as vessel speed, location and communication gaps. Our empirical analysis suggests that our improved method outperforms its predecessor in terms of inferring the missing labels, at the price of a tolerable increase in processing time.

4.1 Background

We present existing methods that are employed in the rest of the manuscript. We begin by briefly presenting SPLICE, an approach proposed for online semi-supervised learning for complex event recognition. Then, in Section 4.1.2 we present a metric learning technique targeted to *k*NN classification and in Section 4.1.3 a data-driven dissimilarity based on probability mass estimation. Finally, in Section 4.2 we discuss temporal label propagation, a method proposed for semi-supervised learning via label propagation in data streams.

4.1.1 Online Semi-Supervised Learning for Composite Event Recognition

SPLICE [103] is a recent approach that enables online structure learning for complex event recognition in the presence of incomplete supervision. Towards that goal, it employs a graph-based technique proposed by [154], to derive labels for the unlabelled data, based on their distance to their labelled counterparts. Composite events are usually defined over multi-relational data [37], instead of simple numerical data points. Therefore, an appropriate distance function for sets of logical atoms is employed, which basically represent the incoming training examples. The labelling process is achieved in a single-pass over the data stream by storing previously seen labelled examples, and filtering out noisy ones that may compromise accuracy.

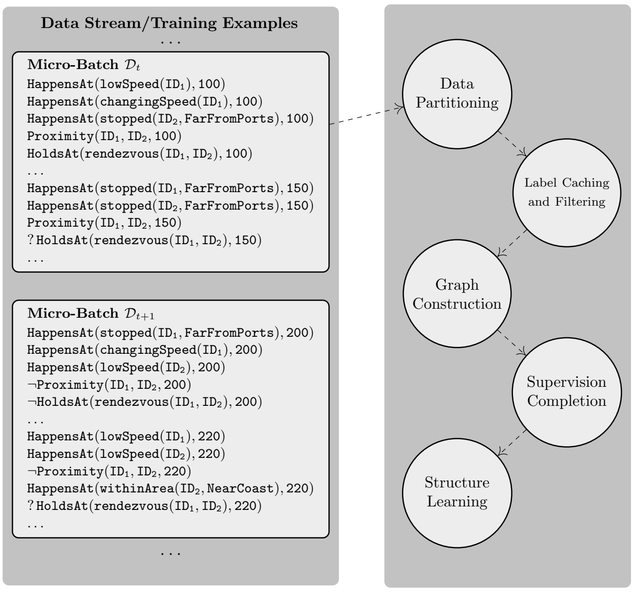

Data Stream/Training Examples
...
Micro-Batch $\mathcal{D}_t$
HappensAt(lowSpeed(ID_1), 100)
HappensAt(changingSpeed(ID_1), 100)
HappensAt(stopped(ID_2, FarFromPorts), 100)
Proximity(ID_1, ID_2, 100)
HoldsAt(rendezvous(ID_1, ID_2), 100)
...
HappensAt(stopped(ID_1, FarFromPorts), 150)
HappensAt(stopped(ID_2, FarFromPorts), 150)
Proximity(ID_1, ID_2, 150)
? HoldsAt(rendezvous(ID_1, ID_2), 150)
...
Micro-Batch $\mathcal{D}_{t+1}$
HappensAt(stopped(ID_1, FarFromPorts), 200)
HappensAt(changingSpeed(ID_1), 200)
HappensAt(lowSpeed(ID_2), 200)
¬Proximity(ID_1, ID_2, 200)
¬HoldsAt(rendezvous(ID_1, ID_2), 200)
...
HappensAt(lowSpeed(ID_1), 220)
HappensAt(lowSpeed(ID_2), 220)
¬Proximity(ID_1, ID_2, 220)
HappensAt(withinArea(ID_2, NearCoast), 220)
? HoldsAt(rendezvous(ID_1, ID_2), 220)
...
...
Data Partitioning
Label Caching and Filtering
Graph Construction
Supervision Completion
Structure Learning

Figure 8: The Semi-Supervised Online Structure Learning (SPLICE) procedure.

SPLICE procedure, as depicted in Fig. 8, is fivefold. It assumes that the training sequence arrives for processing in micro-batches. Each incoming micro-batch contains a sequence of ground evidence atoms, i.e., a set of first-order logic ground atoms. For instance, micro-batch $\mathscr{D}_t$ describes two vessels far from ports, one sailing at low speed and another one stopped, while being close to each other. The micro-batches can be fully labelled, partially labelled or contain no labels at all. For instance, at time 100, a labelled query atom arrives stating that the two vessel are performing some illegal activity. On the other hand, query atom at time 150 is unlabelled. Unlabelled query atoms are prefixed by the symbol '?'.

Each micro-batch is first passed onto the data partitioning component which groups the ground atoms into examples, as depicted in Fig. 9. Grouped examples contain exactly one ground query atom and a proper subset of the evidence atoms in the micro-batch. These evidence atoms are linked to the query through their shared constants, i.e., they are relevant to the complex event of interest. Each example is essentially a bottom rule, that is, the most specific rule that explains a single ground query atom. Therefore, every example can also be seen as a clause by replacing it's constants with variables. For instance, the top example of Fig. 9 can be transformed into the following clause:

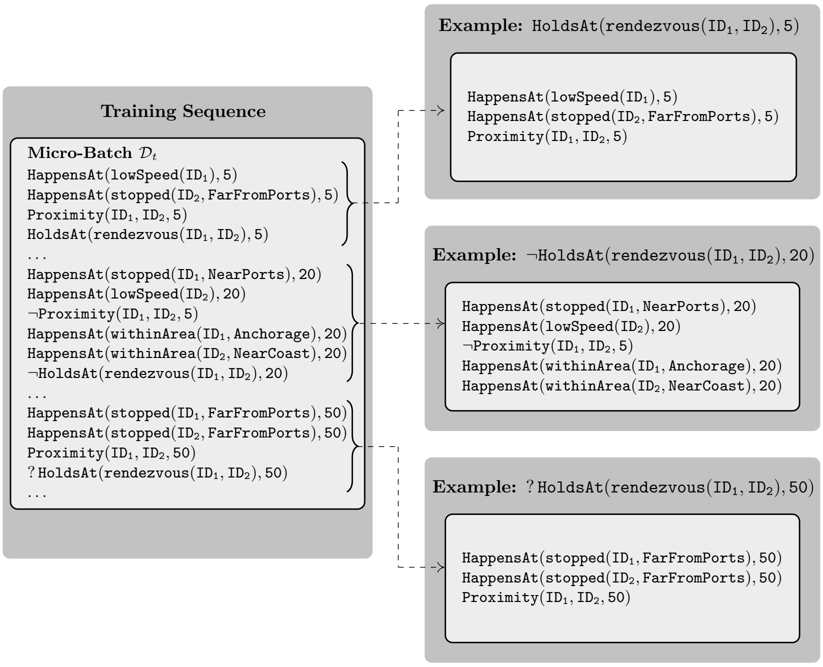

Training Sequence
Micro-Batch $\mathcal{D}_t$
HappensAt(lowSpeed(ID1), 5)
HappensAt(stopped(ID2, FarFromPorts), 5)
Proximity(ID1, ID2, 5)
HoldsAt(rendezvous(ID1, ID2), 5)
...
HappensAt(stopped(ID1, NearPorts), 20)
HappensAt(lowSpeed(ID2), 20)
¬Proximity(ID1, ID2, 5)
HappensAt(withinArea(ID1, Anchorage), 20)
HappensAt(withinArea(ID2, NearCoast), 20)
¬HoldsAt(rendezvous(ID1, ID2), 20)
...
HappensAt(stopped(ID1, FarFromPorts), 50)
HappensAt(stopped(ID2, FarFromPorts), 50)
Proximity(ID1, ID2, 50)
?HoldsAt(rendezvous(ID1, ID2), 50)
...
Example: HoldsAt(rendezvous(ID1, ID2), 5)
HappensAt(lowSpeed(ID1), 5)
HappensAt(stopped(ID2, FarFromPorts), 5)
Proximity(ID1, ID2, 5)
Example: ¬HoldsAt(rendezvous(ID1, ID2), 20)
HappensAt(stopped(ID1, NearPorts), 20)
HappensAt(lowSpeed(ID2), 20)
¬Proximity(ID1, ID2, 5)
HappensAt(withinArea(ID1, Anchorage), 20)
HappensAt(withinArea(ID2, NearCoast), 20)
Example: ?HoldsAt(rendezvous(ID1, ID2), 50)
HappensAt(stopped(ID1, FarFromPorts), 50)
HappensAt(stopped(ID2, FarFromPorts), 50)
Proximity(ID1, ID2, 50)

Figure 9: Data partitioning into examples. Each example contains a ground query atom, either labelled or unlabelled, as well as a set of true ground evidence atoms that are linked to the query atom through their constants.

$$\begin{aligned}&\texttt{HoldsAt}(\texttt{rendezvous}(x,y),t) \Leftarrow\\&\quad\texttt{HappensAt}(\texttt{lowSpeed}(x),t) \wedge \texttt{HappensAt}(\texttt{stopped}(y,\texttt{FarFromPorts}),t) \wedge \texttt{Proximity}(x,y,t)\end{aligned}$$

In order to address the online processing requirement and the fact that labels are infrequent, SPLICE caches previously seen labelled examples for future usage. The caching mechanism only stores unique examples, along their frequency (the number of times they have appeared in the stream) in order to favour compression. To that end, SPLICE also stores the clausal form of each example and employs logical unification in order to check for uniqueness. The stored labelled examples along the unlabelled examples of the current micro-batch compose the example vertices of the graph used in the subsequent steps.

Once the example vertices have been collected, they are connected by edges representing the structural similarity of their underlying evidence atoms. The structural similarity is adapted from [111] by replacing the Haussdorf metric by the Kuhn–Munkres algorithm [87]. More formally, let a pair of vertices $v_i = \{e_{i1} \dots, e_{iM}\}$ and $v_j = \{e_{j1}, \dots, e_{jK}\}$ consisting of M and K evidence atoms respectively. SPLICE first computes the structural distance between each pair of evidence atom $d(e_{im}, e_{jk})$ where $m \in \{1, \dots, M\}$ and $k \in \{1, \dots, K\}$ resulting in a $M \times K$ distance matrix $\mathbf{D}$, where $M > K$.

The matrix $\mathbf{D}$ is then given as an input cost matrix to the Kuhn-Munkres algorithm, in order to find the optimal mapping of evidence atoms. The optimal mapping is denoted here by the function $g : V \times V \mapsto \{(m, k) : m, k \in \{1, \dots, K\}\}$ and is the one that minimises the total cost, i.e., the sum of the distances of the mappings. Finally, SPLICE computes the total distance between the vertices v_i, v_j as the sum of the distances yielded by the optimal mapping normalised by the greater dimension, that is M, of the matrix:

$$d(v_i,v_j) = \frac{1}{M}\left[(M-K) + \sum_{(m,k)\in g(v_i,v_j)} \mathbf{D}_{mk}\right] \tag{12}$$

where $M-K$ penalises all unmatched evidence atoms by the greatest possible distance, that is 1. Thus, $M-K$ can be seen as a regularisation term. The distance is turned into a similarity as $s(v_i,v_j) = 1-d(v_i,v_j)$ and yields a similarity matrix $\mathbf{W}$. Subsequently, a sparsification method is used on the fully connected graph in order to retain only edges between very similar vertices. To do so, SPLICE employs a classic k-nearest neighbour approach. In order to avoid tie-breaking, kNN selects the top k distinct weights in a vertex neighbourhood, and then connect all neighbours having such a weight. The resulting graph is used to derive labels for all unlabelled examples in the micro-batch by obtaining the harmonic solution and apply thresholding at zero to produce binary labels.

As a final step, the labelled micro-batch is passed down to a structure learner, such as [79, 100, 81], in order to induce CE rules or enhance existing ones that capture the complex events of interest. The process is repeated as long as micro-batches continue to stream-in.

Although SPLICE aids the automated discovery of CE rules in the presence of incomplete supervision, its online procedure and distance function are far from perfect. Specifically, its performance is compromised in the presence of irrelevant features because the distance function is agnostic about the feature information. Moreover, it does not provide any guarantee about the harmonic solution computed per micro-batch with respect to the solution on the entire graph. In fact, as the micro-batch size gets smaller the harmonic solution tends to be more independent from the unlabelled examples. It is interesting to note that, in the case of true streaming (one example per micro-batch), the graph is actually the k-nearest neighbour function and thus the optimisation reduces to k-nearest neighbour classification ([28], Section 11.6).

4.1.2 Large-Margin Nearest Neighbour

Graph-based methods to semi-supervised learning rely on the idea of the *cluster assumption*, that is, similar examples should yield the same labelling. Thus, the distance function constitutes an essential component of these methods and in fact controls the quality of the solution. A common issue of distance measures is that they are agnostic about the input feature information. As a consequence, their measurements may suffer in the presence of irrelevant or noisy features.

Large margin nearest neighbour (LMNN) [148] is a state-of-the-art metric learning technique that learns a distance pseudo-metric targeted to kNN classification. Intuitively, LMNN attempts to increase the number of training examples whom k-nearest neighbours share the same label. To that end, it learns a linear transformation of the input feature space using the Euclidean distance. Euclidean distances can be parametrised by a matrix $\mathbf{L}$ that applies a linear transformation to each data vector $\mathbf{x}$ as follows:

$$d_{\mathbf{L}}(\mathbf{x}_i,\mathbf{x}_j) = ||\mathbf{L}(\mathbf{x}_i-\mathbf{x}_j)||_2^2 \tag{13}$$

Euclidean distances in the transformed space can equivalently be viewed as Mahalanobis distances in the original space in terms of a square matrix $\mathbf{M} = \mathbf{L}^\top\mathbf{L}$ as follows:

$$d_{\mathbf{M}}(\mathbf{x}_i,\mathbf{x}_j) = (\mathbf{x}_i-\mathbf{x}_j)^\top\mathbf{M}(\mathbf{x}_i-\mathbf{x}_j) \tag{14}$$

where the Euclidean distance can be recovered by setting **M** equal to the identity matrix **I**.

In order to optimise *k*NN classification, one seeks a linear transformation such that nearest neighbours computed from the distance in eq. (13) share the same labels. Towards that goal, LMNN minimises a loss function consisting of two terms, one which pulls target neighbours closer together, and another which pushes differently labelled examples apart.

The first term penalises large distances between nearby instances, e.g., nearest neighbours, sharing the same label. In terms of the linear transformation **L** of the input space, the sum of these squared distances is given by:

$$\varepsilon_{pull}(\mathbf{L}) = \sum_{j \in \mathcal{N}_i^k} ||\mathbf{L}(\mathbf{x}_i - \mathbf{x}_j)||^2 \tag{15}$$

where $\mathcal{N}_i^k$ denotes the set of *target* k-nearest neighbours of the instance $\mathbf{x}_i$. The target neighbours of $\mathbf{x}_i$ are those instances that we desire to be the closest to $\mathbf{x}_i$.

The second term penalises small distances between differently labelled examples, called impostors. More formally, for an example $\mathbf{x}_i$ with label y_i and target neighbour $\mathbf{x}_j$, an impostor is any example $\mathbf{x}_l$ with label $y_l \neq y_i$ such that:

$$||\mathbf{L}(\mathbf{x}_i - \mathbf{x}_l)||^2 \leq ||\mathbf{L}(\mathbf{x}_i - \mathbf{x}_j)||^2 + 1 \tag{16}$$

In other words, an impostor $\mathbf{x}_l$ is any differently labelled example that invades the perimeter plus unit margin defined by any target neighbour $\mathbf{x}_j$ of the example $\mathbf{x}_i$. Therefore, the second term penalizes violations of the above inequality as follows:

$$\varepsilon_{push}(\mathbf{L}) = \sum_{i,j \in \mathcal{N}_i^k} \sum_l (1 - y_{il}) \left[1 + ||\mathbf{L}(\mathbf{x}_i - \mathbf{x}_j)||^2 - ||\mathbf{L}(\mathbf{x}_i - \mathbf{x}_l)||^2\right]_+ \tag{17}$$

where the indicator variable $y_{il} = 1$ if and only if $y_i = y_l$, and $y_{il} = 0$ otherwise. Moreover, $[z]_+ = \max(z, 0)$ denotes the standard hinge loss which monitors the inequality of eq. (16). If the inequality does not hold (i.e., the input $\mathbf{x}_l$ lies a safe distance away from $\mathbf{x}_i$), then its hinge loss has a negative argument and makes no contribution to the overall loss. The combined loss derived from eq. (15) and eq. (17) is as follows:

$$\varepsilon(\mathbf{L}) = (1 - \mu)\, \varepsilon_{pull}(\mathbf{L}) + \mu\, \varepsilon_{push}(\mathbf{L}) \tag{18}$$

were the weighting parameter $\mu \in [0, 1]$ balances the two goals.

Since the loss function is not convex, LMNN reformulates the optimisation problem as an instance of semidefinite programming (SDP) by substituting eq. (14) into eq. (18) to obtain:

$$\varepsilon(\mathbf{M}) = (1 - \mu) \sum_{j \in \mathcal{N}_i^k} d_{\mathbf{M}}(\mathbf{x}_i, \mathbf{x}_j) + \mu \sum_{i,j \in \mathcal{N}_i^k} \sum_l (1 - y_{il}) \left[1 + d_{\mathbf{M}}(\mathbf{x}_i, \mathbf{x}_j) - d_{\mathbf{M}}(\mathbf{x}_i, \mathbf{x}_l)\right]_+ \tag{19}$$

The resulting loss function is expressed over positive semidefinite matrices $\mathbf{M} \succeq 0$, as opposed to real-valued matrices $\mathbf{L}$. Therefore, an SDP is obtained by introducing non-negative slack variables ξ_{ijl} to mimic the effect of the hinge loss. In particular, each slack variable $\xi_{ijl} \geq 0$ is used to measure the amount by which the large margin inequality in eq. (16) is violated. The final SDP is formulated as follows:

$$\begin{aligned}
\textbf{minimise} \quad & (1-\mu) \sum_{j \in \mathcal{N}_i^k} (\mathbf{x}_i - \mathbf{x}_j)^\top \mathbf{M} (\mathbf{x}_i - \mathbf{x}_j) + \mu \sum_{i,l,j \in \mathcal{N}_i^k} (1 - y_{il}) \xi_{ijl} \\
\textbf{subject to} \quad & \textbf{(1)}\ (\mathbf{x}_i - \mathbf{x}_l)^\top \mathbf{M} (\mathbf{x}_i - \mathbf{x}_l) - (\mathbf{x}_i - \mathbf{x}_j)^\top \mathbf{M} (\mathbf{x}_i - \mathbf{x}_j) \geq 1 - \xi_{ijl} \\
& \textbf{(2)}\ \xi_{ijl} \geq 0 \\
& \textbf{(3)}\ \mathbf{M} \succeq 0
\end{aligned}$$

Large margin feature weighting (LMFW) [29] is a derivation of the LMNN technique that aims to learn a weighting feature vector $\mathbf{m}$, instead of a distance, by assuming that $\mathbf{M}$ is a diagonal matrix with $\mathbf{M}_{pp} = m_p \geq 0$, and m_p is the weighting factor of the pth feature. Thus, the loss function depicted in eq. (19) becomes:

$$\varepsilon(\mathbf{m}) = (1-\mu) \sum_{j \in n_i^k} d_{\mathbf{m}}(\mathbf{x}_i, \mathbf{x}_j) + \mu \sum_{i,j \in n_i^k} \sum_{l} (1 - y_{il}) \left[1 + d_{\mathbf{m}}(\mathbf{x}_i, \mathbf{x}_j) - d_{\mathbf{m}}(\mathbf{x}_i, \mathbf{x}_l)\right]_+ \tag{20}$$

The minimisation of the simplified objective function presented above can be represented as a linear optimisation problem with linear constraints:

$$\begin{aligned}
\textbf{minimise} \quad & (1-\mu) \sum_{j \in \mathcal{N}_i^k} ||\mathbf{m}(\mathbf{x}_i - \mathbf{x}_j)||^2 + \mu \sum_{i,l,j \in \mathcal{N}_i^k} (1 - y_{il}) \xi_{ijl} \\
\textbf{subject to} \quad & \textbf{(1)}\ ||\mathbf{m}(\mathbf{x}_i - \mathbf{x}_l)||^2 - ||\mathbf{m}(\mathbf{x}_i - \mathbf{x}_j)||^2 \geq 1 - \xi_{ijl} \\
& \textbf{(2)}\ \xi_{ijl} \geq 0 \\
& \textbf{(3)}\ \mathbf{m} \geq 0
\end{aligned} \tag{21}$$

4.1.3 Mass-based Dissimilarity

Supervised learning approaches to feature selection require explicit or implicit computation of the information/importance per feature using the labels available in the training examples. However, in a semi-supervised learning task, that information may be inaccurate due the limited labels. Therefore, such criteria are not always reliable and their optimality guarantees suffer from the fact that only very few training examples are used during the optimisation.

[141] recently proposed a mass-based dissimilarity, that employs estimates of the probability mass to quantify the dissimilarity of two points rather than the classic geometric model. Geometric approaches, such as the Euclidean distance, depend on the geometric positions of data points alone to derive a measurement. Instead mass dissimilarity measures mainly depend on the distribution of the data. The intuition is that the dissimilarity of two points primarily depends on the amount of probability mass in the region of space covering the two points. Thus, two points in a dense region are less similar to each other than two points of the same interpoint distance in a sparse region.

More formally, let H denote a hierarchical partitioning of a space $\mathbb{R}^q$ into a set of non-overlapping regions that collectively span $\mathbb{R}^q$. Moreover, each region in the hierarchy corresponds to the union of its child regions. Let $\mathscr{H}(D)$ denote the set of all such hierarchical partitions H that are admissible under a data set D such that each non-overlapping region contains at least one point from D. Then the smallest region covering a pair of points $\mathbf{x}, \mathbf{y} \in \mathbb{R}^q$ with respect to a hierarchical partitioning model H of $\mathbb{R}^q$ is defined as:

$$R(\mathbf{x}, \mathbf{y}|H) = \underset{r \in H \ s.t. \{\mathbf{x}, \mathbf{y}\} \in r}{\operatorname{argmax}} \ depth(r; H)$$

where $depth(r;H)$ is the depth of region r in the hierarchical model H.

Suppose that a dataset D is sampled from an unknown probability density function F. Then, the mass-based dissimilarity of $\mathbf{x}$ and $\mathbf{y}$ w.r.t. D is defined as the expectation of the probability that a randomly chosen point would lie in the region $R(\mathbf{x}, \mathbf{y}|H)$:

$$m(\mathbf{x}, \mathbf{y}|D) = E_{\mathscr{H}(D)} \left[P_F(R(\mathbf{x}, \mathbf{y}|H_i; D)) \right]$$

where the expectation is computed over all possible partitioning $\mathscr{H}(D)$ of the data. In practice, however, mass-based dissimilarity can be estimated from a finite number of partitioning models $H_i \in \mathscr{H}(D)$, $i = 1, \ldots, T$ as follows:

$$\tilde{m}(\mathbf{x}, \mathbf{y}|D) = \frac{1}{T} \sum_{i=1}^{T} \tilde{P}(R(\mathbf{x}, \mathbf{y}|H_i; D)) \tag{22}$$

where $\tilde{P}(R) = \frac{1}{|D|} \sum_{\mathbf{z} \in D} \mathbb{1}(\mathbf{z} \in R)$ estimates the probability of the region R by counting the data points in that region; and $\mathbb{1}(\cdot)$ denotes an indicator function. Thus, the probability of the data falling into the smallest region containing both $\mathbf{x}$ and $\mathbf{y}$, is analogous to the shortest distance between them measured in the geometric model.

In order to generate partitioning models H, a recursive partitioning scheme is employed based on *Isolation Forest* [96]. Isolation Forest is essentially an ensemble of random trees, called *Isolation Trees*. Each Isolation Tree is built independently using a subset of the data. At each internal node of the tree a random split is made to partition the data at that node into two non-empty subsets. The process is repeated recursively until either every data point is isolated or a given maximum tree height is reached.

Subsequently the resulting Isolation Forest can be used to compute the mass-based dissimilarity of eq.(22). Since each Isolation Tree essentially represents a partitioning H_i, the mass-based dissimilarity can be defined as:

$$\tilde{m}(\mathbf{x}, \mathbf{y}) = \frac{1}{T} \sum_{i=1}^{T} \frac{|R(\mathbf{x}, \mathbf{y}|H_i)|}{|D|} \tag{23}$$

where $\frac{|R(\mathbf{x}, \mathbf{y}|H_i)|}{|D|}$ estimates the probability of region R, as denoted by $\tilde{P}(R)$ in eq. (22). To compute eq. (23), $\mathbf{x}$ and $\mathbf{y}$ are passed through each Isolation Tree to find the mass of the deepest node containing both $\mathbf{x}$ and $\mathbf{y}$ i.e., $\sum_i |R(\mathbf{x}, \mathbf{y}|H_i)|$. Finally, $\tilde{m}$ is the mean of these masses over the T trees.

4.2 Temporal Label Propagation

Traditionally, graph-based methods to semi-supervised learning [155] assume that all labelled and unlabelled data are stored in memory and thus are available during the optimisation (label propagation) that yields the harmonic solution. However, that is an unrealistic assumption in online processing of data streams. SPLICE, as presented in Section 4.1.1, relaxes that assumption by storing previously seen labelled examples and reusing them in subsequent micro-bathes. Nevertheless, it still ignores previously seen unlabelled examples and by extension their respective distances to the rest of the graph. Therefore, SPLICE cannot guarantee that the global harmonic solution obtained by label propagation on the entire graph is preserved on the local graph of the micro-batch.

TLP [146] is a recently proposed approach to label propagation for fast-moving data streams. TLP stores a synopsis of the full history of the stream in order to retain the previous knowledge for both labelled and unlabelled examples and incorporate it into the subsequent optimisations. To that end, TLP draws inspiration from the connection of label propagation to the theory of electric networks [154] and, in particular, the idea of the short-circuit operator. The latter allows for a graph $\mathscr{G}$ to be encoded into a smaller (re-weighted) graph using only a subset V_τ of the actual vertices V, called *terminals*. The reduced graph $\mathscr{G}\langle V_\tau \rangle$ is called short-circuit graph and it is known to retain the global properties of $\mathscr{G}$; most importantly, it preserves the effective weights between every pair of terminal vertices. [146] proved that the aforementioned property allows for the harmonic solution to be preserved in the synopsis graph.

The Laplacian matrix of $\mathscr{G}\langle V_\tau \rangle$, required to obtain the harmonic solution, is given by the *Schur Complement* [46]. Since computing the Shur Complement is as expensive as computing the harmonic solution on the entire graph $\mathscr{G}$, it provides no substantial speed-up for the offline label propagation. However, TLP operates in a online fashion and computes $\mathscr{G}\langle V_\tau \rangle$ as a sequence of local operations, called *star-mesh transforms*. The latter is a direct consequence of the sequential property of Schur complement ([150], Theorem 4.10; [46], Lemma III.1).

Definition 3. *Star-mesh transform on a vertex v_o in a graph $\mathscr{G} = (V, E, \mathbf{W})$ is as follows:*

1. *Star: Remove v_o from $\mathscr{G}$ along its incident edges.*

2. *Mesh: For every pair of vertices $v, v' \in V$ such that $(v, v_o) \in E$ and $(v', v_o) \in E$, add the edge (v, v') to E with weight $w_{v_o,v} w_{v_o,v'} / degree(v_o)$. If (v, v') is already in E, then add the new weight to its current weight.*

The intuition is to apply star-mesh transforms as the data arrive for processing in order to continuously update the in-memory graph synopsis and deliver labels for the incoming unlabelled examples by computing the harmonic solution on the compressed graph. The star-mesh transforms remove edges by meshing their weights with the remaining graph, so that the information provided by the removed vertex v_o remains encoded. Thus the synopsis retains the ability to compute the harmonic solution for the rest of the vertices as if v_o was still in the graph ([146], Theorem 4.1).

More formally, consider a (possibly infinite) data stream $\{v_t\}_{t=1}^{\infty}$ of incoming example vertices that can be either labelled or unlabelled. TLP maintains a graph $\mathscr{G}\langle V_\tau \rangle$ that stores the τ more recent unlabelled examples, in addition to a pair of labelled node clusters containing all the labelled examples seen so far. When a new unlabelled example arrives, TLP appends it to $\mathscr{G}\langle V_\tau \rangle$ and evicts the oldest unlabelled example by applying the star-mesh transform of Definition 3. In the simplest case where a labelled example arrives it just appends it to the appropriate cluster node, thus always maintaining $\tau + 2$ nodes. The harmonic solution for each new unlabelled example is then computed on $\mathscr{G}\langle V_\tau \rangle$ and it is provably equal to the one computed on the entire stream seen so far.

4.3 Robust Supervision Completion

SPLICE, as presented in Section 4.1.1, aims to effectively learn the structure of composite event rules in the presence of incomplete supervision. However, there are a couple of downsides, related to graph construction (see Fig. 8), that may compromise the online labelling of the unlabelled data. First, the underlying structural distance may be deluded in the presence of irrelevant or noisy features. Second, the distance measurements between labelled and unlabelled data, inevitably, are as informative as the provided labels. If the given labels are not representative of the underlying class distribution, so are the measurements. Third, the online labelling inferred from the local graphs per micro-batch, provides no guarantee with respect to the global solution obtained if all data where to be accessed at once.

Our goal is to improve the quality of the graph construction component leading to a more robust and accurate labelling of the incoming unlabelled data. Towards that goal, we propose a hybrid distance measure composed of two elementary distances, that combined aim to eliminate the drawbacks of the structural distance alone. The former distance is an enhanced version of eq. (12) that accounts for irrelevant or noisy features by selecting only a subset of them, that is, the ones optimising kNN classification in labelled data. Since such a feature selection is achieved using only the labelled data, the selected features may not always be representative of the underlying classes. Therefore, we combine the optimised structural distance with a data-driven mass-based dissimilarity, adapted for logical structures. The latter samples the space of logical structures, and employs mass estimation theory to compute the relative distance between examples, which is measured as the probability density of their least general generalisation [120].

In order to render SPLICE aware of the temporal aspect of the data, that is native to online processing and CER, we further alter its strategy for interconnecting graph vertices. We connect each unlabelled vertex to its k-nearest labelled neighbours, as well as, the temporally previous unlabelled vertex. This way we promote temporal interactions between temporally adjacent unlabelled vertices during label propagation. Finally, we store a synopsis of the full history of the stream, by means of a short-circuit operator, that preserves the effective distances of labelled and unlabelled example vertices to subsequent optimisations.

Henceforth, we refer to our enhanced approach as SPLICE^{+}. The proposed improvements introduced by our method are detailed in the following subsections. To aid the presentation, we employ examples from maritime monitoring.

4.3.1 Large Margin Feature Selection for Logical Structures

In order to render the structural distance of eq. (12) aware of irrelevant or noisy features, we introduce a mechanism for feature selection based on the ideas of LMNN metric learning. We adapt the idea of feature weighting, as presented in Section 4.1.2, by learning a binary vector, instead of real-value one, that represents the set of selected logical atoms that should be used for computing distances. More formally, let $\mathscr{A}$ be a set of first-order atoms that can be constructed from a Hebrand base $\mathscr{B}$ and a set of mode declarations $\mathscr{M}$, by replacing constants with variables. Assuming a strict ordering of atoms in $\mathscr{A}$, let $\mathbf{b}$ be a vector of binary variables, one of each first-order atom $a_i \in \mathscr{A}$. Thus, each indicator variable $b_i = 1$ if the ith atom is selected, and $b_i = 0$ otherwise. Since each labelled training example is essentially a clause c, it can also be seen as a binary vector $\mathbf{x}_c = [x_1, \dots, x_{|\mathscr{A}|}]^\top$, where each variable x_i refers to the presence of the corresponding atom a_i from $\mathscr{A}$ in the clause represented by $\mathbf{x}_c$. For instance, assuming that $\mathscr{B}$ contains the ground atoms appearing in Fig. 9, we can create an ordered set of atoms as follows:

$$\begin{aligned}\mathscr{A} = \big\{&\texttt{HappensAt}(\texttt{lowSpeed}(x),t), \texttt{HappensAt}(\texttt{lowSpeed}(y),t), \texttt{Proximity}(x,y,t),\\ &\texttt{HappensAt}(\texttt{stopped}(x,\texttt{FarFromPorts}),t), \texttt{HappensAt}(\texttt{stopped}(y,\texttt{FarFromPorts}),t),\\ &\texttt{HappensAt}(\texttt{withinArea}(x,\texttt{NearCoast}),t), \texttt{HappensAt}(\texttt{withinArea}(y,\texttt{NearCoast}),t)\big\}\end{aligned}$$

Then, the top example from Fig. 9 is represented as $\mathbf{x}_{top} = [1,0,1,0,1,0,0]$, then middle as $\mathbf{x}_{mid} = [0,1,0,1,0,1,1]$ and the bottom as $\mathbf{x}_{bot} = [0,0,1,1,1,0,0]$. Thus, the distance between two such examples is essentially a Hamming distance, which is equivalent to the general Minkowski distance for $p = 1$. Since the Minkowski distance is a generalisation of the Euclidean distance, we reformulate the loss function of eq. (20) as follows:

$$\varepsilon(\mathbf{b}) = (1-\mu)\sum_{j\in\mathscr{N}_i^k}\mathbf{b}|\mathbf{x}_i-\mathbf{x}_j| + \mu\sum_{i,j\in\mathscr{N}_i^k}\sum_l(1-y_{il})\big[1+\mathbf{b}|\mathbf{x}_i-\mathbf{x}_j|-\mathbf{b}|\mathbf{x}_i-\mathbf{x}_l|\big]_+$$

where $\mathbf{x}$ is the clausal form of an example represented as a binary vector according to a predetermined strict ordering over $\mathscr{A}$. The resulting minimisation problem of the above loss function is an integer programming problem and can be solved by any linear programming solver, however less efficiently than the real value one.

$$\begin{aligned}\textbf{minimize}\quad & (1-\mu)\sum_{j\in\mathscr{N}_i^k}\mathbf{b}|\mathbf{x}_i-\mathbf{x}_j| + \mu\sum_{i,l,j\in\mathscr{N}_i^k}(1-y_{il})\xi_{ijl}\\ \textbf{subject to}\quad & \textbf{(1) } \mathbf{b}|\mathbf{x}_i-\mathbf{x}_l| - \mathbf{b}|\mathbf{x}_i-\mathbf{x}_j| \geq 1-\xi_{ijl}\\ & \textbf{(2) } \mathbf{b}\mathbf{x}_i \geq 1\\ & \textbf{(3) } \xi_{ijl}\in\mathbb{N}^{\geq}\\ & \textbf{(4) } \mathbf{b}\in\{0,1\}^{|\mathscr{A}|}\end{aligned}\tag{24}$$

Similarly to LMFW (see Section 4.1.2), the intuition of our proposed feature selection (LMFS) is to keep the minimal set of features that are necessary to have optimal distances according to the given set of labelled examples. Note that the slack variables that monitor the hinge loss are integers instead of real values since a hamming distance yields only integer differences. Moreover, we have added an extra constraint that forces all labelled examples to have at least one positive atom that is selected. This constraint is necessary to avoid extremely sparse solutions that remove many atoms yielding empty examples. As soon as the optimal vector **b** has been found, we can generalise all example vertices by removing irrelevant features, that is, features for which $b_i = 0$. Then, the structural distance can be computed as usual using eq. (12) over the generalised vertices:

$$d_s^{\mathbf{b}}(v_i,v_j) = d_s(v_i^{\mathbf{b}},v_j^{\mathbf{b}})\tag{25}$$

where v_i, v_j are vertices and $v_i^{\mathbf{b}}, v_j^{\mathbf{b}}$ have been generalised by removing the corresponding first-order atoms. In case an atom does not appear in the labelled examples we assume it is selected ($b = 1$) and use it for measuring distances.

LMNN requires that training examples are accompanied by some form of labelling. Then the optimisation above would retain the features that are necessary to discriminate between these labels. However, in a Hamming space distances change quite abruptly because a single mismatch between two binary vectors always yields a penalty of 1 between the vectors. Therefore, while in an Euclidean space two points can be close or far in a specific dimension, according to their real-value difference, in a Hamming space either they are close or they are far. Moreover, clauses formed from training examples can be very different inside the boundaries of a specific class, leading to very sparse solutions since the optimisation would try to make them similar by removing atoms that mismatch between them. To avoid such situations, we perform clustering into the examples of each class and use the clusters as distinct classes to solve the optimisation problem.

Since we are interested in grouping together the examples of each class in such a way that clusters contain patterns of similar structure, we employ a hierarchical clustering approach based on θ-subsumption. Specifically for each set of labelled examples, we select the one having the maximum number of atoms and create a unit cluster. Then for each remaining example we add it in an existing cluster, if it θ-subsumes at least one example from it, or create a new unit cluster otherwise.

ALGORITHM 7: LMFS($\mathscr{V}_L$, $\mathscr{M}$)

```
Input: $\mathscr{V}_L$: a set of labelled example vertices, $\mathscr{M}$: a set of mode declarations
Output: b: a vector of binary values corresponding to selected features
1 Partition $\mathscr{V}_L$ into positive $\mathscr{V}_P$ and negative $\mathscr{V}_N$ vertices;
2 Find $v_i^{\max} = \text{argmax}_{v_i \in \mathscr{V}_P} |v_i|$ and $v_j^{\max} = \text{argmax}_{v_j \in \mathscr{V}_N} |v_j|$;
3 Form unit clusters $C = \{\{v_i^{\max}\}, \{v_j^{\max}\}\}$;
4 foreach $v_i \in \mathscr{V}_{L/v_i^{\max}, v_j^{\max}}$ do
5     foreach $c \in C$ do
6         if $\exists\, v' \in c : clause(v_i)\theta \subseteq clause(v')$ then
7             $c = c \bigcup v_i$
8 Solve optimisation of eq. (24) using $C$ as a set of examples;
9 return b;
```

Algorithm 7 presents the pseudo-code for selecting the first-order atoms that best discriminate the known labelled vertices. The algorithm requires as an input a set of labelled example vertices, a set of mode declarations, and produces a vector of selected features. At line 1 we partition the given example vertices into positive and negative examples. Then for each of these groups we find the vertex having the maximum number of evidence atoms (we break ties randomly) and create unit clusters (lines 2–3). For each of the remaining vertices we either append it to an existing cluster, if another vertex exists that is θ-subsumed by the candidate, or create a new unit cluster (lines 4–7). Finally, we solve the optimisation of eq. (24) using the clusters as distinct classes and return a the vector of selected features (lines 8–9).

4.3.2 Mass Dissimilarity for Logical Structures

Supervised learning approaches to feature selection require explicit or implicit computation of the information/importance per feature using the labels available in the training examples. However, in a semi-supervised learning task, that information may be inaccurate due the limited labels. Therefore, such criteria are not always reliable and their optimality guarantees suffer from the fact that only very few training examples are used during the optimisation.

In order to address these issues, we combine the optimised distance, as presented in Section 4.3.1, to a data-driven dissimilarity that uses mass estimation theory to measure the distance between data points. The intuition of the measure is that two points are considered to be more similar if they coexist in a sparse space rather than a dense one. Moreover, since it is data-driven it exploits both labelled and unlabelled data to quantify the distances between examples of interest.

To that end, we adapt the approach presented in Section 4.1.3 to logical structures by means of the Herbrand base $\mathscr{B}$ and a set of mode declarations $\mathscr{M}$ that combined span a set of logical atoms $\mathscr{A}$. Since the space of logical atoms is a hypercube $\{0,1\}^{|\mathscr{A}|}$, we can define a hierarchical partitioning H of the hypercube by randomly constructing Isolation Tree. In contrast to the approach presented by [141], which assumes real value features, we can construct the trees beforehand since each internal node of the tree can only have one possible split.

ALGORITHM 8: CreateForest($\mathscr{A}$, T, h)

```
Input: $\mathscr{A}$: a set of first-order atoms, $T$: number of trees, $h$: maximum tree height
Output: $\mathscr{F}$: an isolation forest (set of isolation trees)
1  $\mathscr{F} = \emptyset$;
2  foreach $i = 1$ to $T$ do
3      $\mathscr{F} = \mathscr{F} \bigcup$ CreateTree($\mathscr{A}, 0, h$)
4  return $\mathscr{F}$;
5  CreateTree($\mathscr{A}$, $d$, $h$)
6  if $d > h \vee |\mathscr{A}| < 1$ then
7      return Node(size $\leftarrow$ 0, split $\leftarrow \emptyset$, left $\leftarrow \emptyset$, right $\leftarrow \emptyset$);
8  else
9      Randomly select an atom $a \in \mathscr{A}$;
10     return Node(size $\leftarrow$ 0, split $\leftarrow a$, left $\leftarrow$ CreateTree($\mathscr{A}/a, d+1, h-1$),
11                  right $\leftarrow$ CreateTree($\mathscr{A}/a, d+1, h-1$));
```

Algorithm 8 presents the pseudo-code for creating a forest of binary isolation trees. The algorithm requires as an input a set of first-order atoms, a number of trees, and a maximum height for each tree. We start from an empty set and iteratively we generate random trees (see lines 1–4). Each tree is built recursively by picking a random atom from the given set of available atoms and creating two random subtrees on the remaining atoms (lines 9–11). The process terminates if no more atoms are left in the set $\mathscr{A}$ or the maximum height is achieved.

Note that during tree creation, each internal node has zero size. The internal node's size of each tree updates its counts as more data stream-in. More specifically, for each incoming example we pass its clausal representation through each constructed Isolation Tree and match the internal nodes. The path from the root the the leaf that contains that matches the atoms of the given example increment their counts.

ALGORITHM 9: UpdateForest($\mathscr{F}$, V)

```
Input: $\mathscr{F}$: a set of isolation trees, $V$: a set of example vertices
foreach $tree \in \mathscr{F} \wedge v \in V$ do
    UpdateSize(tree, $v$);
UpdateSize($tree$, $v$)
tree.size $\leftarrow$ tree.size $+ 1$;
if $tree.left \neq \emptyset \wedge tree.split \notin v$ then
    UpdateSize(tree.left, $v$);
else
    tree.right $\neq \emptyset \wedge$ tree.split $\in v$
UpdateSize(tree.right, $v$/split);
```

Algorithm 9 presents the pseudo-code for updating the mass of a forest. The algorithm requires as an input a binary isolation forest and a set of example vertices. For each vertex it updates the counts of the internal nodes of each tree (lines 1–2). The update procedure is a recursive process that increments the size of the current node and then proceeds to the update of the child node that matches the split criterion of the current node (lines 5–7).

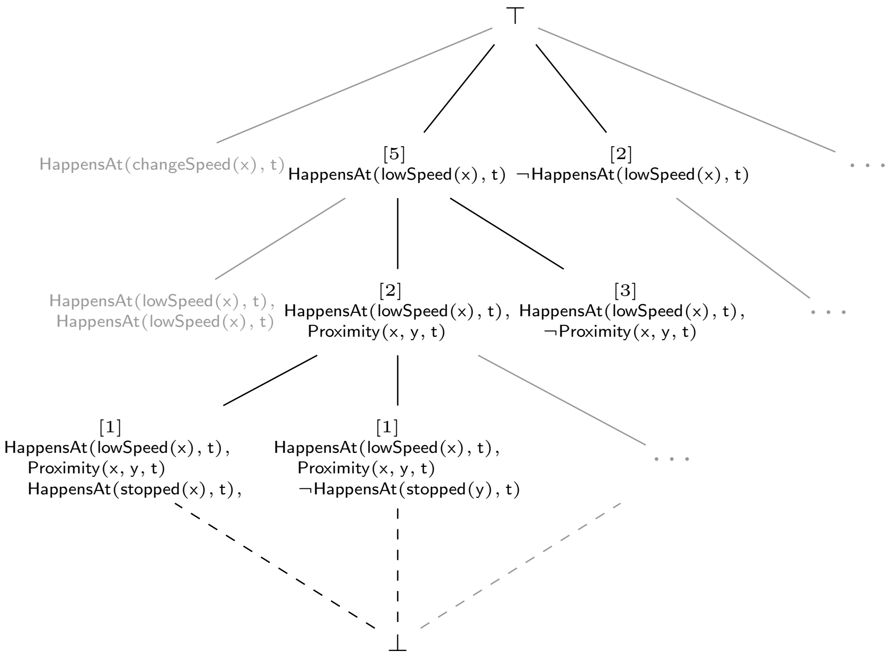

⊤
HappensAt(changeSpeed(x), t)
[5] HappensAt(lowSpeed(x), t)
[2] ¬HappensAt(lowSpeed(x), t)
...
HappensAt(lowSpeed(x), t), HappensAt(lowSpeed(x), t)
[2] HappensAt(lowSpeed(x), t), Proximity(x, y, t)
[3] HappensAt(lowSpeed(x), t), ¬Proximity(x, y, t)
...
[1] HappensAt(lowSpeed(x), t), Proximity(x, y, t) HappensAt(stopped(x), t),
[1] HappensAt(lowSpeed(x), t), Proximity(x, y, t) ¬HappensAt(stopped(y), t)
...
⊥

Figure 10: Path selected by a random tree from the subsumption lattice.

The intuition behind these relational version of Isolation Trees is that we estimate the mass of specific areas of the subsumption lattice generated from a given Herbrand base $\mathscr{B}$ and constrained by the mode declarations $\mathscr{M}$. Fig. 10 depicts a part of the subsumption lattice constructed from the atoms appearing in the training sequence of Fig. 8. The highlighted part of the lattice presents a possible Isolation Tree constructed by selecting one split atom per level, while the numbers represents the counts of each node. Therefore, each tree essentially represents only a part of the lattice and their mean represent the probability of the overlap of two rules to be located in a sparse part of the space. The overlap of two rules on a subsumption lattice is their least general generalisation.

The resulting Isolation Forest can be used to compute the mass-based dissimilarity of eq. (23) for example vertices as follows:

$$\tilde{m}(v_i, v_j) = \frac{1}{T}\sum_{i=1}^{T}\frac{|R(v_i, v_j|H_i)|}{|D|}$$

4.4 Robust Graph Construction and Labelling

In order to construct the similarity graph for performing label propagation we combine the mass-based dissimilarity as presented in Section 4.3.2 to the optimised structural distance of eq. (25) as follows:

$$d_h^{\mathbf{b}}(v_i, v_j) = \alpha\, d_s^{\mathbf{b}}(v_i, v_j) + (1-\alpha)\, \tilde{m}(v_i, v_j)$$

where α controls the balance between the two distances. Fully connecting the vertices generates a $N \times N$ symmetrical adjacency matrix $\mathbf{W}$ comprising the weights of all graph edges. In order to turn the similarity matrix $\mathbf{W}$ into a graph, we use a connection heuristic, which retain edges only between vertices that are very similar, i.e., they have a high weight. SPLICE$^+$ uses a temporal variation of kNN that connects each unlabelled vertex to its temporally adjacent vertices and its k-nearest labelled neighbours. Intuitively, that heuristic captures the notion that far away vertices should not directly affect the labelling of each other, but only through their temporal neighbours.

Moreover, in order to provide guarantees for the online labelling found by label propagation on the local graphs built as the micro-batches stream-in, we store a synopsis of the graph, as presented in Section 4.2. The synopsis removes older vertices from the graph in order to make room for newer ones by meshing their edges to the rest of the graph using star-mesh transforms. The harmonic solution computed on the compressed graph is guaranteed to be equal to the one computed on the entire stream seen so far. Therefore the synopsis renders the labelling invariant to different batch sizes.

ALGORITHM 10: GRAPHCONSTRUCTION($\mathscr{F}$, $\mathscr{V}$)

```
   Input: $\mathscr{F}$: a set of isolation trees, $\mathscr{V}$: a set of example vertices
 1 Partition example vertices into labelled and unlabelled $\mathscr{V} = (\mathscr{V}_L, \mathscr{V}_U)$;
 2 UPDATEFOREST($\mathscr{F}, \mathscr{V}_L^t \cup \mathscr{V}_U^t$);
 3 if $\mathscr{V}_L^t \neq \emptyset$ then
 4     $\mathscr{X} = \text{LMFS}(\mathscr{V}_L, \mathscr{M})$;
 5 $\mathscr{V}_\tau \leftarrow \mathscr{V}_L \cup \mathscr{V}_U / \mathscr{V}_U^t$;
 6 foreach $v_i \in \mathscr{V}$ do
 7     foreach $v_j \in \mathscr{V}_U^t$ do
 8         $w_{v_i,v_j} \leftarrow 1 - d_h(v_i, v_j)$;
 9 $V_\tau \leftarrow V_\tau \cup \mathscr{V}_U^t$;
10 Apply the connection heuristic: $\mathbf{W}_h = h(\mathbf{W})$;
11 while $|\mathscr{V}_\tau| > \tau + |\mathscr{V}_L|$ do
12     Find oldest vertex $v_o \leftarrow \mathscr{V}_\tau / \mathscr{V}_L$;
13     foreach vertex pair $v \neq v'$ in $\mathscr{V}_\tau$ do
14         $w_{v,v'} \leftarrow w_{v,v'} + \frac{w_{v_o,v} w_{v_o,v'}}{\text{degree}(v_o)}$;
15     Remove all $v_o$ edges from $\mathbf{W}_h$;
16 return $\mathbf{W}_h$
```

Algorithm 10 presents the pseudo-code for constructing the graph. The algorithm requires as an input a pre-built Isolation Forest, and a set of example vertices. The example vertices are partitioned into labelled and unlabelled at line 1. Then only the example vertices (both labelled and unlabelled) received in the current micro-batch t are used for updating the forest counts at line 2. Subsequently, if labelled vertices exist in the current micro-batch, the feature selection optimisation is re-computed to update the selected features (lines 3–4). In lines 5–9 the graph connection process takes place. Each stored vertex is connected to the unlabelled vertices received at micro-batch t. The set of τ stored vertices is composed of all the labelled vertices and the previously stored unlabelled ones. Then, the connection heuristic is applied at line 10. As a final step, while the number of stored vertices is greater than the given memory the algorithm evicts the oldest vertex along its edges and applies star-mesh transforms to its neighbours (lines 11–15).

4.5 Empirical Evaluation

We compare SPLICE$^+$ to its predecessor (SPLICE) on the task of complex event recognition using a real maritime surveillance dataset.

4.5.1 Experimental Setup

The maritime dataset consists of vessel position signals sailing in the Atlantic Ocean, around Brest, France. The SEs express compressed trajectories in the form of 'critical points', such as communication gap (a vessel stops transmitting position signals), vessel speed change, and turn. It has been shown that compressing vessel trajectories in this way allows for accurate trajectory reconstruction, while at the same time improving stream reasoning times significantly [116]. We focus on the `rendezvous` and `pilotOps` CEs. The former expresses an illegal activity where two vessels are moving slowly in the open sea and are close to each other possibly exchanging commodities, while the latter describes the activity of a pilot boat. Since the dataset is unlabelled, we produced synthetic annotation by performing CER using the RTEC engine [18] and hand-crafted definitions of `rendezvous` and `pilotOps` CEs. We have extracted 6 sequences for each CE. Regarding `rendezvous`, the total length of the sequences is 11,930 timestamps, while for `pilotOps` sequences comprise of 6,678 timestamps. There are 1,425 example instances in which `rendezvous` occurs and 769 in which `pilotOps` occurs.

The evaluation concerns two learning scenarios. In the first one, a number of micro-batches was selected uniformly at random to remain fully labelled while the rest of the micro-batches arrive completely unlabelled. We experimented using 5%,10%,20%,40% and 80% of the total supervision, retaining the corresponding proportion of labelled micro-batches. We repeated the uniform selection 20 times, leading to 20 datasets per supervision level, in order to obtain a good estimate of the performance. This scenario was the one assumed by SPLICE [103] and thus we also present results on it here for consistency. However, on a typical learning task from data streams, usually, the assumption of labels arriving on-stream is unrealistic. A more appropriate situation is when a training set appears in the beginning of the stream, or stored in a database as historical data, while the rest of the data stream-in completely unsupervised. To that end, the second scenario assumes a number of fully labelled training sequences is provided in the beginning of the stream, while the rest of the data arrive completely unsupervised. We experimented using 1, 2 and 4 labelled training sequences. We created all possible datasets having a single labelled training sequence, while the rest of the sequences, for creating 2 and 4, were randomly selected. We considered only sequences that contained both positive and negative examples, leading to 30 datasets.

Throughout the experimental analysis, we used T=100 isolation trees for computing the mass-based dissimilarity and set α=0.5 in order to balance the influence of each distance measure. Following the experimental results of SPLICE we chose to run the evaluation using k-nearest neighbour for k=1 and k=2. The accuracy results for supervision completion were obtained using the F_1-score. All reported statistics are micro-averaged over the instances of CEs. The experiments were performed in a computer having an Intel i7 4790@3.6GHz CPU (4 cores, 8 threads) and 16GiB of RAM.

4.5.2 Experimental Results

We compare the performance of SPLICE$^+$ against SPLICE, for both `pilotOps` and `rendezvous` CEs. Figure 11 depicts the F$_1$-score achieved by the supervision completion on both scenarios. The notation k_t refers to temporal kNN, as described in Section 4.4, while d_s and $d_h^{\mathbf{b}}$ refer to the structural and hybrid distances respectively. The results suggest that SPLICE$^+$ effectively infers the missing labels and its performance increases as more supervision is given. More importantly, it significantly outperforms SPLICE in all cases for both k=1 and k=2 even for high supervision levels (80% uniform supervision or 4 sequences). As expected, the difference is greater in the second scenario since labelled data are provided only in the beginning of the training sequence making available labels unrepresentative of the actual distribution that the underlying classes follow. For SPLICE, $k = 2$ performs much better than $k = 1$. SPLICE$^+$, on the other hand, performs better for $k = 1$, however, it also performs well for $k = 2$, since the F$_1$-score does not drop significantly, which is an indication of robustness to noise in the neighbourhood.

On the other hand, the improved performance comes at the cost of a decrease in runtime, as shown in Figure 12. The presented runtime is macro-averaged over all samples. Note that SPLICE$^+$ is always slower than SPLICE since it has to update the isolation trees for every micro-batch, as well as selecting appropriate features, which are time-consuming. However the penalty is tolerable since, taking into account the standard deviation, it lies between 10 to 25 seconds for the whole training procedure. Note that there is a significant increase in runtime, on the first scenario between 5% and 20%, that decreases as more supervision is given. That is due to the fact that SPLICE$^+$ runs feature selection only when a labelled micro-batch arrives followed by an unlabelled one. Therefore, in the cases of 20% or 40%, such sequences occur much more frequently than 5% or 80%. The same applies to the second scenario, where the supervision only appears in the beginning of training, thus feature selection only runs once.

In Table 7, we present the change in F$_1$-score as the batch size increases. Note that SPLICE tends to have larger changes in F$_1$-score compared to SPLICE$^+$as the batch size increases. For instance, on `pilotOps`, when 1 supervised sequence is provided, SPLICE accuracy varies from 0.02 to 0.1, while SPLICE$^+$ varies only 0.01. For 2 supervised sequences the variation is even greater, since SPLICE varies from 0.03–0.18, in contrast to SPLICE$^+$ where the variation remains 0.01. The same situation holds for the `rendezvous` CE, where for 1 supervised sequence, SPLICE varies from 0.08–0.15, while SPLICE$^+$ varies only 0.01. These results suggest that SPLICE$^+$ seem to be more robust to different batch sizes than its predecessor.

CE	Batch size	Number of supervised sequences		
		1	2	4
`pilotOps`	10	0.63/0.96	0.88/0.97	0.92/0.97
	25	0.69/0.96	0.85/0.96	0.91/0.97
	50	0.71/0.96	0.88/0.96	0.91/0.97
	100	0.61/0.95	0.70/0.96	0.75/0.97
`rendezvous`	10	0.63/0.74	0.77/0.86	0.87/0.93
	25	0.58/0.74	0.72//0.86	0.84/0.90
	50	0.56/0.74	0.75/0.86	0.83/0.90
	100	0.48/0.75	0.61/0.81	0.83/0.92

Table 7: F$_1$-score for varying batch sizes for `pilotOps` and `rendezvous` (SPLICE/SPLICE$^+$).

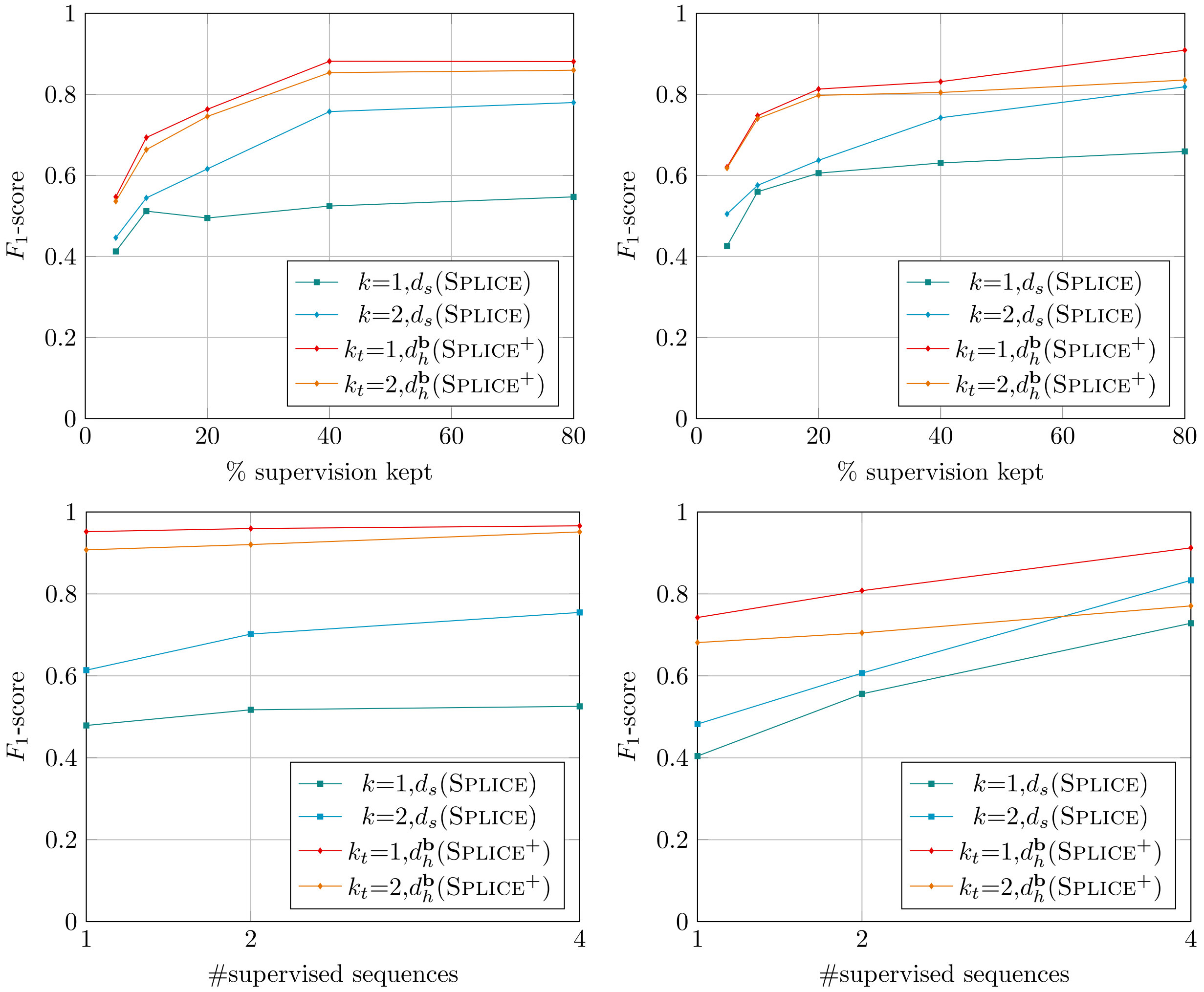


Figure 11: F$_1$-score of supervision completion on `pilotOps` (left) and `rendezvous` (right) as supervision increases. First scenario, where supervision arrives uniformly (top), and the second one, where supervision is provided only in the beginning of the training sequence (bottom).

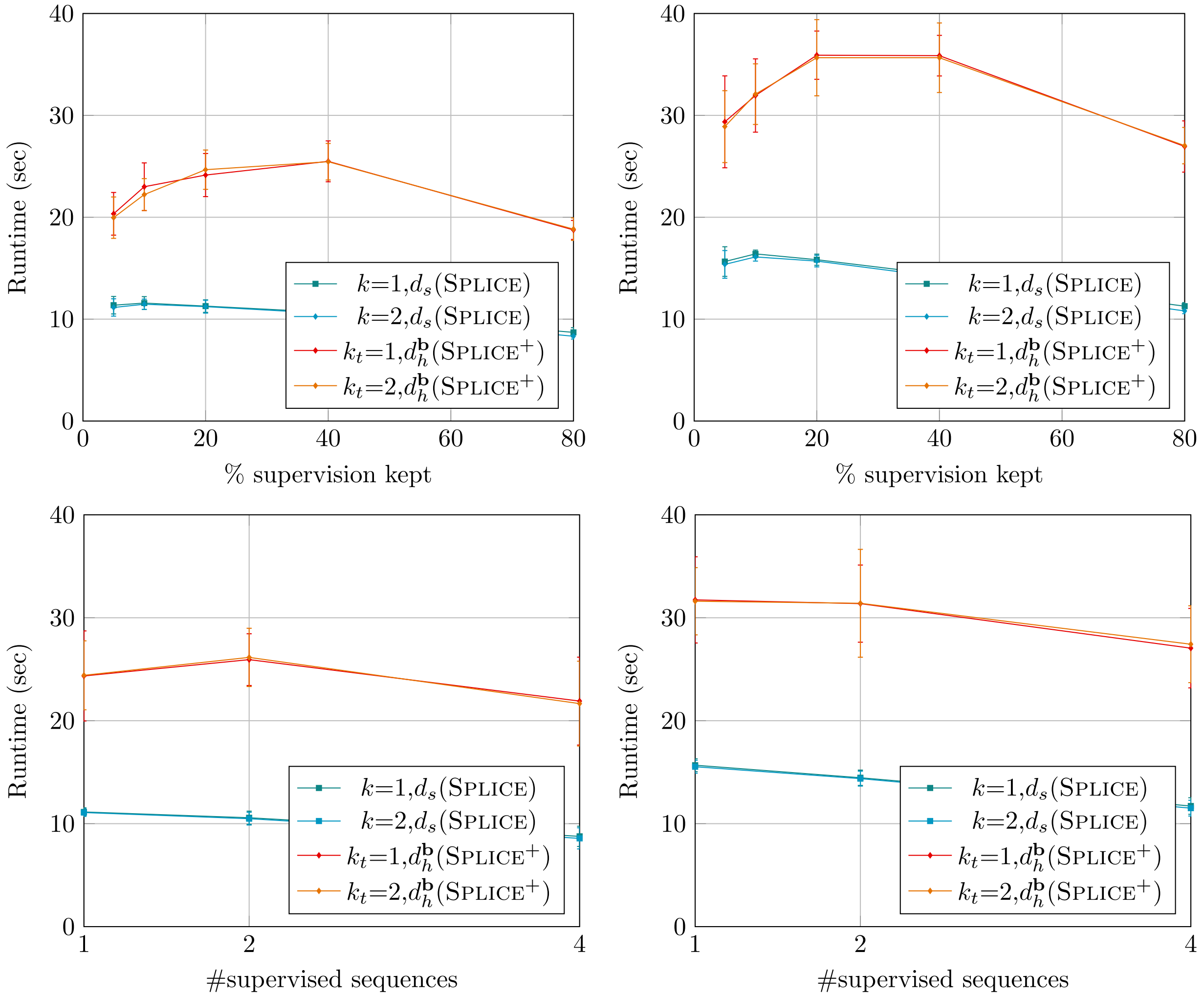


Figure 12: Runtime of supervision completion on `pilotOps` (left) and `rendezvous` (right) as supervision increases. First scenario, where supervision arrives uniformly (top), and the second one, where supervision is provided only in the beginning of the training sequence (bottom).

5 Online Machine Learning and Data Mining Component

This section outlines the design and implementation state of the Online Machine Learning and Data Mining (OMLDM) component of the INFORE architecture. The goal of the OMLDM component is to provide a general-purpose infrastructure for a broad collection of extreme-scale training of online machine-learning and data mining algorithms, that will work seamlessly with other components of the INFORE platform.

5.1 Overview of the OMLDM component and previous art

Extreme-scale online learning has been approached in many different ways. In the OMLDM component, we strive to achieve this by distributed computation, which can unleash the inexhaustible potential of modern cloud-based infrastructures. To this end, the OMLDM component is first and foremost a high-performance computing platform, able to perform a broad family of distributed learning algorithms on rapid streams of training data. In addition, the component has to provide excellent support for model-based prediction over massive data streams.

Several high-quality frameworks for large-scale machine learning and data mining have existed for many years, and it is not our intention to build yet another such framework. For example, RapidMiner and ADAMS (among many others) provide excellent end-to-end facilities, including great visualization and interactivity. Vowpal Wabbit, SparkML, scikit.learn, ELKI, and many others, host a very broad pallette of different algorithms, while Petuum and DistBelief can scale to thousands of cores.

First and foremost, the scope of OMLDM component is focused on a particular scenario; online *and* distributed machine learning and data mining. In this arena, the competition from previous work is much more limited. Although, in some sense, many platforms out there could be considered to offer some distributed OMLDM facilities, to our knowledge the only platform with a clear commitment to the same goals as the OMLDM is SAMOA.

It should not come as a surprise that OMLDM and SAMOA have many similarities in common. Both platforms are specialized to distributed machine learning over rapid, massive data streams. Both platforms are embedded into large scale distributed stream processing platforms—Samza for SAMOA, although it is portable to others—and Flink for OMLDM. Both frameworks employ these platforms for scalability and fault tolerance.

However, OMLDM takes a decidedly different approach to SAMOA when it comes to its model of computation. The architecture of SAMOA follows the Agent-based pattern; an algorithm is a set of distributed processors that comunicate with streams of messages. Little more is provided, and this is intentional [85]. It is argued that providing a more structured model of computation, reduces the applicability of the framework unacceptably.

We do not agree with this proclamation, and we are building OMLDM in order to validate our own vision. Our system architecture is much more structured. We follow a *parameter server* distributed model, where a collection of identical learners distribute amongst themselves the computational load of training on an incoming data stream of training samples [93]. The OMLDM platform aims to provide a set of well-engineered. high-performance-oriented policies for orchestrating the distributed computation. In addition, the platform strives to supply a number of added-value services, such as monitoring the state of the learning task and allowing the user to interact heavily with it. Used within the INFORE architecture, users of our component will benefit from a synergistic set of services, including learning from synopses maintained by the Synopsis Data Engine (SDE), and being automatically tuned by the workflow optimizer.

In the following, we present the current design and a high-level view of the implementation of the OMLDM component. First, we describe the abstractions it offers to the user, and also to the developer. Then, we discuss some of the technical aspects of our research and finally we present the salient points of our implementation on top of Apache Flink.

5.2 Architecture of the OMLDM component

The top-level concept in the architecture of OMLDM is a *pipeline*. A pipeline is a linear sequence of stages, with a stream of tuples entering the first stage. Each subsequent stage applies some operation on the incoming data and generates data for the next component in line. The last stage may output a stream of data to the rest of the system. Pipelines are of two types, training pipelines and prediction pipelines.

Training pipelines are specialized to performing data mining on the incoming stream, in order to train some Machine Learning model on the data or estimate some other statistical model, such as advanced summarization, feature extraction, clustering, etc. Training pipelines are data sinks, that is, they do not generate an output stream. Instead, they maintain a model and make it available on-demand to other operators of an INFORE workflow.

Prediction pipelines are transformations of the incoming data stream to an outgoing data stream. Although in principle these transformations can be arbitrary, the intended purpose of this type of pipeline is to employ some model for classification, interpolation or inference. The result of this transformation is an output stream which can be processed further by the INFORE workflow.

5.3 Training pipelines

A training pipeline accepts a stream of training samples and maintains online a *model* trained on these samples. Structurally, the pipeline contains a sequence of *preprocessing* stages, and terminates at a *learning* stage, as per Fig. 13.

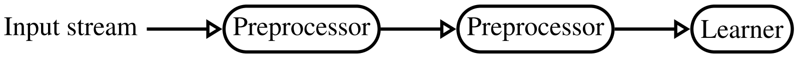


Figure 13: Structure of a training pipeline.

Preprocessors are customizable by hyper-parameters. Two useful preprocessors are the following:

Scaler This preprocessor transforms individual features (coordinates) of the feature vector of a training sample. The output of this transformation consists of features with a mean of 0 and scaled to multiples of the standard deviation of the input.

Polynomial features This preprocessor extends the feature vector of each training sample by adding powers of the initial features, up to some predefined degree (a hyperparameter of the preprocessor). For example, the 2nd degree transformation on a vector with two features is the following mapping:

$$(x_1, x_2) \to (x_1, x_2, x_1^2, x_1 x_2, x_2^2)$$

After preprocessing, the incoming stream of training samples is forwarded to the learner. A configurable portion of the stream (say 3%), is sampled and stored in a buffer, in order to perform periodic *scoring* of the learned model. The bulk of the stream is used to fit the model at the learner.

The learner stage supports parallel, high-throughput processing of the input stream, as it continually fits a model on the training samples. It is implemented as a collection of parallel instances of a learning algorithm, called *local learners*. Each local learner receives a *partition* of the incoming data stream and fits a local model to this partition. Periodically, the local learners synchronize with a *Parameter Server*, which maintains the global model. The OMLDM component's implementation can assign each local learner on a different compute node, which can be multithreaded and/or may support GPU-based or TPU-based accelarators. Furthermore, in order to minimize communication overheads, the preprocessor stages are also parallelized, and each local learner is co-located on its node with a parallel instance of its preprocessors. This architecture is depicted in Fig. 14

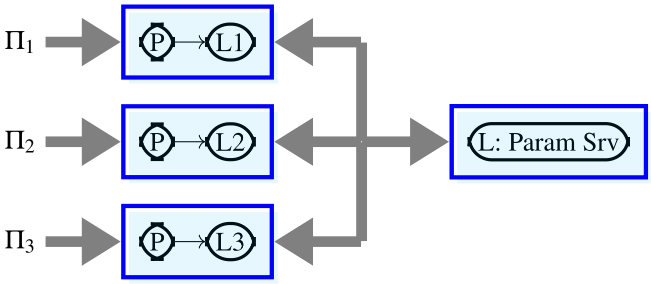


Figure 14: A training pipeline of 2 components, with parallelism set to 3. The diagram shows allocation of objects to distributed nodes (blue rectangles). Three of the nodes, accept partitions Π_i, $i = 1, \ldots, 3$ of the input stream. The preprocessor component is replicated in each node. The local learners synchronize periodically with the Parameter Server.

5.4 Prediction pipelines

Prediction pipelines accept a data stream of feature objects (vectors) and annotate each object with additional information, inferred by some learned model. A prediction pipeline is a linear sequence of stages, applying transformations and filtering. The output of the prediction pipeline is a stream of transformed objects.

Prediction pipelines do not train the models they apply; these models are loaded externally. A model can be changed dynamically during execution of the pipeline, if there has been *concept drift* in the data stream. A prediction pipeline may need to contain preprocessor stages that are compatible with the preprocessing that occurs during training of the model. For example, assume that some application requires labeling an incoming stream with the output of some classifier C. Furthermore, assume that during training (in another pipeline) of classifier C, the feature vector of each sample was augmented with additional Polynomial features. In order to apply this classifier to unlabeled records, the features of each unlabeled record need to be complemented with the result of the Polynomial features preprocessor. Furthermore, the smae pipeline may apply multiple inferences per sample; for example, one may apply both a classifier and a regressor on the every sample.

In order to achieve high throughput, prediction pipelines are embarrassingly parallelizable. Both input and output streams are *partitioned* into a number of partitions, which is matched by the parallelism of the inference process. For correctness, all parallel instances of the pipeline objects are identical and are kept identical during dynamic updates. This leads to a very simple architecture, depicted in Fig. 15.

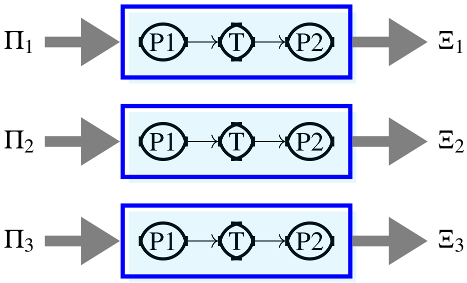


Figure 15: A prediction pipeline of 3 stages, with parallelism set to 3. The first stage is preprocessor P1, the second stage is transformer T and the third stage is postprocessor P2. The diagram shows allocation of objects to distributed nodes (blue rectangles). Each node i, accepts partition Π_i, $i = 1, \ldots, 3$ of the input stream and generates partition Ξ_i of the output stream.

5.5 Implementing Machine Learning and Data Mining algorithms

The OMLDM component is not dedicated to any particular ML or DM task in INFORE, but strives to integrate many such tasks under the same architecture, in order to provide a uniform, well-tested, high-throughput implementation which is easily customizable with new algorithms. To this end, it employs the paradigm of *model averaging* in order to distribute large-scale training computations on several nodes.

Model averaging applies to models which are representable as high-dimensional vectors, and where the learner incrementally updates this vector by repeatedly adding small increments. In particular, we assume some model update rule of the form $w_{t+1} = w_t + F(w_t, x_t)$, where w_t is the model being trained at time step t and x_t represents a subset of training samples processed at step t. This type of iteration is extremely common in ML and DM online streaming techniques.

The implementation of an algorithm in OMLDM requires customizing (via inheritance) a number of classes, that can then become instantiated into training and/or prediction pipelines. These classes are of tree types, and for each type there is an interface (declared in Scala) that they must implement.

Parameter servers hold and update the state of the learning process, including the current version of the trained model.

The parameter server interface contains the following methods:

Method	Description
push(update)	Add an update to the current model.
pull()	Return the current model.
marshalModel()	Return a textual representation of the current model.
unmarshalModel(mtxt)	Set the current model by parsing the provided textual representation.

Learners compute the updates to the model and coordinate via the paramater server. Learners also act as predictors, if passed an unlabeled sample.

The learner interface contains the following methods:

Method	Description
fit(x)	Compute an update to the current model, using the training sample (x,y).
score(x)	Return a score (loss) reflecting the accuracy of the model on training sample x.
predict(x)	Return a vector y representing the current model's prediction for sample x.
setHyperParam(p,v)	Set the value of hyperparameter p
getHyperParam(p)	Return the value of a hyperparameter
marshalModel()	Return a textual representation of the current model.
unmarshalModel(mtxt)	Set the current model by parsing the provided textual representation.

Preprocessors transform data samples before the learning algorith is applied, and/or after a prediction is made (in prediction pipelines).

The preprocessor interface contains the following methods:

Method	Description
transform(x)	Return a vector x' representing the result of the transformation applied to x.
setHyperParam(p,v)	Set the value of hyperparameter p
getHyperParam(p)	Return the value of a hyperparameter

5.5.1 Algorithms supported

We have implemented some standard online machine learning algorithms, which we describe briefly.

Passive Agressive Classifier. Proposed in [35], it is a binary linear classifier. The model is a vector $w \in \mathbb{R}^n$. A sample $x \in \mathbb{R}^n$ is classified as

$$\operatorname{sign} w^T x \in \{-1, -1\}$$

The algorithm has one hyperparameter, the *aggressiveness* C (typically set to 0.01).

Passive Agressive Regression. This online regression scheme maintains an online linear regression $y = w^T x$. An additional hyper-parameter is the *sensitivity* ε.

Support Vector Machine (SVM). One of the most funndamental techniques in machine learning, provides a *maximum margin* linear classifier.

Hoeffding Trees where proposed in [44] under the name VFDT (Very Fast Decision Tree) and has become the de facto standard for incremental (streaming) decision trees.

Online Ridge Regression. Ridge regression (aka. regularized linear regression) is useful in under-constrained problems, where the dimensionality of the model is comparable (maybe exceeds) the number of samples.

AutoRegressive Iterated Moving Average (ARIMA). This is a well-known technique for autoregressive forecasting of time series.

Online *k*-means++. This is the well-known unsupervised learning algorithm, whose model represents a multiclass classifier. Our implementation exploits coreset synopses maintained by the Synopsis Data Engine.

BIRCH. This is another very popular unsupervised clustering algorithm, introduced in [152]. It is highly scalable, as it coalesces samples into *clustering features* (CF), which are represented succinctly. Our implementation can exploit the Synopsis Data Engine in collecting CF.

By implementing the above methods for each type of pipeline stage, the pipeline is able to orchestrate the flow of data through its stages and drive the computation forward. The responsibilities of a pipeline include:

- Controlled initialization of all stages, on different nodes. Some learning algorithms require information about the level of parallelism, or other global information.
- Efficient transfer of data through the pipeline. Incoming streaming data is batched in customizable ways and processing can occur on a per-minibatch basis.
- Distributed checkpointing of the pipeline state on each node, in order to support restart semantics in case of failure.
- Restoration of state of a node, after a failure. To this extent, the pipeline manager on a local node which is restarted after a failure, is respo
- Avoiding race conditions when the hyperparameters of various stages are changed interactively. Since pipelines are distributed on multiple nodes, a hyperparameter change may, if not done carefully, lead to inconsistencies among nodes.
- Collection of statistics related to the execution. In particular, two types of statistics are maintained; statistics that concern the performance of the computation (samples processed, processing rate, etc), and statistics that concern the state of the computation. The latter is meaningful mostly for training pipelines, but is also useful in debugging pipelines.

  An important statistic is the prequential score [58] for learners; a customizable fraction of the training data is used to score the model continuously; naturally, this data is then used for fitting as well.

5.6 Distributed coordination for training pipelines

In addition to the responsibilities outlined previously, for training pipelines, it is the pipeline's responsibility to synchronize the processing between the local learners and the parameter server. The synchronization policy is extremely important to the performance of the training computation, both in terms of the quality of the learned model (accuracy, size) and in terms of computational cost (CPU utilization, communication cost, memory consumption etc).

Ideally, we would like to make the details of this policy customizable at a fine grain; in practice, this is too complex to do correctly. Therefore, we decided to support a set of discrete policies, with little customization in each policy. We note that this is the approach followed by the majority of previous ML platforms.

In OMLDM we currently support three distinct policies:

- Bulk Synchronous Parallel policy (BSP)

- ASP Asynchronous Parallel policy (ASP)
- Dynamic Averaging policy (DA)

The synchronous policy (BSP) works in rounds. In each round, a mini-batch of training data is allocated to each learner. Each learner obtains the current model w_t from the parameter server and fitting is performed. Each learner forwards its update to the parameter server and w_{t+1} is computed. This policy is popular in theoretical analyses of machine learning algorithms, and exhibits good learning in practice; however performance-wise it is not very scalable; when many learners are used, the total utilization is low, because of stragglers. However, when the performance of the system is network-bound instead of CPU-bound, this method may be preferable, since the stragglers' delay will be masked by network delays.

The asynchronous policy (ASP) on the other hand does not enforce any synchronization between learners. When a learner is allocated a training minibatch, it receives the current model from the parameter server, fits the model using the minibatch and sends its update to the parameter server. Parameter servers sequentially apply updates to the model, as they receive them. This policy is the policy of choice in large-scale machine learning; although it may lead to slightly slower convergence in terms of the amount of training data processed, the processing speed is much higher when many learners are used, and therefore training is more efficient.

One problem with both the synchronous and the asynchronous method is the choice of the right batch size. There is empirical evidence [83] that large minibatch size tends to overfit. On the other hand, a small minibatch size (32 to 256 samples is the pervailing rule of the thumb) results in too frequent interactions with the parameter server and therefore high network cost.

The dynamic averaging policy was introduced by Kamp et al. [76, 77, 75] and takes a different approach to synchronization. The protocol is synchronous in that it works in rounds. At the beginning of round t, the model w_t is sent to every learner. However, instead of of a fixed batch size, each learner is provided with a threshold Δ_t. The learners start updating their local copy of the model, by fitting incoming training samples independently of each other. Each learner monitors the local condition

$$\|w^{(i)} - w_t\|^2 \leq \Delta_t,$$

where $w^{(i)}$ is the (updated) model at learner i. When this condition fails in some node, the parameter server collects all local updated models and updates the global model by

$$w_{t+1} = \frac{1}{k}\sum_{i=1}^{k} w^{(i)}, \tag{26}$$

where k is the number of learners. Because of the convexity of the Euclidean norm, this process guarantees the following constraint at each round:

$$\|w_{t+1} - w_t\|^2 \leq \Delta_t. \tag{27}$$

DA has been empirically evaluated on several learning scenarios, where it was shown to outperform BSP or ASP.

5.7 Functional Dynamic Averaging

Dynamic averaging is a method inspired from Geometric Monitoring techniques, developed for distributed streaming algorithms. The success of DA prompted us to test a more recent improvement, the Functional Geometric Monitoring of [130]. We propose Functional Dynamic Averaging (FDA), a synchronization policy that retains the quality of training afforded by the DA policy but reduces the communication cost even further.

Our starting point is a constraint of the form

$$U(w_{t+1}, w_t) \leq \Delta_t. \tag{28}$$

Function U could be the squared Euclidean norm as in DA, or any other function; indeed, the choice of the Euclidean norm is rather arbitrary in many conceivable scenarios. Given w_t, U and Δ_t, the FDA introduces the concept of a *safe function*:

Definition 4. *Given U, w_t and Δ_t, a function $\phi : \mathbb{R}^n \to \mathbb{R}$ is safe if*

- *ϕ is convex, and*
- $\forall u \in \mathbb{R}^n,\ \phi(u) \leq 0 \implies U(w_t + u, w) \leq \Delta_t$.

Our protocol, similarly to DA, works in rounds. At the beginning of a round, the parameter server ships w_t to each learner. Learners start to fit based on the inconing training data. Collectively, the system monitors the following value:

$$\psi = \sum_{i=1}^{k} \phi(w^{(i)} - w_t).$$

The round lasts, as long as $\psi \leq 0$. When the round ends, the protocol behaves similarly to the DA; the updated local models are sent to the parameter server, and the global model is updated as per Eq. 26.

Proposition 1. *The FDA protocol guarantees the constraint of Eq. 28.*

Proof. The key property is the convexity of ϕ. As long as $\psi \leq 0$, we have

$$\psi = \frac{1}{k}\sum_{i=1} \phi(w^{(i)}) \leq \phi(\frac{1}{k}\sum_{i=1}^{k} w^{(i)}) \leq 0.$$

The last inequality and the safety of ϕ imply the condition of Eq. 28. □

As an example, by setting $\phi(u) = \|u\|^2 - \Delta_t$ we obtain exactly the condition of Eq. 27, which is the condition of DA. However, even with this choice of ϕ, a round in FDA will last much longer that a round in DA. Indeed, a DA round ends the moment $\phi(w^{(i)})$ becomes non-negative at any learner i, whereas in FDA, it is the average (ψ/k) that must become non-negative, to end the round. This implies less communication between learners and the coordinator.

The FDA has been experimentally evaluated on various ML scenaria. It is shown to be much more scalable than DA when the number of learners grows. See Fig. 16.

The FDA protocol is under actively researched and more results will be available as the project progresses.

We have extended our experimental evaluation of FDA in settings including Convolutional Neural Networks and the Extreme Learning Machine [70].[10]

[10] We are in the process of writing a conference paper where we report on our empirical results.

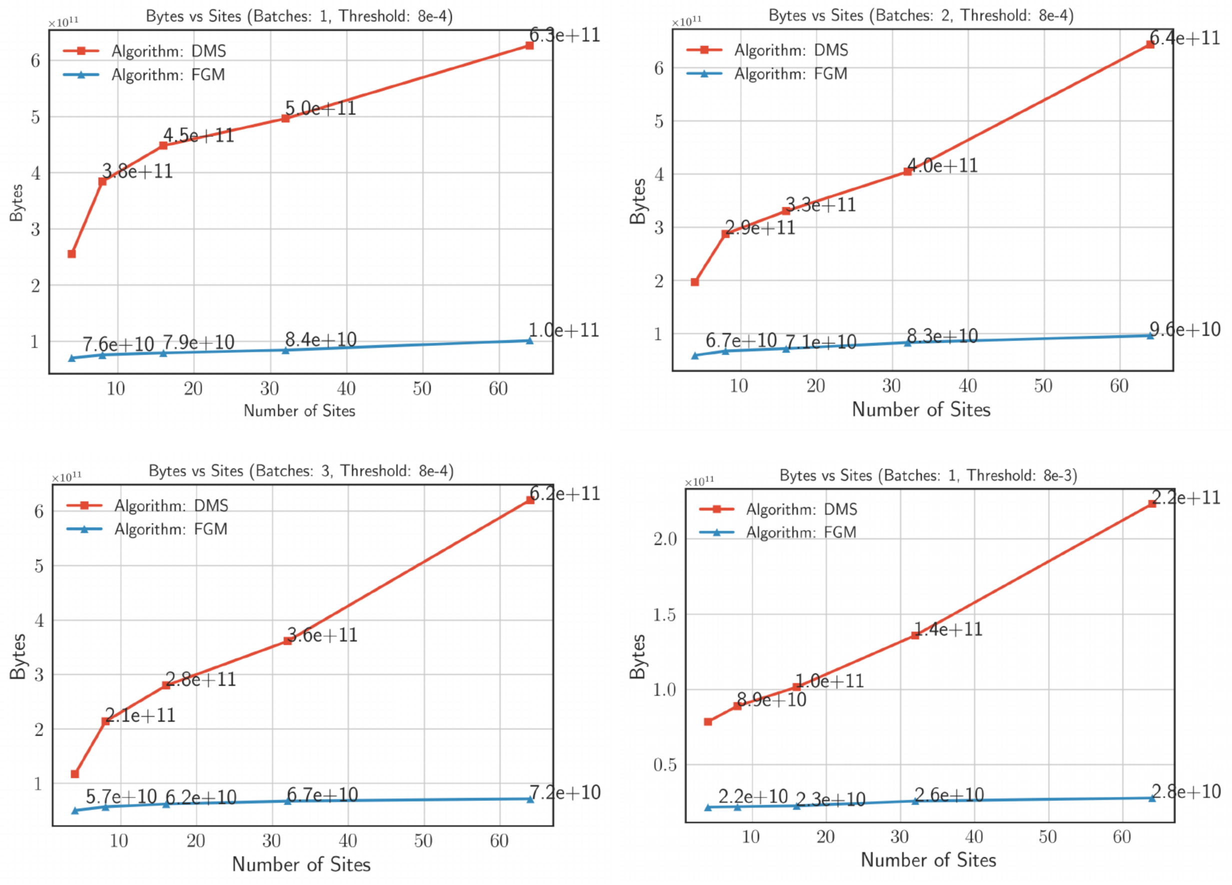


Figure 16: A comparison of the communication cost of several FDA (blue line) and DA (red line) scenaria, with CNN learners.

In the latter case we have shown that the Adaptive Online Sequential ELM variant [30], in a scenario with significant concept drift, achieves about an order of magnitude less communication under the FDA policy compared to the DA policy, and in addition achieves much better learning accuracy.

In addition, in all our experiments it was demonstrated that the FGM is much more robust in scenaria where the streaming rates at different learners are significantly different. This scenario is motivated by a case where some of processing nodes in the system are equipped with GPU and TPU accelerators, whereas others aren't.

5.8 Implementation of the OMLDM component

In this section we report on the the implementation of the OMLDM component, its current state, future directions, and on the challenges that we faced.

The OMLDM component is implemented on top of Apache Flink 1.9. Most of the implementation is done in Scala.

The component can be run independently of other components, but there is providence for tighter integration with other parts of the INFORE platform, including the optimization module and the user interface.

Operationally, an instance of the OMLDM component is deployed on the cluster as a Flink *job*. We refer to this instance as an OMLDM job (or simply a job). One important design choice was that there may be several OMLDM jobs within an INFORE workflow. This choice allows for future scenaria where an INFORE workflow may span multiple compute clusters.

Each OMLDM job processes a single type of data. If a particular INFORE workflow requires processing on different types of data (for example, both Level 1 and Level 2 financial data), there need to be two distinct OMLDM jobs in the workflow. This choice was dictated by particular limitations imposed on Flink jobs; namely, a Flink job declares a maximum level of parallelism that is not changeable at runtime.

Communication with the outside world happens through Apache Kafka, in accordance with the overall INFORE architecture. Each instance of the component interacts with at most 5 Kafka topics:

Training data stream: This topic provides training data to training pipelines.

Prediction input stream: This topic provides unlabeled data to prediction pipelines.

Prediction output stream: This topic outputs the result of prediction pipelines.

Control request topic: This topic is used to receive control requests from the INFORE workflow orchestration.

Control response topic: This topic is used to publish responses to control requests.

Each OMLDM job can support multiple pipelines. The data arriving at the training data stream is delivered to all training pipelines. Similarly, the data received from the prediction input stream is forwarded to all prediction pipelines and the outputs of all pipelines are combined into the prediction output stream. Overall, the flow of data in a job is shown in Fig. 17.

Resource type	Create	Read	Update	Delete	Remarks
Job	-	get full status	-	-	
Pipeline	create	get full status	-	delete	
Model	-	get current	set current	-	Only for training pipelines, obtained from Param server.
Score	-	get current model score	-	-	Only for training pipelines.
Parameter	-	get value	set value, if it is updateable	-	By name.
Preprocessor	-	get full status		-	
Learner	-	get full status		-	Only for training pipelines
Param Server	-	get full status		-	Only for training pipelines

Table 8: The control API of an OMLDM job. Empty cells imply that the call is unsupported

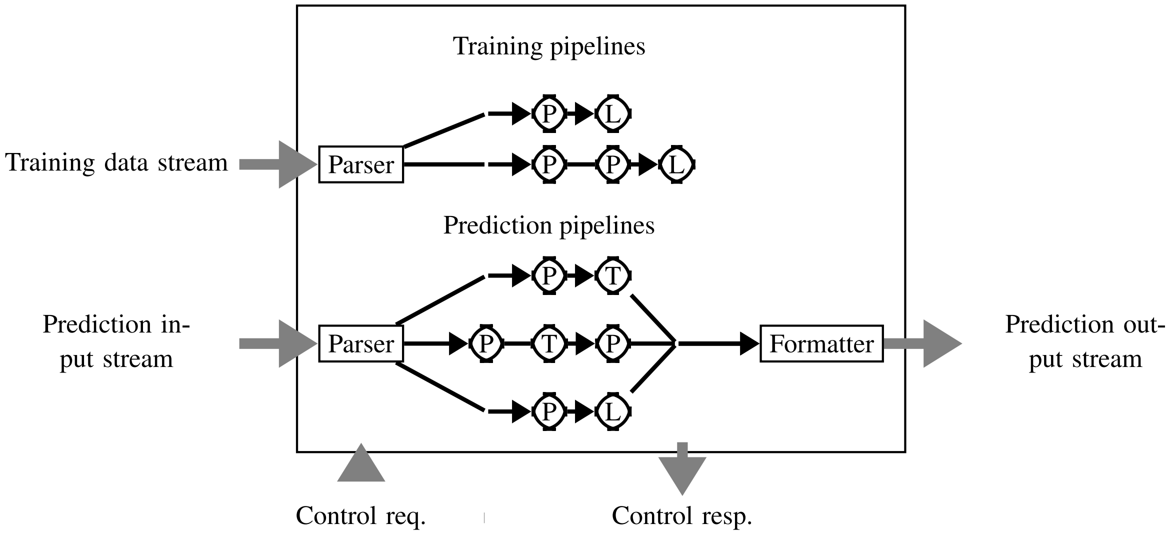


Figure 17: Overall architecture of a OMLDM job.

The control topics deliver control messages to and from a job. These messages follow the request-response pattern. They implement a *ReSTful*-like API, although the message format is not http, but instead is based on JSON. Entities in a job have path-like identifiers and the job control API understands Create/Read/Update/Delete (CRUD) commands for each type of resource. A brief overview of these messages appears in Table 8.

5.8.1 Pipeline implementation issues

Pipelines are implemented by flatMap operators in Flink. A flatMap operator is a collection of Java objects, each living on a different node. The number of these objects is the flatMap operator's parallelism. A flatMap is equipped with a callback, which Flink calls every time there is more data to be processed. Unfortunately, Flink does not allow for new operators to be created dynamically, while in OMLDM we support dynamic creation and deletion of pipelines. Therefore, all pipelines are implemented on a fixed number of Flink operators, as shown in Fig. 18.

All flatMaps (including the parsers) are fully parallel, so that the load can be distributed on multiple compute nodes.

The training pipelines are implemented by a pair of flatMaps. The iterative nature of learning and mining computations requires communication between all parallel nodes, and this is done by the parameter servers in each pipeline. When the parameter servers respond to learners, they need to perform "upstream" communication. Such a flow would cause the Flink operator graph to be cyclic, which Flink does not allow; however, in order to support iterative computations, it supports a limited and controlled manner of upstream data flow.

Unfortunately, the current support for iteration leaves too much to be desired. Using the iterative construct in Flink, invalidates the fault tolerance mechanisms that Flink supports. That is, in case of a node failure, there will be loss of data from the system. This can be quite catastrophic, since it is impossible to detect. It could very well lead to deadlocks in our processing logic.

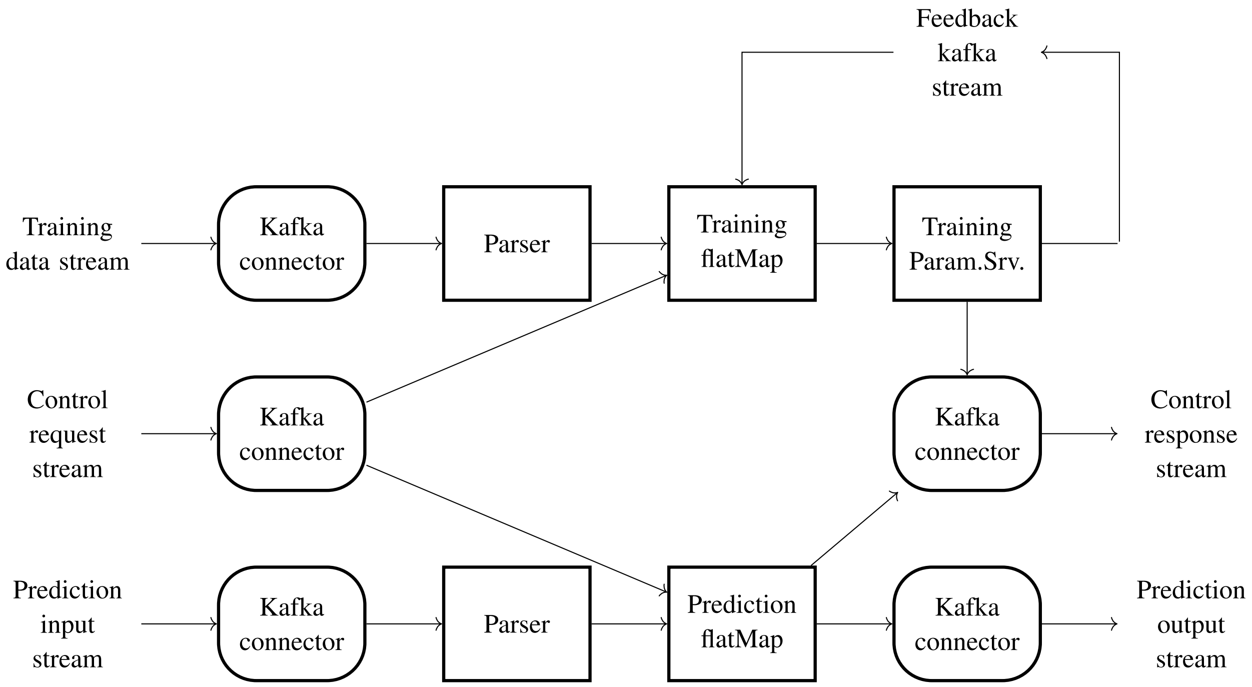


Figure 18: A simplified view of the implementation of OMLDM in Flink. The Flink operators form a directed acyclic graph. Square nodes indicate flatMaps, whereas rounded nodes are connectors. The feedback Kafka topic is used to implement upstream communication from the parameter server flatMap back to the Training flatMap.

Therefore, we investigated alternative ways to implement the upstream communication functionality. One option, which is compatible with fault tolerance, is to use a dedicated Kafka topic to perform upstream communication. This scheme can be made fault tolerant, because Kafka does not lose records in case of failure. Therefore, by inserting marker records in the feedback Kafka stream, using the Chandy-Lamport algorithm.

A concern with this scheme was performance. To this end, we performed timing experiments to discern the overhead of the feedback communication. Surprisingly, we found the the overall performance of this scheme is better than Flink's native iteration mechanism, as shown in Fig. 19.

5.9 Future directions

The OMLDM is currently actively developed. The features described in this section have been tested reasonably well and are likely to remain in the design, but as the integration of the INFORE platform progresses, new features are likely to become important.

The OMLDM component is an excellent testbed for much research in the context of WorkPackage 6. The main research directions we are currently pursuing are the development and empirical evaluation of more distributed ML and DM algorithms, development of the FDA synchronization policy and scalability to extreme data rates via model-based parallelism.

In terms of algorithmic development, we note that distributed online machine learning is a very active research area,

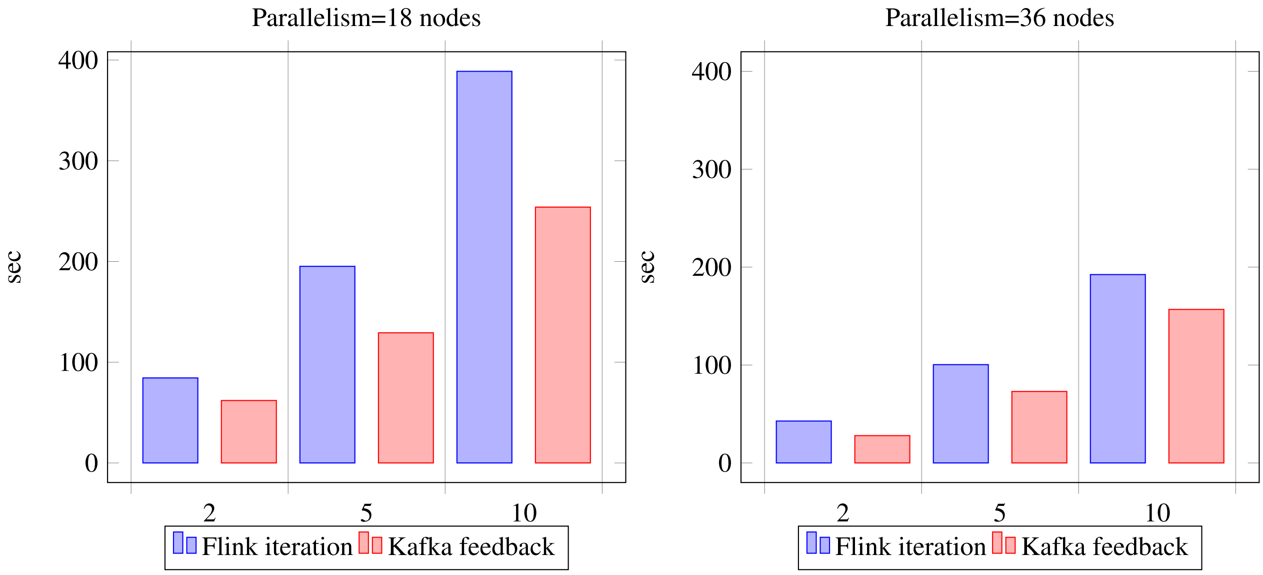


Figure 19: Total time for processing a dataset in OMLDM, with feedback channel implemented by Flink iteration, or Kafka. The processed dataset sizes where 2, 5 and 10 million records. Parallelism was 18 or 36 nodes.

and new developments happen daily. The INFORE demonstrators provide an excellent opportunity to work on these problems with real-world datasets. We believe that the OMLDM component will prove a versatile tool in this regard.

Closely related, is our work on FDA synchronization for distributed learning. The broader idea behind FDA is to steer a distributed computation by lightweight monitoring of the *global* state of the processing. We believe that this method has the potential to make distributed Machine Learning both more efficient, in terms of performance, more effective, in terms of the quality of learning, and more user friendly, in terms of robustness with less need for fine tuning. Towards this vision, much work is needed. Our primary focus is on proving theoretical properties of FDA, to bring it on par with previous techniques, to study its effectiveness on a broad spectrum of machine learning techniques, and to constrast it emprirically with state-of-the-art competitors.

6 Forecasting for the Cancer Simulations of the Life Sciences Use Case

Our work for the Life Sciences use case (WP1) aims in contributing to the field of cancer research by using the expertise of the areas of forecasting and machine learning. More specifically, we explore the effect of several cancer treatments on cell populations, with the goal of finding interesting treatment properties that can provide some new perspectives for clinical research. Our contribution to the Life Sciences use case is manifold. The first is to create a dataset consisting of simulation results that correspond to different cancer treatments, which can be used for future cancer research. The main challenge is to create a diverse dataset, which contains simulations that correspond to interesting and implementable treatments, so that they can potentially give insights to researchers. Second, our work aims in learning how to forecast whether a running cancer simulation will eventually produce interesting results. Finally, although several differect forecasting approaches have been proposed, in order for our results to be interpretable and lead to usefull conclusions, we have created a symbolic form of our dataset that will be used for forecasting purposes (Section 7).

6.1 Cancer Simulations

The dataset that we created consists of multi-cellular tumour simulations that have been produced using PhysiBoSS [11] [92]. PhysiBoSS is an open source software that allows exploring the results of several environmental and genetic alterations to populations of cells. It employs MaBoSS [139], an environment for stochastic boolean modeling, that allows simulating populations of cells and modeling stochastically intracellular mechanisms. PhysiBoSS also uses PhysiCell [61], an open source agent-based modeling framework for multicellular simulations. Thus, for example PhysiBoSS can be used for studying heterogeneous cell populations' response to different treatments. PhysiBoSS allows users to define several system parameters, including global properties for the simulation, cell properties, properties of the computational network of the cells and properties about the initial configuration of the simulation.

We used PhysiBoSS to produce simulations that can be used for studying the response of heterogeneous cell populations to TNF treatment. Each simulation corresponds to a different combination of values for the following parameters:

- time_add_TNF : Frequency of TNF injections.
- time_remove_TNF : Time point at which TNF is removed from the system.
- duration_add_TNF : Duration of the TNF injections.
- TNF_concentration : Concentration of TNF.

The rest of the simulation parameters are fixed to their default values. Figure 20 shows the course of a simulation produced by PhysiBoSS. The figure contains three lines, one corresponding to the number of alive cells for each time point, one corresponding to the number of cells that have been programmed to die (apoptotic), and another line (necrotic) for the number of cells that have died due to other causes (in our case due to TNF injections).

So far, the afformentioned TNF parameter values have been chosen at random from their allowed range, in order to produce an adequate number of simulations and perform sufficient exploration of the parameter space. Currently, new methods of parameter selection have been employed, which aim in discovering the areas of the parameter space that lead to interesting simulations. These methods will be further explained in subsection 6.2.

[11] https://github.com/gletort/PhysiBoSS

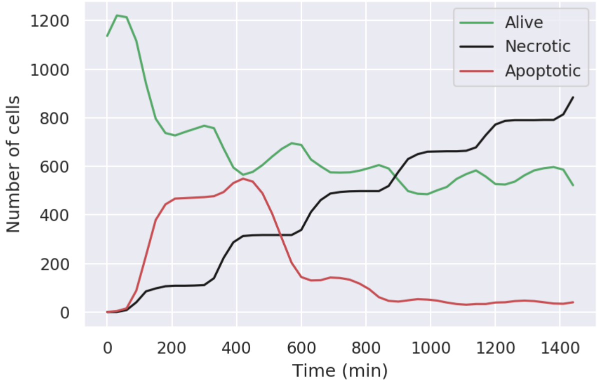
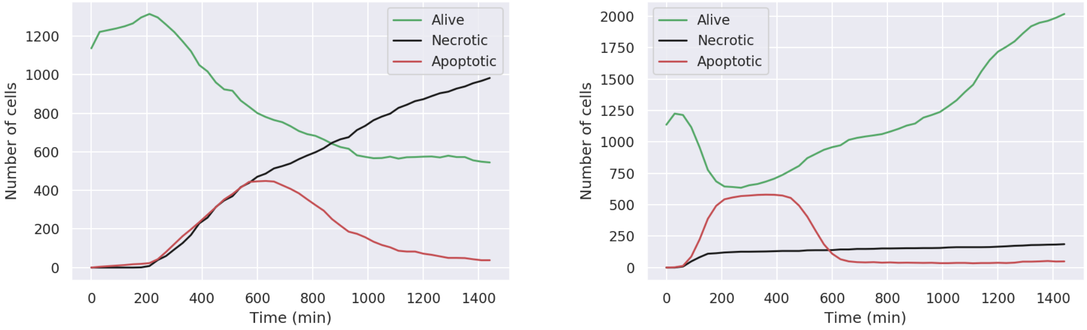


Figure 20: A PhysiBoSS simulation from the biological use case dataset.

As we mentioned before, the simulations can be used to study the response of cell populations to TNF treatment. Therefore, we roughly characterize a simulation as interesting if there is a significant response to the treatment. In practice, in order to classify the simulations in our dataset, we have used a 1-NN classifier, which uses some known cases to determine if a simulation is interesting or not. Figure 21a shows the course of a simulation that has been classified as interesting. In this simulation the number of necrotic cells significantly increases over time, therefore we could say that the population responds to the treatment. On the other hand, in Figure 21b the number of alive cells is constantly increasing during the simulation time, thus we can surmise that the treatment selected will not lead to positive results. The generated dataset contains around 900 simulations, out of which around 150 have been classified as interesting and 750 as not interesting.



(a) An interesting simulation.

(b) A not-interesting simulation.

Figure 21: Examples of interesting and not interesting simulations.

6.2 Methods for Exploring the Parameter Space of Biological Simulations

As mentioned, the parameter space defined by the allowed ranges of the parameters of interest (*time_add_TNF, time_remove_TNF, duration_add_TNF and TNF_concentration*) is too large to be sampled at random. Therefore, we are using some additional methods, in order to explore the parameter space more efficiently and define the boundaries between areas that lead to interesting and not-interesting simulations, following the idea of [112].

6.2.1 Genetic Algorithm

For performing better exploration of the parameter space, we have used a genetic algorithm. The genetic algorithm is used with the goal of converging to interesting areas of the parameter space and, thus, revealing interesting combinations of the parameters. For the implementation of the genetic algorithm, we used the DEAP framework [12], as follows:

Individuals: Each individual is a vector of 4 elements and each one of the elements represents a value for the corresponding parameter. In other words, an individual is a combination of values for the parameters mentioned, that will be given as input to PhysiBoSS.

Population: The population consists of 20 individuals. This number has been selected to be relatively small because each simulation requires substantial amount of time.

Crossover: The crossover probability has been selected equal to 0.7. The crossover function picks at random an index and swaps the corresponding values between the two individuals, after these values have been multiplied by a factor close to 1. The reason behind this multiplication is that individuals consist of only four values. Therefore, we are trying to avoid replication of individuals. In addition, this function swaps each other pair of corresponding values of two individuals with probability 0.5. Finally, since the parameter values have to respect some additional rules (e.g. $time\_add\_TNF \geq duration\_add\_TNF$), if after swapping the values these conditions are not met, we pick random acceptable values for each parameter, in order to return two new, acceptable individuals.

Mutation: With probability 0.5 the algorithm replaces each parameter with a random value that is uniformely selected from an acceptable range. Again, the acceptable ranges depend on the values selected for the other parameters. Moreover, the function checks that at least one value of the individuals will change.

Objective function: The algorithm evaluates an individual by running a PhysiBoSS simulation with the parameters that it indicates. For now, the objective function has been selected to be the difference between the final number of necrotic and alive cells during the corresponding simulation, as proposed in [112] for a similar setup. Of course, the objective function could be altered to take into account more values of the simulation.

Generations: Currently, the algorithm uses 20 generations. Again, because of the long simulation time, we selected a relatively small number of generations.

6.2.2 Random Forest

While genetic algorithms can be very efficient in discovering optimal solutions in large spaces, they are not sufficient for estimating the structure of complex parameter spaces. On the other hand, random forests [68] can provide useful insight about the parameter space. In order to avoid the tendency of decison trees to overfit, random forests construct several decison trees and compute the mode of the resulted classes for each input sample.

In our setup, random forests can be used in an active learning way, in order to better determine the boundaries of the

[12] https://github.com/DEAP/deap

different areas of the parameter space. Iteratively, such an algorithm fits a random forest and then selects samples close to the classification boundaries, evaluates them and adds them in the training dataset. At each step, this procedure produces better classification boundaries and improves the estimation of the parameter space. Note that the level of space exploration can be increased by introducing randomness to the point selection procedure. For the implementation of random forests, we have used the scikit-learn library in Python [118]. We will now provide a description of the algorithm implemented:

The algorithm requires a training dataset, which contains examples of interesting and not-interesting simulations. For this purpose we have used the dataset that we have created so far, from our random sampling method and classified via 1-NN and known cases. This dataset is used to construct an initial random forest.

During each iteration:

- The algorithm constructs a random forest based on the examples that are at that point in the dataset.
- Creates a fine grid in the parameter space and classifies its points using the random forest that has been constructed.
- Computes the class uncertainty of these points and keeps the points with the highest uncertainty value, as candidate points.
- If the number of candidate points is bigger than a threshold (k), the algorithm clusters the candidate points into k groups, based on their position in the grid. (We do that in order to avoid neglecting some areas of the parameter space). Then, it selects one point from each cluster at random
- Evaluates the selected points by running a PhysiBoss simulation and using a 1-NN classifier.
- These points are added to the dataset.

6.3 Discretization

Having produced an adequate number of biological simulations, the goal is to use these examples to learn how to forecast whether a running simulation will eventually become interesting or not. Moreover, from a biological perspective it is usefull to extract some features or patterns that can facilitate the forecasting procedure, while being in some easily interpretable form that will allow researchers to draw conclusions about the results. A promising way to achieve both of these goals is to transform the produced time-series into strings, and learn regular expressions patterns that can be used by the complex event forecasting module (Section 7).

Over the years, many time-series representation methods have been introduced, including traditional DWT [41] and DFT [34] based methods. We employed the Symbolic Aggregate Approximation (SAX) method [94] to discretize the time-series and transform them into symbolic sequences, since it produces an appropriate representation for learning regular expressions and led to very small information loss in our dataset. The method consists of two steps, one for dimensionality reduction and another one for time-series discretization.

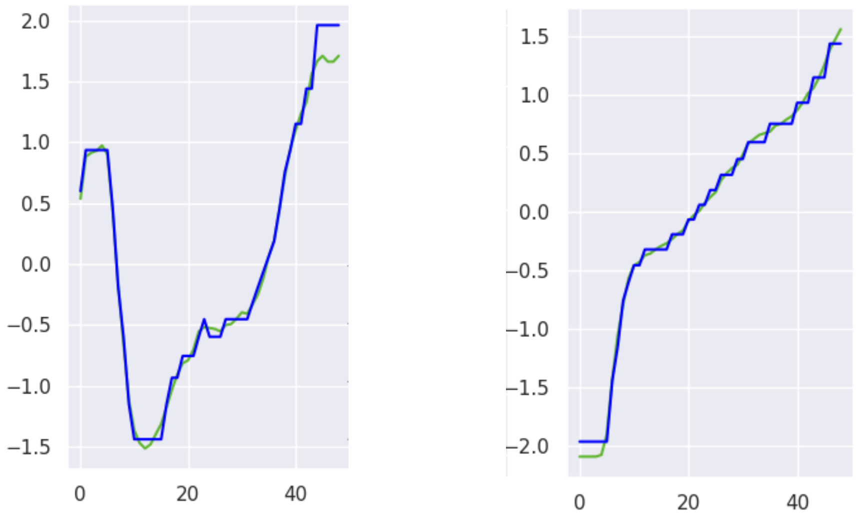
More formally, SAX allows a time-series of arbitrary length n, to be reduced to a string of arbitrary length w (traditionally $w < n$), using symbols from an alphabet that has an arbitrary size a. To that end, SAX employs the Piecewise Aggregate Approximation (PAA) method [82] as an intermediate step for dimensionality reduction, by dividing the time-series into w equal sized frames and computing the mean value of the series in each frame:

More formally, let C be a time-series of length n. Then, C can be represented in a w-dimensional space by a vector $\overline{C} = \overline{c_1}, ..., \overline{c_w}$, where:

$$\overline{c_i} = \frac{w}{n} \sum_{j=\frac{n}{w}(i-1)+1}^{\frac{n}{w}i} c_j$$

An important note for using this method is that the input time-series should be z-normalized (with a mean of 0 and a standard deviation of 1). This step is neccessary when comparing time-series with different offsets and amplitudes, because otherwise we run the risk of huge information loss.

Having transformed the time-series into the PAA representation, the SAX method dictates applying the following transformation to obtain a discrete representation. Note that it is desirable to have a discretization technique that produces equiprobable symbols, that is symbols that are expected to appear in the discretized dataset with almost equal frequency. Since normalized time-series have a highly Gaussian distribution, according to the SAX method we determine the breakpoints that produce a equal sized areas under a Gaussian curve. Having determined the breakpoints that divide the range of values for the time-series into a areas, we can match each of the areas with a discrete symbol and extract the symbolized version of the time-series.



(a) Alive cells line.

(b) Necrotic cells line.

Figure 22: Original vs discretized time-series for the alive and necrotic cells.

Figure 22 shows an application of the SAX method to a time-series produced by PhysiBoSS. For this representation we have used $a = 20$ discrete symbols and $w = n$, because the dimensionality of our time-series was quite small. Figure 22a depicts the course of alive cells during the simulation. The green line contains the normalized original time-series points and the blue line is their corresponding discrete form. Similarly, figure 22b corresponds to the number of necrotic cells during the simulation.

We measured the information loss from applying the SAX discretization tecnhique in our dataset, using a 1-NN classifier with several distance measures, including Euclidian, Hamming and Dynamic Time Warping distances. We considered as a misclassification the discrepancy between the original (continuous time-series) classification result and the result when we reclassified our dataset using symbolic representations only. In all cases, the classification error was bellow 1%.

Additionally, we performed a 10-fold cross validation with a 1-NN classifier and Dynamic Time Warping distance, which was the distance measure that led to the smallest classification error during our reclassification tests. Again, we measured a classification error of less than 1%.

7 Complex Event Forecasting

7.1 Introduction

The avalanche of real-time data in the last decade has sparked an interest in fields focused on processing high-velocity data streams. Complex Event Recognition (CER) is one of these fields which have enjoyed increased popularity [37, 62]. 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 learning and forecasting event patterns [62], despite the fact that forecasting has attracted considerable attention in various related research areas, such as time-series forecasting [105], sequence prediction [23, 26, 127, 33, 149], temporal mining [145, 89, 153, 31] and process mining [99]. The need for Complex Event Forecasting (CEF) has been acknowledged though, as evidenced by several conceptual proposals [55, 32].

In this section, we present a complete and 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 and then constructs a probabilistic model for it 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 [39] to encode a pattern and that of prediction suffix trees [127, 126] to learn a probabilistic model for the pattern. We formally show how symbolic automata can be combined with prediction suffix trees to perform event forecasting. 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 methods for CEF, based on low order models or non-Markov models (like Hidden Markov Models).

7.1.1 Running example

We now present the general idea behind CER systems, along with an example that we will use throughout the rest of the section to make our presentation more accessible. The input to a CER system consists of two main components: a stream of events, also called simple events (SEs); a set of patterns that define relations among the SEs 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 [97, 51, 67].

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 9 shows a possible stream of AIS messages, where we only include the *speed* attribute and the *timestamp* is shown as a sequence of increasing indexes. 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 SEs to detect CEs.

Table 9: Example stream.

type	fishing	fishing	fishing	under way	under way	under way	...
vessel id	78986	78986	78986	78986	78986	78986	...
speed	2	1	3	22	19	27	...
timestamp	1	2	3	4	5	6	...

7.1.2 Structure of the Section

We start by presenting in Section 7.2 the relevant literature on CEF. Since work on CEF has been limited thus far, we also briefly mention forecasting ideas from some other related fields that can provide inspiration to CEF. Subsequently, in Section 7.3 we discuss the formalism of symbolic automata and how it can be adapted to perform recognition on real-time event streams. Section 7.4 shows how we can create a probabilistic model for a symbolic automaton by using prediction suffix trees. We then demonstrate the efficacy of our framework in Section 7.5, by showing experimental results on two datasets. We conclude with Section 7.6, discussing some possible directions for future work. The section assumes a basic familiarity with automata theory, logic and Markov chains.

7.2 Related Work

There are multiple ways to define the task of forecasting over time-evolving data streams. Before proceeding with the presentation of previous work on forecasting, we first begin with a terminological clarification. It is often the case that the terms “forecasting” and “prediction” are used interchangeably as equivalent terms. For reasons of clarity, we opt for the term of “forecasting” to describe our work, since there does exist a conceptual difference between forecasting and prediction, as the latter term is understood in machine learning. In machine learning, the goal is to “predict” the output of a function on previously unseen input data. The input data need not necessarily have a temporal dimension and the term “prediction” refers to the output of the learned function on a new data point. For this reason we avoid using the term “prediction”. Instead, we choose the term “forecasting” to define the task of predicting the temporally future output of some function or the occurrence of an event. Time is thus a crucial component for forecasting. Moreover, an important challenge stems from the fact that, from the (current) timepoint where a forecast is produced until the (future) timepoint for which we try to make a forecast, no data is available. A forecasting system must (implicitly or explicitly) fill in this data gap in order to produce a forecast, whereas in a typical machine learning task a prediction can be made using all available data that refer to the target point of the prediction.

In what follows, we present previous work on forecasting, as defined above, in order of increasing relevance to CER. Since work on CEF has been limited thus far, we start by briefly mentioning some forecasting ideas from other fields and discuss how CEF differs from these research areas.

Time-series forecasting. Time-series forecasting is an area with some similarities to CEF, with a significant history of contributions [105]. However, it is not possible to directly apply techniques from time-series forecasting to CEF. Time-series forecasting typically focuses on streams of (mostly) real-valued variables and the goal is to forecast relatively simple patterns. On the contrary, in CEF we are also interested in categorical values, related through complex patterns and involving multiple variables. Another limitation of time-series forecasting methods is that they often do not target a pattern, but try to forecast the next value(s) from the input stream/series. In CER, the equivalent task would be to forecast the next input event(s) (SEs). This task in itself is not very useful for CER though, since most SEs are often "irrelevant" and do not contribute to the detection of CEs (see the discussion on selection policies in Section 7.3). CEs are more like "anomalies" and their number is typically orders of magnitude lower than the number of SEs. One could possibly try to leverage techniques from SE forecasting to perform CE forecasting. At every timepoint, we could try to estimate the most probable sequence of future SEs, then perform recognition on this future stream of SEs and check whether any future CEs are detected. We have experimentally observed that such an approach yields extremely 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. Because of this serious under-performance of this method, we do not present detailed experimental results. We will, however, present results showing that, if a method has a higher accuracy in terms of SE forecasting, this does not necessarily imply that it will also have better results in terms of CE forecasting.

Sequence prediction (compression). Another related field is that of prediction of discrete sequences over finite alphabets and is closely related to the field of compression, as any compression algorithm can be used for prediction and vice versa. The relevant literature is extensive. Here we focus on a sub-field with high importance for our work, as we have borrowed ideas from it. It is the field of sequence prediction via variable-order Markov models [23, 26, 127, 126, 33, 149]. As the name suggests, the goal is to perform prediction by using a high-order Markov model. Doing so in a straightforward manner, by constructing a high-order Markov chain with all its possible states, is prohibitively expensive due to the combinatorial explosion of the number of states. Variable-order Markov models address this issue by retaining only those states that are "informative" enough. In Section 7.4.2, we discuss the relevant literature in more details. The main limitation of previous methods for sequence prediction is that they focus exclusively on next symbol prediction, i.e., they try to forecast the next symbol(s) in a stream/string of discrete symbols. As already discussed, this is a serious limitation for CER. An additional limitation is that they work on single-variable discrete sequences of symbols, whereas CER systems consume streams of events, i.e., streams of tuples with multiple variables, both numerical and categorical. Notwithstanding these limitations, we show that variable-order models can be combined with symbolic automata in order to overcome their restrictions and perform CEF.

Temporal mining. Forecasting methods have also appeared in the field of temporal pattern mining [145, 89, 153, 31]. A common assumption in these methods is that patterns are usually defined either as association rules [5] or as frequent episodes [98]. In [145] the goal is to identify sets of event types that frequently precede a rare, target event within a temporal window, using a framework similar to that of association rule mining. In [89], a forecasting model is presented, based on a combination of standard frequent episode discovery algorithms, Hidden Markov Models and mixture models. The goal is to calculate the probability of the immediately next event in the stream. In [153] a method is presented for batch, online mining of sequential patterns. The learned patterns are used to test whether a prefix matches the last events seen in the stream and therefore make a forecast. The method proposed in [31] starts with a given episode rule (as a Directed Acyclic Graph) and detects the minimal occurrences of the antecedent of a rule defining a complex event, i.e., those "clusters" of antecedent events that are closer together in time. From the perspective of CER, the disadvantage of these methods is that they usually target simple patterns, defined either as strictly sequential or as sets of input events. Moreover, the input stream is composed of symbols from a finite alphabet, as is the case with the compression methods mentioned previously.

Process mining. Compared to the previous categories for forecasting, the field of process mining is more closely related to CER [142]. Processes are typically defined as transition systems (e.g., automata or Petri nets) and are used to monitor a system, e.g., for conformance testing. Process mining attempts to automatically learn a process from a set of traces, i.e., a set of activity logs. Since 2010, a significant body of work has appeared, targeting process prediction, where the goal is to forecast if and when a process is expected to be completed (for surveys, see [99, 53]). According to [99], until 2018, 39 papers in total have been published dealing with process prediction. At a first glance, process prediction seems very similar to CEF. At a closer look though, some important differences emerge. An important difference is that processes are usually given directly as transition systems, whereas CER patterns are defined in a declarative manner. The transition systems defining processes are usually composed of long sequences of events. On the other hand, CER patterns are shorter, may involve Kleene-star, iteration operators (usually not present in processes) and may even be instantaneous. Consider, for example, a pattern for our running example, trying to detect speed violations by simply checking whether a vessel's speed exceeds some threshold. This pattern could be expanded to detect more violations by adding more disjuncts, e.g., for checking whether a vessel is sailing within a restricted area, all of which might be instantaneous. A CEF system cannot always rely on the memory implicitly encoded in a transition system and has to be able to learn the sequences of events that lead to a (possibly instantaneous) CE. Another important difference is that process prediction focuses on traces, which are complete, full matches, whereas CER focuses on continuously evolving streams which may contain many irrelevant events. A learning method has to take into account the presence of these irrelevant events. In addition to that, since CEs are rare events, the datasets are highly imbalanced, with the vast majority of "labels" being negative (i.e., most forecasts should report that no CE is expected to occur, with very few being positive). A CEF has to strike a fine balance between the positive and negative forecasts it produces in order to avoid drowning the positives in the flood of all the negatives and, at the same time, avoid over-producing positives that lead to false alarms. This is also an important issue for process prediction, but becomes critical for a CEF system, due to the imbalanced nature of the datasets. In Section 7.5, we have included one method from the field of process prediction to our empirical evaluation. This "unfair" comparison (in the sense that it is applied on datasets more suitable for CER) shows that this method consistently under-performs with respect to other methods from the field of CEF.

Complex event forecasting. Contrary to process prediction, forecasting has not received much attention in the field of CER, although some conceptual proposals have acknowledged the need for CEF [55, 50, 32]. To the best of our knowledge, the first concrete attempt at CEF was presented in [110]. 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 7.4.2, in the worst case, such an approach assumes that all SEs are independent and is thus unable to encode higher order dependencies. Another example of event forecasting is presented in [6]. 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, 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 SE forecasting one. The idea is put forward that these future SEs may be used by a CER engine to detect future CEs. As we have already mentioned though, in our experiments, this idea has yielded poor results. In [113], 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 SEs. 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 7.4.2 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 [8, 9]. The main difference of these methods compared to [110] 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 [8] 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 [9], 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.

7.3 Complex Event Recognition with Symbolic Automata

Before presenting our proposed approach for CEF, we begin by first presenting a formal framework for CER. For surveys of CER, please see [37, 11, 62]. 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 be a CER language and what their semantics should be. In our work, we follow [62] and [65] which have established some core operators that are most often used. In a spirit similar to [65], 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 [39, 38, 144]. 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 [65] for a similar line of work, based on symbolic transducers).

7.3.1 Symbolic Expressions and Automata

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. 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 5 (Effective boolean algebra [39])**.** *An effective Boolean algebra is a tuple (*$\mathscr{D}$*,* Ψ*,* $[\![\_]\!]$*,* $\bot$*,* $\top$*,* $\vee$*,* $\wedge$*,* $\neg$*) where*

- $\mathscr{D}$ *is a set of domain elements;*
- Ψ *is a set of predicates closed under the Boolean connectives;*
- $\bot, \top \in \Psi$ *;*
- *the component* $[\![\_]\!] : \Psi \rightarrow 2^{\mathscr{D}}$ *is a denotation function such that*
  - $[\![\bot]\!] = \emptyset$
  - $[\![\top]\!] = \mathscr{D}$
  - *and* $\forall \phi, \psi \in \Psi$*:*
    - $[\![\phi \vee \psi]\!] = [\![\phi]\!] \cup [\![\psi]\!]$
    - $[\![\phi \wedge \psi]\!] = [\![\phi]\!] \cap [\![\psi]\!]$

- $[\![\neg\phi]\!] = \mathscr{D} \setminus [\![\phi]\!]$

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 $\mathscr{D}$ are called *characters* and finite sequences of characters are called *strings*. A set of strings $\mathscr{L}$ constructed from elements of $\mathscr{D}$ ($\mathscr{L} \subseteq \mathscr{D}^*$, where * denotes Kleene-star) is called a language over $\mathscr{D}$.

As with classical regular expressions [69], we can use symbolic regular expressions to represent a class of languages over $\mathscr{D}$.

Definition 6 (Symbolic regular expression)**.** *A symbolic regular expression (SRE) over an effective Boolean algebra ($\mathscr{D}$, Ψ, $[\![\_]\!]$, $\bot$, $\top$, $\vee$, $\wedge$, $\neg$) is recursively defined as follows:*

- *The constants ε and $\emptyset$ are symbolic regular expressions with $\mathscr{L}(\varepsilon) = \{\varepsilon\}$ and $\mathscr{L}(\emptyset) = \{\emptyset\}$;*

- *If $\psi \in \Psi$, then $R := \psi$ is a symbolic regular expression, with $\mathscr{L}(\psi) = [\![\psi]\!]$, i.e., the language of ψ is the subset of $\mathscr{D}$ for which ψ evaluates to TRUE;*

- *If R_1 and R_2 are symbolic regular expressions, then $R := R_1 + R_2$ is also a symbolic regular expression, with $\mathscr{L}(R) = \mathscr{L}(R_1) \cup \mathscr{L}(R_2)$;*

- *If R_1 and R_2 are symbolic regular expressions, then $R := R_1 \cdot R_2$ is also a symbolic regular expression, with $\mathscr{L}(R) = \mathscr{L}(R_1) \cdot \mathscr{L}(R_2)$, where $\cdot$ denotes concatenation. $\mathscr{L}(R)$ is then the set of all strings constructed from concatenating each element of $\mathscr{L}(R_1)$ with each element of $\mathscr{L}(R_2)$;*

- *If R is a symbolic regular expression, then $R' := R^*$ is a symbolic regular expression, with $L(R^*) = (L(R))^*$, where $L^* = \bigcup\limits_{i \geq 0} L^i$ and L^i is the concatenation of L with itself i times.*

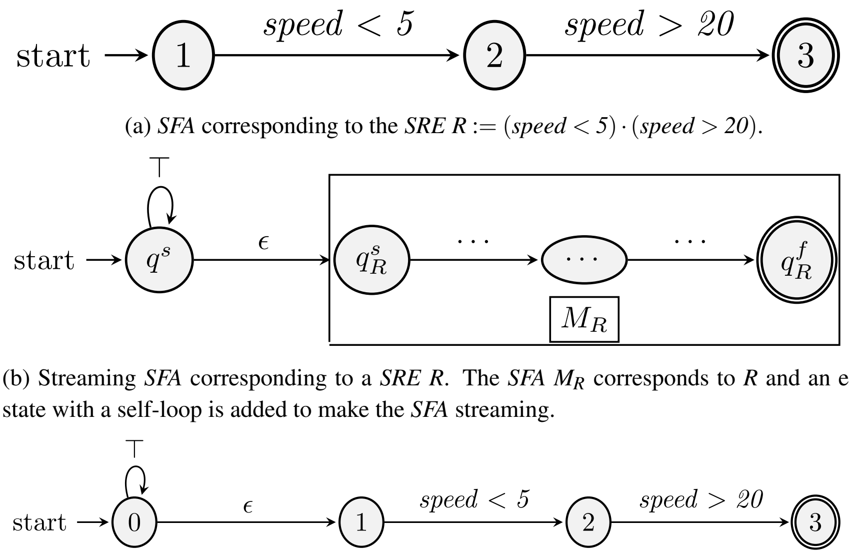
As an example, if we want to detect instances of a vessel accelerating suddenly, we could write the expression $R := (speed < 5) \cdot (speed > 20)$. The third and fourth events of the stream of Table 9 would then belong to the language of R.

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

Definition 7 (Symbolic automaton [39])**.** *A symbolic finite automaton (SFA) is a tuple M =($\mathscr{A}$, Q, q^s, F, Δ), where*

- *$\mathscr{A}$ is an effective Boolean algebra;*

- *Q is a finite set of states;*



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

(b) Streaming *SFA* corresponding to a *SRE* R. The *SFA* M_R corresponds to R and an extra state with a self-loop is added to make the *SFA* streaming.

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

Figure 23: Examples of symbolic automata and streaming symbolic automata.

- *$q^s \in Q$ is the initial state;*
- *$Q^f \subseteq Q$ is the set of final states;*
- *$\Delta \subseteq Q \times \Psi_{\mathscr{A}} \times Q$ is a finite set of transitions.*

A string $w = a_1 a_2 a \cdots a_k$ is accepted by a *SFA* M iff, for $1 \leq i \leq k$, there exist transitions $q_{i-1} \overset{a_i}{\to} q_i$ 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 $\mathscr{L}(M)$ [39]. Figure 23a 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 2. *For every symbolic regular expression R there exists a symbolic finite automaton M such that $\mathscr{L}(R) = \mathscr{L}(M)$.*

Proof. The proof is essentially the same as that for classical expressions and automata [69]. It is a constructive proof starting from the base case of an expression that is a single predicate (instead of a symbol, as in classical expressions) and then proceeds in a manner identical to that of the classical case. □

7.3.2 Streaming Expressions and Automata

Our discussion thus far has focused on how *SRE* and *SFA* can be applied to strings that are known in their totality before recognition. 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. 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 terminology we can speak of tuples from relations of a database schema [65]. These tuples constitute the set of domain elements $\mathscr{D}$. A stream S then has the form of an infinite sequence $S = t_1, t_2, \cdots$, where each t_i is a tuple ($t_i \in \mathscr{D}$). Our goal is to report the indexes 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 $S_{m..k} \in \mathscr{L}(R)$. In order to detect CEs of a *SRE* R on a stream, we use a streaming version of *SRE* and *SFA*.

Definition 8 (Streaming SRE and SFA)**.** *If R is a SRE, then $R_s = \top^* \cdot R$ is called the streaming SRE (sSRE) corresponding to R. A SFA M_{R_s} constructed from R_s is called a 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 $m, 1 \leq m \leq k$. 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 23b shows how a *sSFA* can be constructed from a *SRE* R and Figure 23c shows a concrete *sSFA* for $R := (speed < 5) \cdot (speed > 20)$.

Proposition 3. *If $S = t_1, t_2, \cdots$ is a stream of domain elements from an effective Boolean algebra ($\mathscr{D}$, Ψ, $[\![\_]\!]$, $\bot$, $\top$, $\vee$, $\wedge$, $\neg$) ($t_i \in \mathscr{D}$) and R is a symbolic regular expression over the same algebra, then, for every $S_{m..k}$, $S_{m..k} \in \mathscr{L}(R)$ iff $S_{1..k} \in \mathscr{L}(R_s)$ (and $S_{1..k} \in \mathscr{L}(M_{R_s})$).*

Proof. The proof is trivial. First, assume that $S_{m..k} \in \mathscr{L}(R)$ for some $m, 1 \leq m \leq k$ (we set $S_{1..0} = \varepsilon$). Then, for $S_{1..k} = S_{1..(m-1)} \cdot S_{m..k}$, $S_{1..(m-1)} \in \mathscr{L}(\top^*)$, since $\top^*$ accepts every string (sub-stream), including ε. We know that $S_{m..k} \in \mathscr{L}(R)$, thus $S_{1..k} \in \mathscr{L}(\top^*) \cdot \mathscr{L}(R) = \mathscr{L}(\top^* \cdot R) = \mathscr{L}(R_s)$. Conversely, assume that $S_{1..k} \in \mathscr{L}(R_s)$. Therefore, $S_{1..k} \in \mathscr{L}(\top^* \cdot R) = \mathscr{L}(\top^*) \cdot \mathscr{L}(R)$. As a result, $S_{1..k}$ may be split as $S_{1..k} = S_{1..(m-1)} \cdot S_{m..k}$ such that $S_{1..(m-1)} \in \mathscr{L}(\top^*)$ and $S_{m..k} \in \mathscr{L}(R)$. Note that $S_{1..(m-1)} = \varepsilon$ is also possible, in which case the result still holds, since $\varepsilon \in \mathscr{L}(\top^*)$. □

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 9, $S_{1..4}$ would belong to the language of $\mathscr{L}(R)$ and $S_{3..4}$ to the language of $\mathscr{L}(R')$. 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 23b shows how a *sSFA* can be constructed from a *SRE* R and Figure 23c shows a concrete *sSFA* for $R := (speed < 5) \cdot (speed > 20)$.

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

Definition 9 (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 (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:*

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

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

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

It is easy to see that a run ρ of a *sSFA* M_{R_s} over a stream $S_{1..k}$ is accepting iff $S_{1..k} \in \mathscr{L}(R_s)$, 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.

We conclude this section with some remarks about the expressive power of *SRE* and *SFA* and how it meets the requirements of a CER system. As discussed in [62, 65], besides the three operators of regular expressions that we have presented and implemented, there exist some extra operators which should be supported by a CER system. *Negation* is one them. If we use ! to denote the negation operator, then $R' :=!R$ defines a language which is the complement of the language of R. Since *SFA* are closed under complement [39], negation is an operator that can be supported by our framework. The same is true for the operator of *conjunction*. If we use $\wedge$ to denote conjunction, then $R := R_1 \wedge R_2$ is an expression whose language consists of concatenated elements of $\mathscr{L}(R_1)$ and $\mathscr{L}(R_2)$, regardless of their order, i.e., $\mathscr{L}(R) = \mathscr{L}(R_1) \cdot \mathscr{L}(R_2) \cup \mathscr{L}(R_2) \cdot \mathscr{L}(R_1)$. This operator can thus be equivalently expressed using the already available operators of concatenation ($\cdot$) and disjunction ($+$). Another important notion in CER is that of *selection policies*. An expression like $R := R_1 \cdot R_2$ typically implies that an instance of R_2 must immediately follow an instance of R_1. As a result, for the stream of Table 9 and $R := (speed < 5) \cdot (speed > 20)$, only one match will be detected at $timestamp = 4$. With selection policies, we can relax the requirement for contiguous instances. For example, with the so-called skip-till-any-match policy, any number of events are allowed to occur between R_1 and R_2. If we apply this policy on $R := (speed < 5) \cdot (speed > 20)$, we would detect six CEs, since the first three events of Table 9 can be matched with the two events at $timestamp = 4$ and at $timestamp = 6$, if we ignore all intermediate events. Selection policies can also be accommodated by our framework. For a proof, using symbolic transducers, see [65]. Notice, for example, that an expression $R := R_1 \cdot R_2$ can be evaluated with skip-till-any-match by being rewritten as $R' := R_1 \cdot \top^* \cdot R_2$, so that any number of events may occur between R_1 and R_2. Support for *hierarchies*, i.e., the ability to define patterns in terms of other patterns, is yet another important feature in many CER systems. Since *SRE* and *SFA* are compositional by nature, hierarchies are supported by default in our framework. Although we do not treat these operators and functionalities explicitly, their incorporation is possible within the expressive limits of *SRE* and *SFA* and the results that we present in the next sections would still hold.

However, there is one functionality that we do not currently support and whose incorporation would require us to move to a more advanced automaton model. This is the functionality of applying n-ary (with $n > 1$) predicates to two or more sub-expressions of an expression, instead of only unary predicates, as is allowed in symbolic automata. As an example, consider the pattern $R := x \cdot y$ WHERE $y.speed > x.speed$, detecting an increase in the speed of a vessel, where we now need to use the variables x and y. Such patterns cannot be captured with *SFA* since they would require a memory structure to store some of the past events of a stream, as is possible with extended symbolic automata [38]. We intend to present in future work an automaton model which can support patterns with memory, suitable for CER. Finally, 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 functonality would require the use of symbolic transducers in order to mark the input events according to whether they belong to a match or not [65]. We do not pay particular attention to this requirement in our work, since our goal is to forecast the occurrence of CEs, without needing to report the input events that lead to a CE occurrence.

7.4 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 converted to a Markov chain that encodes dependencies among the events in an input stream. The Markov chain is learned from a portion of the input stream which acts as a training dataset and it is then used to derive probabilistic 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 is how to build a Markov chain which retains only meaningful long-term dependencies.

7.4.1 Preliminary definitions and results

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. It is important to 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 11 (Deterministic SFA [39])**.** *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 [39]. The determinization process first needs to create the *minterms* of the predicates of a *SFA M* (the set of maximal satisfiable Boolean combinations of such predicates), denoted by $Minterms(Predicates(M))$, and then use these minterms as guards for the *DSFA* [39].

Note that there are then two factors that can lead to a combinatorial explosion of the number of states of the resulting *DSFA*: first, the fact that the powerset of the states of the original *SFA* must be constructed (similarly to classical automata); second, the fact that the number of minterms (and, thus, outgoing transitions from each *DSFA* state) is an exponential function of the number of the original *SFA* predicates. In order to mitigate this doubly exponential cost, we follow two simple optimization techniques. As is typically done with classical automata as well, instead of constructing the powerset of states of the *SFA* and then adding transitions, we construct the states of the *DSFA* incrementally, starting from its initial state, without adding states that will be inaccessible in the final *DSFA*. We can also reduce the number of minterms by taking advantage of some previous knowledge about some of the predicates that we might have. In cases where we know that some of the predicates are mutually exclusive, i.e., at most one of them can evaluate to TRUE, then we can both discard some minterms and simplify some others. This is a very common case in CER where we typically know that each event may have only one event type. For example, if we have two predicates, $\psi_A := speed < 5$

Table 10: The set of simplified minterms for the predicates $\psi_A := speed < 5$ and $\psi_B := speed > 20$.

Original	Simplified	Reason
$\psi_A \wedge \psi_B$	discard	unsatisfiable
$\psi_A \wedge \neg\psi_B$	ψ_A	$\psi_A \vDash \neg\psi_B$
$\neg\psi_A \wedge \psi_B$	ψ_B	$\psi_B \vDash \neg\psi_B$
$\neg\psi_A \wedge \neg\psi_B$	$\neg\psi_A \wedge \neg\psi_B$	for events whose speed is between 5 and 20

and $\psi_B := speed > 20$, then we also know that ψ_A and ψ_B are mutually exclusive. As a result, we can simplify the minterms, as shown in Table 10.

Before moving to the discussion about how a *DSFA* can be converted to a Markov chain, we provide a lemma that will help in simplifying our notation. First note that $Minterms(Predicates(M))$ induces a finite set of equivalence classes on the (possibly infinite) set of domain elements of M [39]. For example, if $Predicates(M) = \{\psi_1, \psi_2\}$, then $Minterms(Predicates(M)) = \{\psi_1 \wedge \psi_2, \psi_1 \wedge \neg\psi_2, \neg\psi_1 \wedge \psi_2, \neg\psi_1 \wedge \neg\psi_2\}$ 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 12 (Stream induced by the minterms of a *DSFA*)**.** *If S is a stream from the domain elements of the algebra of a DSFA M and $T = Minterms(Predicates(M))$, then we say that S' is the stream induced by applying T on S, i.e., it is the equivalence class of S induced by T.*

We can now prove a useful lemma, which states that for every *DSFA* there exists an "equivalent" classical deterministic automaton.

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

Proof. From an algebraic point of view, the set $T = Minterms(Predicates(M))$ may be treated as a generator of the monoid T^*, with concatenation as the operation. If the cardinality of T 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) $\phi : T^* \rightarrow \Sigma^*$ that maps each minterm to exactly one, unique a_i. A classical deterministic automaton 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 [129]. Or, equivalently, the language of M_c is the set of strings induced by applying $T = Minterms(Predicates(M_s))$ to $\mathcal{L}(M_s)$. □

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

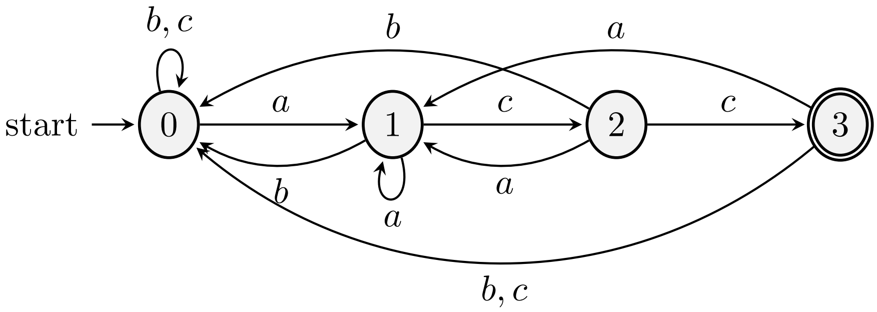


Figure 24: A classical automaton for the expression $R := a \cdot c \cdot$. State 1 can always remember the last symbol seen, since it can be reached only with a, whereas state 0 can be reached either with a b or a c.

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*.

7.4.2 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. A straightforward approach would be to simply 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 from each state to every other target states and use these counts to calculate the transition probabilities. This is the approach followed in [110]. 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 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. Due to this fact, 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 24, 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 24, which we can reach by seeing either a b or a c symbol. 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 [9, 8], 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 [23, 26, 127, 126, 33, 149]. 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. Additionally, even when the accuracy of a VMM-based method is comparable to that of a full-order Markov model, we can show that VMM-based methods typically have better training times and throughput performance (when used in a streaming setting, as in our case). An alternative would be to use hidden Markov models (HMMs) [121], 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 [23, 3]. 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 [23]. 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, such as Prediction by Partial Matching (PPM) [33], Context Tree Weighting (CTW) [149] and Prediction Suffix Trees (PST) [127, 126]. Event the famous LZ-78 compression algorithm [156] and its variants may be viewed as belonging to the class of VMMs. For a nice comparative study, see [23]. Compression is the main goal of these methods, but they can also be used for prediction, since compression and prediction are the two sides of the same coin. Every compression algorithm can be used as a prediction algorithm as well and vice versa. The general idea behind all these approaches 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 [23]. The mean log-loss $(-E_P\{log\hat{P}(S_{1..k})\})$ is minimized if the derived predictor $\hat{P}$ is indeed the actual distribution P of the source emitting sequences.

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. 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_{max}, 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. The learning process can be broken down into three main components [23]: counting, where we count how many times a symbol σ appears after a context s and derive conditional probabilities; smoothing, where we take into account symbols that belong to the alphabet but never appear in the training dataset; and modeling itself, which may follow various directions, e.g., either pre-determining m_{max} or let it be decided by the data.

7.4.3 Prediction Suffix Trees

We use Prediction Suffix Trees (*PST*), as described in [127, 126], 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 *sSRE* R, which then allows us to infer when the *DSFA* will reach one of its final states, taking into account at the same time the statistical properties of the stream, as encoded into the *PSA*.

The formal definition of a PST is the following:

Definition 13 (Prediction Suffix Tree [127])**.** *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 25b shows an example of a *PST*. Note that a *PST* whose leaves are all of equal depth m corresponds to a full-order Markov model of order m, as its paths from the root to the leaves correspond to every possible context of length m. The 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 [127] 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 that must hold:

- 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, $\frac{\hat{P}(\sigma|s)}{\hat{P}(\sigma|suffix(s))} > \theta_2$ 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 $suffix(s)$.

The thresholds θ_1 and θ_2 depend, among others, on an approximation parameter ε, measuring how close we want the estimated *PSA* $\hat{M}$ to be compared to M, on n, denoting the maximum number of states that we allow $\hat{M}$ to have and on m_{max}, denoting the maximum order/length of the dependencies we want to capture. For more details, see [127].

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

Definition 14 (Probabilistic Suffix Automaton [127])**.** *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.*

Figure 25c shows an example of a *PSA*. Note that a *PSA* does not act as an acceptor (there are no final states), but can act as a generator of strings. It can use π, its initial distribution on states, to select an initial state and generate its label as a first string and then continuously use γ to generate a symbol, move to a next state and repeat the same process. At every time, the label of its state is always a suffix of the string generated thus far. It is also clear that a *PSA* is a Markov chain as well. τ and γ can be combined into a single function, ignoring the symbols, and this function, together with the first condition, 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.

A *PSA* may also be used to read a string or stream of symbols. In this mode, the state of the *PSA* at every moment corresponds again to a suffix of the stream and the *PSA* can be used to calculate the probability of seeing any given string in the future, given the label of its current state. Our intention is to use this derived *PSA* to process streams of symbols, so that, while consuming a stream $S_{1..k}$, we can know what its meaningful suffix and use that suffix for any inferences.

However, there is a subtle technical issue about the convertibility of a *PST* to a *PSA*. Not every *PST* can be converted to a *PSA* (but every *PST* can be converted to a larger class of so-call probabilistic automata). This is achievable under a certain condition. If this condition does not hold, then the *PST* can be converted to an automaton that is composed of a *PSA* as usual, with the addition of some extra states. These states, viewed as states of a Markov chain, are transient. This means that the automaton will move through these states for some transitions, but it will finally end into the states of the *PSA*, stay in that class and never return to any of the transient states. In fact, if the automaton starts in any of the transient states, then it will enter the single, recurrent class of the *PSA* in at most m_{max} transitions. Given the fact that in our work we deal with streams of infinite length, it is certain that, while reading a stream, the automaton will have entered the *PSA* after at most m_{max} symbols. Thus, instead of checking this condition, we prefer to simply construct only the *PSA* and wait (for at most m_{max} symbols) until the first $m \leq m_{max}$ symbols of a stream have been consumed and are equal to a label of the *PSA*. At this point, we set the current state of the *PSA* to the state with that label and start processing.

7.4.4 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 15 (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;*
- *$\Delta : Q \times \Sigma \rightarrow Q$ 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.*

The language $\mathscr{L}(M)$ of M is defined, as usual, as the set of strings that lead M to a final state. The following conditions must hold:

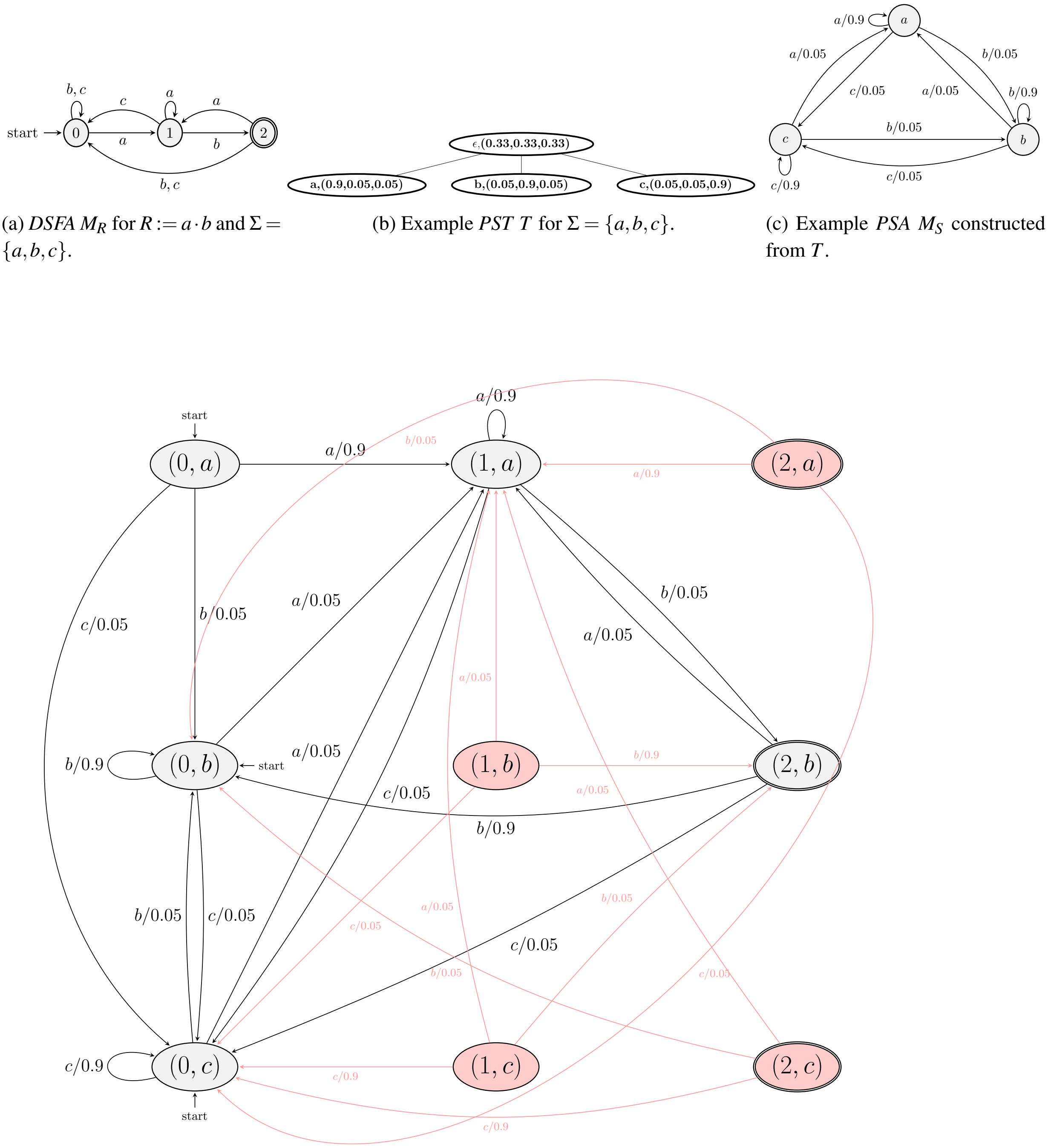
- *$\Sigma = M_R.\Sigma = M_S.\Sigma$;*
- *$\mathscr{L}(M) = \mathscr{L}(M_R)$;*
- *For every string/stream $S_{1..k}$, $P_M(S_{1..k}) = P_{M_S}(S_{1..k})$, where P_M denotes the probability of a string calculated by M (through Γ) and P_{M_S} the probability calculated by M_S (through γ).*

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 3. *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 that is equivalent to both M_R and M_S.*

Proof. Construct an embedding in the following straightforward manner: First let its states be the product $M_R.Q \times M_S.Q$, 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



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

(b) Example *PST* T for $\Sigma = \{a, b, c\}$.

(c) Example *PSA* M_S constructed from T.

(d) Embedding of M_S in M_R.

Figure 25: Embedding example.

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 $\mathscr{L}(M) = \mathscr{L}(M_R)$ 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 $\rho = [l,q_l] \xrightarrow{t_l} [l+1,q_{l+1}] \xrightarrow{t_{l+1}} \cdots \xrightarrow{t_k} [k+1,q_{k+1}]$. 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 $\rho^{'} = [l,q_l^{'}] \xrightarrow{t_l} [l+1,q_{l+1}^{'}] \xrightarrow{t_{l+1}} \cdots \xrightarrow{t_k} [k+1,q_{k+1}^{'}]$. 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 25a for the expression $R = a \cdot b$ with $\Sigma = \{a,b,c\}$. We present it as a classical automaton, but we remind readers that symbols correspond to minterms. Thus, Σ is the set of minterms. For example, a could correspond to the minterm ψ_A, b to ψ_B and c to $\neg\psi_A \land \neg\psi_B$ of Table 10. Figure 25b depicts a possible *PST* T that could be learned from a training stream composed of symbols from Σ. Figure 25c shows the *PSA* M_S constructed from T. For simplicity reasons, we assume that the stream is simple enough so that all labels/contexts are of length 1. Figure 25d shows the embedding M that would be created, following the construction procedure of the proof of Theorem 3. 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 25a 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 25d, including the redundant red states. To avoid the inclusion of such states, we can merge M_R and M_S in an incremental fashion (see Algorithm 11). The resulting automaton would then consist only in the black states and transitions of Figure 25d.

ALGORITHM 11: Embedding of a *PSA* in a *DSFA* (incremental).

```
Input: A DSFA $M_R$ and a PSA $M_S$ learnt with the minterms of $M_R$
Output: An embedding $M$ of $M_S$ in $M_R$ equivalent to both $M_R$ and $M_S$
/* First create the initial states of the merged automaton by combining the initial state of $M_R$ with all the
   states of $M_S$.                                                                                          */
$Q^s \leftarrow \emptyset$;
foreach $s \in M_S.Q$ do
    $q \leftarrow CreateNewState(M_R.q^s, s)$;
    /* $q$ is a tuple $(r,s)$                                                                                */
    $Q^s \leftarrow Q^s \cup \{q\}$;
/* A frontier of states is created, including states of $M$ that have no outgoing transitions yet.  First
   frontier consists of the initial states of $M$                                                            */
$Frontier \leftarrow Q^s$; $Checked \leftarrow \emptyset$; $\Delta \leftarrow \emptyset$; $\Gamma \leftarrow \emptyset$;
while $Frontier \neq \emptyset$ do
    $q \leftarrow$ pick an element from $Frontier$;
    foreach $\sigma \in M_S.\Sigma$ do
        $s^{next} \leftarrow M_S.\tau(q.s, \sigma)$; $r^{next} \leftarrow M_R.\delta(q.r, \sigma)$;
        if $(r^{next}, s^{next}) \notin Checked$ then
            $q^{next} \leftarrow CreateNewState(r^{next}, s^{next})$;
            $Frontier \leftarrow Frontier \cup q^{next}$;
        else
            $q^{next} \leftarrow (r^{next}, s^{next})$;
        /* Both the symbol $\sigma$ and its probability are added to the transition.                         */
        $\delta \leftarrow CreateNewTransition(q, \sigma, q^{next})$;
        $\gamma \leftarrow CreateNewProbability(q, \sigma, M_S.\gamma(q.s, \sigma))$;
        $\Delta \leftarrow \Delta \cup \delta$; $\Gamma \leftarrow \Gamma \cup \gamma$;
    $Checked \leftarrow Checked \cup \{q\}$; $Frontier \leftarrow Frontier \setminus \{q\}$;
$Q \leftarrow Checked$;
/* Create the final states of $M$ by gathering all states of $M$ whose second element is a final state of $M_R$.
   */
$Q^f \leftarrow \emptyset$;
foreach $q \in Q$ do
    if $q.q_R \in M_R.Q^f$ then
        $Q^f \leftarrow Q^f \cup \{q\}$;
$\Sigma \leftarrow M_S.\Sigma$;
return $M = (Q, Q^s, Q^f, \Sigma, \Delta, \Gamma)$;
```

7.4.5 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):

$$\Pi = \begin{pmatrix} N & N_F \\ F_N & F \end{pmatrix} \tag{29}$$

where N is the sub-matrix containing the probabilities of transitions from non-final to non-final states, F the probabilities from final to final states, F_N the probabilities from final to non-final states and 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 starting from a state in N:

Theorem 4 ([54])**.** *Let Π be the transition probability matrix of a homogeneous Markov chain Y_t in the form of Equation* (29) *and ξ_{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*

$$P(Y_n \in F, Y_{n-1} \notin F, ..., Y_1 \notin F \mid \xi_{init}) = \xi_N{}^T N^{n-1}(I-N)1 \tag{30}$$

where ξ_N is the vector consisting of the elements of ξ_{init} corresponding to the states of N.

In our case, the sets N and F have the meaning of being the non-final and final states of M. The above theorem then gives us the desired probability of reaching a final state.

However, notice that this theorem assumes that we start in a non-final state ($Y_1 \notin F$). A similar result can be given if we assume that we start in a final state.

Theorem 5. *Let Π be the transition probability matrix of a homogeneous Markov chain Y_t in the form of Equation* (29) *and ξ_{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*

$$P(Y_n \in F, Y_{n-1} \notin F, \cdots, Y_1 \in F \mid \xi_{init}) = \begin{cases} \xi_F{}^T F 1 & \textit{if } n = 2 \\ \xi_F{}^T F_N N^{n-2}(I-N)1 & \textit{otherwise} \end{cases} \tag{31}$$

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, $P(Y_2 \in F, Y_1 = i \in F \mid \xi_{init}) = \xi(i)\sum_{j \in F} \pi_{ij}$, 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, $P(Y_2 \in F, Y_1 \in F \mid \xi_{init}) = \sum_{i \in F} \xi(i) \sum_{j \in F} \pi_{ij}$. In matrix notation, this is equivalent to $P(Y_2 \in F, Y_1 \in F \mid \xi_{init}) = \xi_F{}^T F 1$.

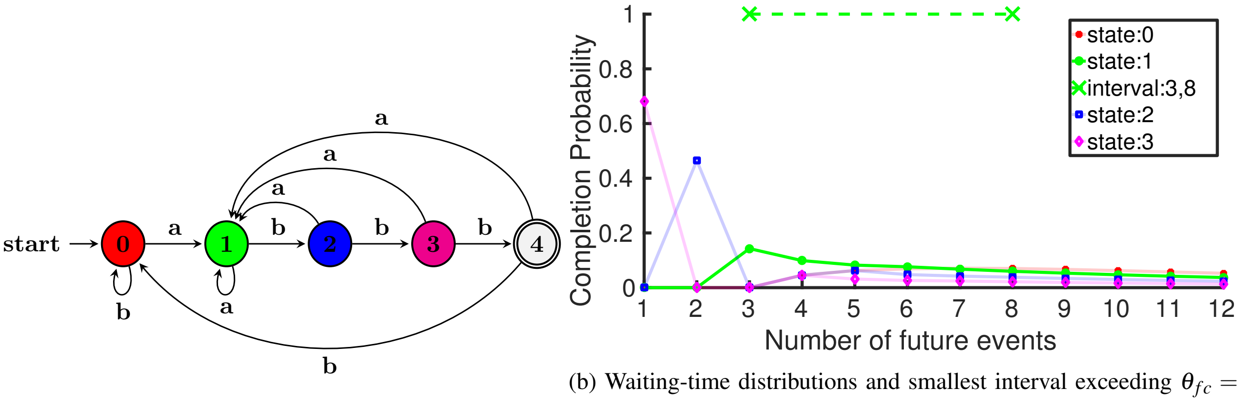
Case where $n > 2$.
In this case, we must necessarily first take a transition from $i \in F$ to $j \in N$, then, for multiple transitions we remain in N and we finally take a last transition from N to F. We can write

$$\begin{aligned} P(Y_n \in F, Y_{n-1} \notin F, ..., Y_1 \in F \mid \xi_{init}) =& P(Y_n \in F, Y_{n-1} \notin F, ..., Y_2 \notin F \mid \xi_N') \\ =& P(Y_{n-1} \in F, Y_{n-2} \notin F, ..., Y_1 \notin F \mid \xi_N') \end{aligned} \tag{32}$$

where ξ_N' is the state distribution (on states of N) after having taken the first transition from F to N. This is given by $\xi_N' = \xi_F{}^T F_N$. By using this as an initial state distribution in 30 and running the index n from 1 to $n-1$, as in 32, we get

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

□



(a) DFA.

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

Figure 26: Automaton and waiting-time distributions for $R = a \cdot b \cdot b \cdot b$, $\Sigma = \{a, b\}$.

Using Theorems 4 and 5, 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 Theorems 4 and 5 provide us with a way to calculate the probability of reaching a final state, given an initial state distribution ξ_{init}. In our case, where we have an automaton moving through its various states, ξ_{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, ξ_{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.

Figure 26 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).

The actual forecast from a state q has the form of an interval whose calculation is based on the waiting-time distribution of q. The meaning of such a forecast interval $I = (start, end)$ is the following: given that we are in a state q, we forecast that the automaton, with confidence at least θ_{fc}, will have reached its final state(s) in n transitions from now, where $start \leq n \leq end$. The confidence threshold θ_{fc} is a parameter set by the user. Since multiple intervals exceeding θ_{fc} might exist, we choose the one with the smallest *spread*, where $spread = end - start$. Figure 26b shows the forecast interval produced for state 1 of the automaton of Figure 26a, with $\theta_{fc} = 50\%$.

7.4.6 Avoiding the construction of the Markov chain

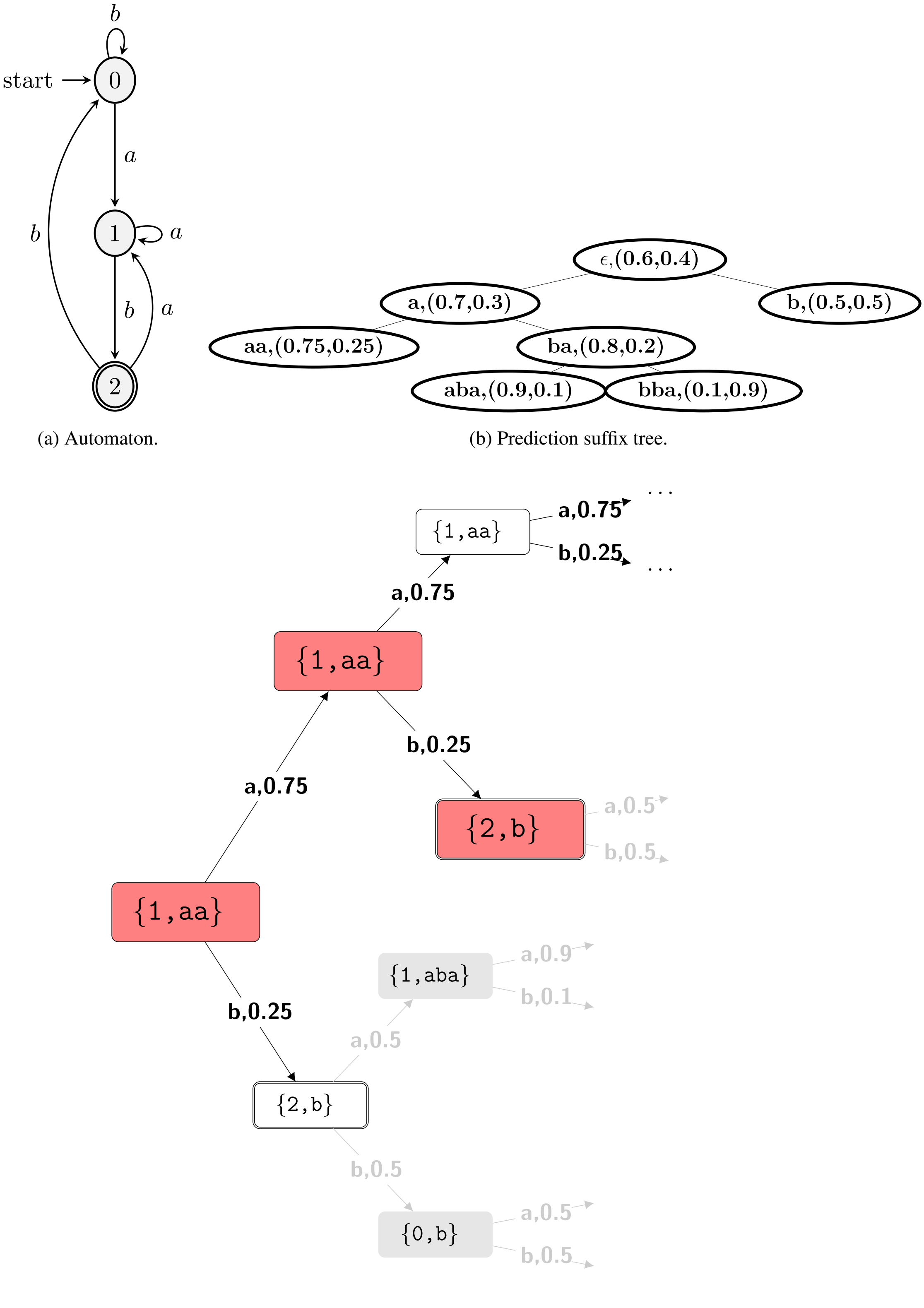
The reason for constructing an embedding, as described above, is that it is based on a variable-order model and the expectation is that it will thus consist of much fewer states than a full-order model. In practice, however, we have observed that the gains from creating an embedding are not as significant as we would expect. As we will also show in Section 7.5, 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 Theorems 4 and 5, 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.

Figure 27 shows an example of this process. Figure 27a shows an example *DSFA* M_R and Figure 27b an example *PST* T. 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 overcome 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 27c 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 27c. 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 27c as the 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 27c, 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



(a) Automaton.

(b) Prediction suffix tree.

(c) Future states visited.

Figure 27: Example of estimating a waiting-time distribution without a Markov chain.

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 27c, 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 27c 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.

7.4.7 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 [127], it is assumed that, before learning a *PST*, the empirical probabilities of symbols given various contexts are available. The suggestion in [127] 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 [127]:

$$\hat{P}(s) = \frac{1}{k-m} \sum_{j=m}^{k-1} \chi_j(s) \tag{33}$$

$$\hat{P}(\sigma \mid s) = \frac{\sum_{j=m}^{k-1} \chi_{j+1}(s \cdot \sigma)}{\sum_{j=m}^{k-1} \chi_j(s)} \tag{34}$$

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

$$\chi_j(s) = \begin{cases} 1 & \text{if } S_{j-|s|+1..j} = s \\ 0 & \text{otherwise} \end{cases} \tag{35}$$

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 $\sigma_0 = \varepsilon$ 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 28, which depicts the *CST* of maximum

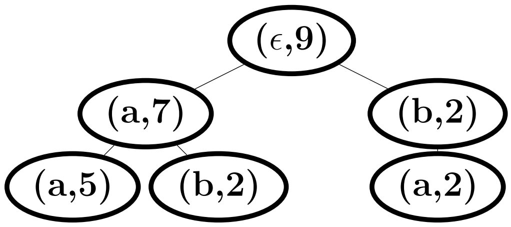


Figure 28: Example of a Counter Suffix Tree with $m = 2$ and $S = aaabaabaaa$.

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 35, and estimate the empirical probabilities of Equations 33 and 34.

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 29 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 ε 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 29 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 [127], is polynomial in m, n, $\frac{1}{\varepsilon}$ and the size of the alphabet (number of minterms in our case). The complexity of estimating the waiting-time distributions 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 30 and 31. A straightforward way to multiply a matrix 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 N. Thus, a total of $N^3(N-1)$ operations would be required for one matrix multiplication. For raising N to the power of $n-1$, a total of $(n-2)N^3(N-1)$ operations would be required, assuming that N^k is calculated by multiplying N by 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 forecast intervals (steps 1,3,4 and 6 in Figure 29). We also give complexity results for the alternative option that bypasses the PSA (step $3'$).

Proposition 4 (Step 1 in Figure 29)**.** *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",

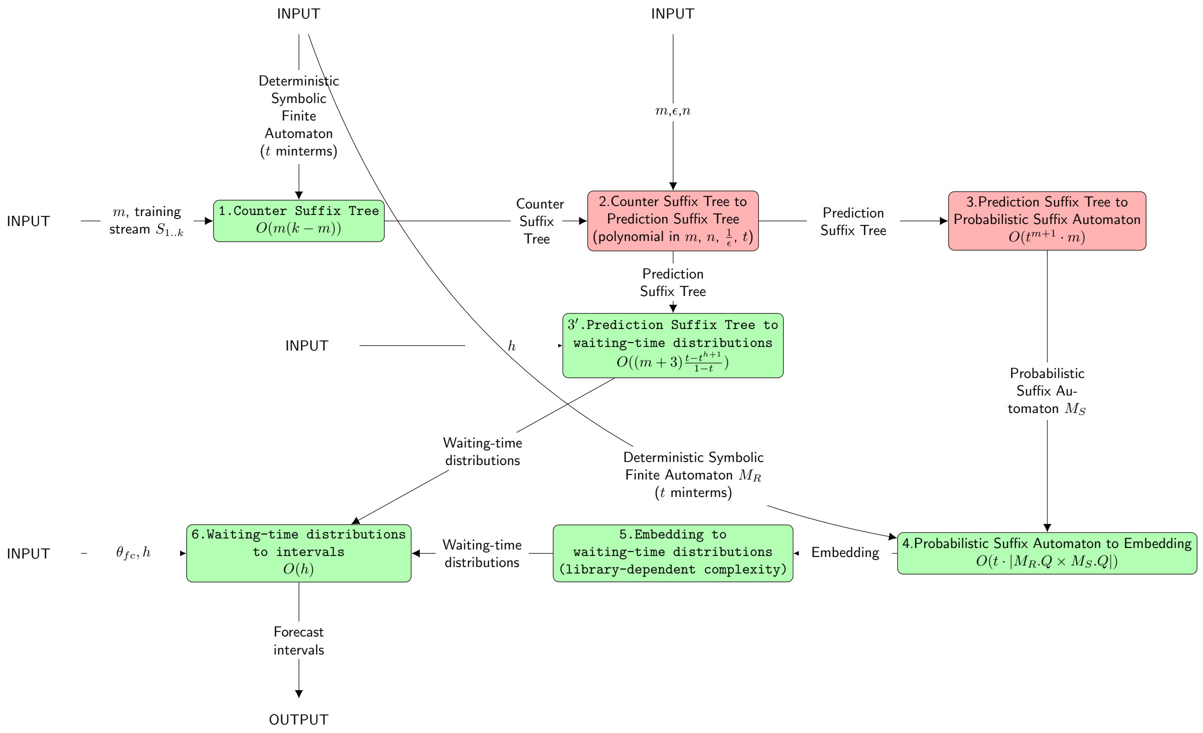

INPUT
Deterministic Symbolic Finite Automaton (t minterms)
INPUT
m,ϵ,n
INPUT
m, training stream $S_{1..k}$
1.Counter Suffix Tree $O(m(k-m))$
Counter Suffix Tree
2.Counter Suffix Tree to Prediction Suffix Tree (polynomial in m, n, $\frac{1}{\epsilon}$, t)
Prediction Suffix Tree
3.Prediction Suffix Tree to Probabilistic Suffix Automaton $O(t^{m+1} \cdot m)$
Prediction Suffix Tree
INPUT
h
3′.Prediction Suffix Tree to waiting-time distributions $O((m+3)\frac{t-t^{h+1}}{1-t})$
Probabilistic Suffix Automaton M_S
Waiting-time distributions
Deterministic Symbolic Finite Automaton M_R (t minterms)
INPUT
θ_{fc}, h
6.Waiting-time distributions to intervals $O(h)$
Waiting-time distributions
5.Embedding to waiting-time distributions (library-dependent complexity)
Embedding
4.Probabilistic Suffix Automaton to Embedding $O(t \cdot |M_R.Q \times M_S.Q|)$
Forecast intervals
OUTPUT

Figure 29: Steps for calculating the forecast intervals of a *DSFA*. m is the maximum assumed order, ϵ the approximation parameter, n the maximum number of states for the *PSA* and θ_{fc} the confidence threhold of the intervals. Red blocks indicate steps described in [127]. Green blocks indicate steps described here.

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. □

Proposition 5 (Step 3 in Figure 29)**.** *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 $O(t^m \cdot t \cdot m) = O(t^{m+1} \cdot m)$. □

Proposition 6 (Step 4 in Figure 29)**.** *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 $O(t \cdot |M_R.Q \times M_S.Q|)$. □

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 $t^m \cdot |M_R.Q|$. This is also the upper bound of the number of states of the automaton the would be created using the method of [8], 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.

A naive way to estimate the forecast interval from a waiting-time distribution whose domain is $[1,h]$ (we call h, the maximum index of the distribution, its *horizon*) is to first enumerate all possible intervals $(start, end)$, such that $1 \leq start, end \leq h$ and $start \leq end$, and then calculate each interval's probability by summing the probabilities of all of its points. The complexity of such an exhaustive algorithm is $O(h^3)$. To prove this, first note that the algorithm would have to check 1 interval of length h, 2 intervals of length $h-1$, etc., and h intervals of length 1. Assuming that the cost of estimating the probability of an interval is proportional to its length l (l points need to be retrieved and $l-1$ additions be performed), the total cost would thus be

$$
\begin{aligned}
1h + 2(h-1) + 3(h-2) + \cdots + h1 =& \sum_{i=1}^{h} i(h-(i-1)) \\
=& \sum_{i=1}^{h} (ih - i^2 + i) \\
=& h\sum_{i=1}^{h} i - \sum_{i=1}^{h} i^2 + \sum_{i=1}^{h} i \\
=& h\frac{h(h+1)}{2} - \frac{h(h+1)(2h+1)}{6} + \frac{h(h+1)}{2} \\
=& \cdots \\
=& \frac{1}{6}h(h+1)(h+2) \\
=& O(h^3)
\end{aligned}
$$

Note that this is just the cost of estimating the probabilities of the intervals, ignoring the costs of actually creating them first and then searching for the best one, after the step of probability estimation.

We can find the best forecast interval with a more efficient algorithm that has a complexity linear in h (see Algorithm 12). We keep two pointers i, j that we initially set them equal to the first index of the distribution. We then repeatedly move i, j in the following manner: We first move j to the right by incrementing it by 1 until $P(i,j)$ exceeds θ_{fc}, where each $P(i,j)$ is estimated incrementally by repeatedly adding $P(j)$ to an accumulator. We then move i to the right by 1 until $P(i,j)$ drops below θ_{fc}, where $P(i,j)$ is estimated by incremental subtractions. If the new interval (i,j) is smaller

ALGORITHM 12: Estimating a forecast interval from a waiting-time distribution.

```
Input: A waiting-time distribution $P$ with horizon $h$ and a threshold $\theta_{fc} < 1.0$
Output: The smallest interval $I = (s,e)$ such that $1 \leq s,e \leq h$, $s \leq e$ and $P(I) \geq \theta_{fc}$
$s \leftarrow -1$; $e \leftarrow -1$; $i \leftarrow 1$; $j \leftarrow 1$; $p \leftarrow P(1)$;
while $j \neq h$ do
    /* Loop invariant: $(s,e)$ is the smallest interval with $P((s,e)) > \theta_{fc}$ among all intervals with $e \leq j$ (or
       $s = e = -1$ in the first iteration).                                                              */
    /* Expansion phase.                                                                                   */
    while $(p < \theta_{fc}) \wedge (j < h)$ do
        $j \leftarrow j+1$;
        $p \leftarrow p + P(j)$;
    /* Shrinking phase.                                                                                   */
    while $p \geq \theta_{fc}$ do
        $i \leftarrow i+1$;
        $p \leftarrow p - P(i)$;
    $i \leftarrow i-1$;
    /* $s = -1$ indicates that no interval has been found yet, i.e., that this is the first iteration.    */
    if $(spread((i,j)) < spread((s,e))) \vee (s = -1)$ then
        $s \leftarrow i$;
        $e \leftarrow j$;
return $(s,e)$;
```

than the smallest interval exceeding θ_{fc} thus far, we discard the old smallest interval and keep this new one. This wave-like movement of i, j stops when $j = h$. This algorithm is more efficient by avoiding intervals that cannot possibly exceed θ_{fc}.

We first present a correctness proof for this algorithm:

Proof of correctness for Algorithm 12. We only need to prove the loop invariant. Assume that after the k^{th} iteration of the outer while loop $i = i_k$ and $j = j_k$ and that after the $(k+1)^{th}$ iteration $i = i_{k+1}$ and $j = j_{k+1}$. If the invariant holds after the k_{th} iteration, then all intervals with $e \leq j_k$ have been checked and we know that (s,e) is the best interval up to j_k. It can be shown that, during the $(k+1)^{th}$ iteration, the intervals up to j_{k+1} that are not explicitly checked are intervals which cannot possibly exceed θ_{fc} or cannot be better than the currently held best interval (s,e). There are three such sets of unchecked intervals (see also Figure 30):

- All intervals (i', j') such that $i' < i_k$ and $j_k \leq j' \leq j_{k+1}$, i.e., we ignore all intervals that start before i_k. Even if these intervals exceed θ_{fc}, they cannot possibly be smaller than (s,e), since we know that $(s,e) = (i_k, j_k)$ or that (s,e) is even smaller than (i_k, j_k).

- All intervals (i', j') such that $i' > i_{k+1}$ and $j_k \leq j' \leq j_{k+1}$, i.e., we ignore all intervals that start after i_{k+1}. These intervals cannot possibly exceed θ_{fc}, since $(i_{k+1}+1, j_{k+1})$ is below θ_{fc} and all these intervals are sub-intervals of $(i_{k+1}+1, j_{k+1})$.

- We are thus left with intervals (i', j') such that $i_k \leq i' \leq i_{k+1}$ and $j_k \leq j' \leq j_{k+1}$. Out of all the interval that can

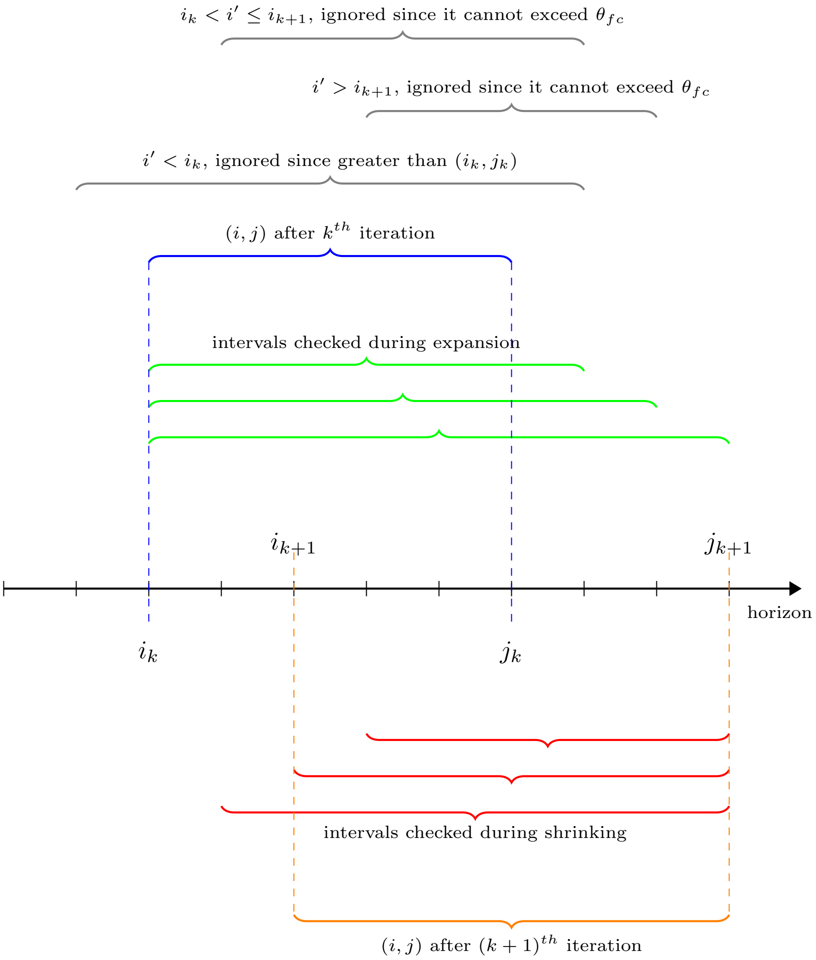

$i_k < i' \leq i_{k+1}$, ignored since it cannot exceed θ_{fc}
$i' > i_{k+1}$, ignored since it cannot exceed θ_{fc}
$i' < i_k$, ignored since greater than (i_k, j_k)
(i, j) after k^{th} iteration
intervals checked during expansion
i_{k+1}
j_{k+1}
horizon
i_k
j_k
intervals checked during shrinking
(i, j) after $(k+1)^{th}$ iteration

Figure 30: One iteration of Algorithm 12. After the k^{th} iteration, (i, j) is either the smallest interval or there already exists a smaller one. In the $(k+1)^{th}$ iteration, we need to check intervals for which $j_k < j' \leq j_{k+1}$, .i.e., whose right limit is greater than j_k and smaller (but including) j_{k+1}. Out of those intervals, the ones that start before i_k need not be checked because they are greater than (i_k, j_k). The ones that start after i_{k+1} are also ignored, since we already know, from the shrinking phase, that their super-interval $(i_{k+1}+1, j_{k+1})$ does not exceed θ_{fc}. Finally, we can also ignore the intervals that start between i_k and i_{k+1} (not including i_k) and end before j_{k+1} (not including j_{k+1}), because they cannot exceed θ_{fc}. If such an interval actually existed, then the expansion phase would have stopped before j_{k+1}.

be constructed from combining these i' and j', the algorithm checks the intervals $(i' = i_k, j')$ and $(i', j' = j_{k+1})$. The intervals that are thus left unchecked are the intervals (i', j') such that $i_k < i' \leq i_{k+1}$ and $j_k \leq j' < j_{k+1}$. The question is: is it possible for such an interval to exceed θ_{fc}. The answer is negative. Assume that there is such an interval (i', j'). If this were the case, then the algorithm, during its expansion phase, would have stopped at j', because (i_k, j') would exceed θ_{fc}. Therefore, these intervals cannot exceed θ_{fc}.

A similar reasoning allows us to show that the loop invariant holds after the first iteration. It thus holds after every iteration. □

Proposition 7 (Step 6 in Figure 29). *For a waiting-time distribution with a horizon of length h, the complexity of finding* *the smallest interval that exceeds a confidence threshold* θ_{fc} *is* $O(h)$.

Proof. Indexes i and j of Algorithm 12 scan the distribution only once. The cost for j is the cost of h points of the distribution that need to be accessed plus $h-1$ additions. Similarly, the cost for i is the cost of (at most) h accessed points plus the cost of (at most) $h-1$ subtractions. Thus the total cost is $O(h)$. □

Proposition 8 (Step $3'$ in Figure 29)**.** *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 *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 27c. 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 $O((m+3)\frac{t-t^{h+1}}{1-t})$. □

7.5 Empirical Evaluation

We now present experimental results on both synthetic and real-world datasets. As described in Section 7.4.5, the output of our system consists of forecast intervals, estimated from the waiting-time distributions of our automaton. These waiting-time distributions are estimated in turn either from a (full- or variable-order) Markov chain corresponding to our automaton or directly from a prediction suffix tree, when the optimization of Section 7.4.6 is employed. This is how we envision a forecasting engine working in practice. However, for experimental testing we proceed with a more thorough evaluation, conducting three different types of experiments: a) SE forecasting, where our goal is to forecast the next SE in the input stream; b) regression CE forecasting, where our goal is to forecast when a CE will occur; c) classification CE forecasting, where our goal is to forecast whether or not a CE will occur within a short future window.

7.5.1 SE Forecasting

Although our main focus is on forecasting occurrences of CEs, we start with some simple experiments, targeting SEs. This will not only allow us to establish a baseline with some more easily interpretable results, but it will also enable us to show the differences between SE and CE forecasting. As we will show, CE forecasting is more challenging than SE forecasting, in terms of the feasibility of looking deep into the past. Another reason for running experiments for SE forecasting is to find the best values for the hyperparameters used for learning a prediction suffix tree. Since it is much faster to run this type of experiments, compared to CE forecasting experiments, we use a hypergrid of hyperparameter values and for each hypergrid point we run SE forecasting. We show the best results and we also keep the hyperparameter values that give us these best results for the later experiments with CE forecasting.

In this type of experiments, our goal is to investigate how well our proposed framework can forecast the next SE to appear in the stream, given that we know the last m SEs. This task is the equivalent of *next symbol prediction* in the terminology of the compression community [23]. As explained in Section 7.4.2, the metric that we use to estimate the quality of a predictor $\hat{P}$ is the average log-loss with respect to a test sequence $S_{1..k} = t_1, t_2, \cdots, t_k$, given by $l(\hat{P}, S_{1..k}) = -\frac{1}{T}\sum_{i=1}^{k} log\hat{P}(t_i \mid t_1 \cdots t_{i-1})$. The lower the average log-loss, the better the predictor is assumed to be. The predictors that we test are the following: a) full-order Markov models for various values of the order m, as is done in previous forecasting studies [8, 9]; b) variable-order Markov models for various values of the maximum order m, in the form of prediction suffix trees, as described in [127, 126] and Section 7.4.3. As a final note, we remind that the "symbols" which we try to predict in these experiments are essentially the minterms of our *DSFA* in our case. In other words, we do not try to predict exactly what the next SE will be, but which minterm the next SE will satisfy. For example, if we have the minterms of Table 10, then our task is to predict whether the next SE will satisfy ψ_A (i.e., the speed of the vessel will be below 5 knots), ψ_B (i.e., the speed will be above 20 knots) or $\neg\psi_A \wedge \neg\psi_B$ (i.e., the speed will be between 5 and 20 knots).

7.5.2 Regression CE Forecasting

After SE forecasting, we move on to regression experiments on CE forecasting. Our goal in this type of experiments is to forecast when a CE will occur. We call them *regression* experiments due to the fact that the forecasts are "continuous" values, in the form of forecast intervals/points. This is in contrast to the next type of experiments, where each forecast is a binary value, indicating whether a CE will occur or not and are thus called *classification* experiments.

One important difference between SE and CE forecasting (both regression and classification) is that, in SE forecasting, a forecast is emitted after every new SE is consumed. On the other hand, in CE forecasting, emitting a forecast after every new SE 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 SEs. As a result, if we emit a forecast after every new SE, 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.

For these reasons, before running a regression experiment, we first go through a preprocessing step. The goal is to find the timepoints within a stream where it is "meaningful" to emit forecasts. We call these timepoints the *checkpoints* of the stream. To do so, we first perform recognition on the stream to find the timepoints where CEs are detected. We then set a required distance d that we want to separate a forecast from its CE, in the sense that we require each forecast to be emitted d events before the CE. After finding all the CEs in a stream and setting a value for d, we set as checkpoints all the SEs which occur d events before the CEs. This typically means that we end up with as many checkpoints as CEs for a given value of d (unless the distance between two consecutive CEs is smaller than d, in which case no checkpoint for the second CE exists). We show results for various values of d, starting from the smallest possible value of 1 (i.e., emitting forecasts from the immediately previous SE before the CE).

At each checkpoint, a forecast is produced as per Section 7.4.5. We do impose, however, an extra constraint, requiring that the maximum spread of each forecast is 0, i.e., we produce point (instead of interval) forecasts. The reason for this is that some of the metrics we use to assess the quality of the forecasts assume that forecasts are in the form of points. Such point metrics are the following: the Root Mean Squared Error (RMSE) and the Mean Absolute Error (MAE) (the latter is less sensitive than RMSE to outliers). If we denote by C the set of all checkpoints, by y_c the actual distance (in number of events) between a checkpoint c and its CE (which is always d) and by $\hat{y}_c$ our forecast, than the definitions for RMSE and MAE are as follows:

$$RMSE = \sqrt{\frac{1}{|C|}\sum_{c \in C}|\hat{y}_c - y_c|^2} \tag{36}$$

$$MAE = \frac{1}{|C|}\sum_{c \in C}|\hat{y}_c - y_c| \tag{37}$$

Besides these points metrics, we also use an interval metric, the so-called *negatively oriented interval score* [64]. If $\hat{y}_c = (l_c, u_c)$ is an interval forecast produced with confidence $b = 1 - a$ and y_c the actual observation (distance), then the negatively oriented interval score (NOIS) for this forecast is given by:

$$NOIS_c = (u_c - l_c) + \frac{2}{a}(l_c - y_c)I_{x<l_c} + \frac{2}{a}(y_c - u_c)I_{x>u_c} \tag{38}$$

We estimate the average negatively oriented interval score (ANOIS) as follows:

$$ANOIS = \frac{1}{|C|}\sum_{c \in C} NOIS_c \tag{39}$$

The best possible value for ANOIS is 0 and is achieved only when all forecasts are point forecasts (so that $u_c - l_c$ is always 0) and all are also correct (so that the last two terms in Eq. 38 are always 0). In every other case, a forecast is penalized if its interval is long (so that focused intervals are promoted), regardless of whether it is correct. If it is indeed correct, no other penalty is added. If it is not correct, then an extra penalty is added, which is essentially the deviation of the forecast from the actual observation, weighted by a factor that grows with the confidence of the forecast. For example, if the confidence is 100%, then $b = 1.0$, $a = 0.0$ and the extra penalty, according to Eq. 38, grows to infinity. Incorrect forecasts produced with high confidence are thus severely penalized. Note that if we emit only point forecasts ($\hat{y}_c = l_c = u_c$), as in our regression experiments, then NOIS and ANOIS could be written as follows:

$$NOIS_c = \frac{2}{a}|\hat{y}_c - y_c| \tag{40}$$

$$ANOIS = \frac{1}{|C|}\sum_{c \in C}\frac{2}{a}|\hat{y}_c - y_c| \tag{41}$$

ANOIS then becomes a weighted version of MAE.

7.5.3 Classification CE Forecasting

The last type of experiments is the most challenging. In contrast to regression experiments, where we emit forecasts only at a specified distance before each CE, in classification experiments we emit forecasts regardless of whether a CE occurs or not. The 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.

One issue with classification experiments is that it is not so straightforward to establish checkpoints in the stream. In regression experiments, CEs provide a natural point of reference. In classification experiments, we do not have such reference points, since we are also required to predict the absence of CEs. As a result, instead of using the stream to find checkpoints, we use the structure of the automaton itself. 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 be dividing them by the greatest among them. A 0% distance will thus refer to final states, whereas a 100% distance to the state(s) that are the most distant to a final state, i.e., the automaton has to take the most transitions to reach a final state. These are the start states. We can then determine our checkpoints by specifying the states in which the automaton is permitted to emit forecasts, according to their “expected” distance. For example, we may establish checkpoints by allowing only states with a distance between 40% and 60% to emit forecasts. The intuition here is that, by increasing the allowed distance, we make the forecasting task more difficult.

The 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, albeit in a manner slightly different than the one described in Section 7.4.5. 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.

7.5.4 Models Tested

In the experiments that we present, we test six different models for event forecasting in total. Two of them are variable-order Markov models that we have presented: one that is based on the embedding of a *PSA* in a *DSFA*, as presented in Section 7.4.4, and one that employs the optimization of Section 7.4.6 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 [8, 9], 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 [110], where automata and Markov chains are used once again. However, the pattern 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 SEs 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 [113]. 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 SEs. 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 [143]. 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".

All of the above mentioned models are tested in the experiments targeting CE forecasting. For the experiments that focus on SE forecasting, we exclude the last two methods (HMMs and mean "waiting times") because they have been developed only for CE forecasting and are not capable of performing "next symbol prediction", at least as originally conceived. As far as our proposed methods are concerned, for SE forecasting, we use directly the *PST* learned from the stream. Since there is no actual pattern in this case, there is no point in converting the *PST* to an automaton and then creating an embedding. We can perform SE forecasting directly with the *PST*.

7.5.5 Hardware and Software Settings

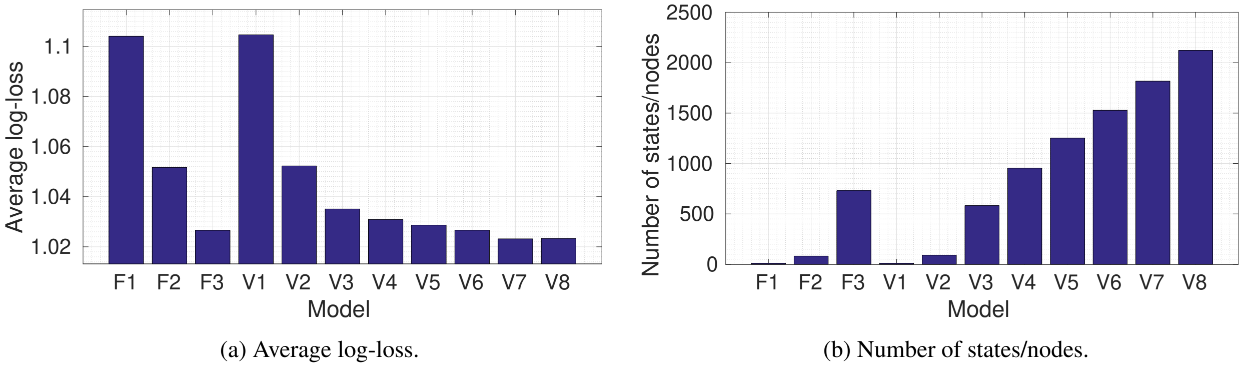
All experiments were run on a simple machine with Intel Core i7-6500U @ 2.50GHz x 4 processors and 8 GB of memory. Our framework was implemented in Scala 2.12.10. We used Java 1.8, using the default values for the heap size. For the HMM models, we relied on the Smile machine learning library [2]. All other models were developed by us. No attempt at parallelization was made. All experiments were run in a centralized setting.

7.5.6 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 [15]. 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 tool 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 [15]. 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



(a) Average log-loss.

(b) Number of states/nodes.

Figure 31: Results for SE 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 (a prediction suffix tree) of maximum order *x*.

is the following:

$$\begin{aligned} R := \ & (amountDiff > 0) \cdot (amountDiff > 0) \cdot (amountDiff > 0) \cdot (amountDiff > 0) \cdot \\ & (amountDiff > 0) \cdot (amountDiff > 0) \cdot (amountDiff > 0) \end{aligned} \quad (42)$$

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 sub-expressions, 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 42, 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 42, 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).

We now present results for SE forecasting. As already mentioned in Section 7.5.4, for this type of experiments we do not use the automaton created by Expression 42. We instead use only its minterms which will constitute our "alphabet". In our case, there are four minterms: a) $amountDiff > 0 \wedge amountDiff > 100$; b) $amountDiff > 0 \wedge \neg(amountDiff > 100)$; c) $\neg(amountDiff > 0) \wedge amountDiff > 100$; d) $\neg(amountDiff > 0) \wedge \neg(amountDiff > 100)$. Thus, the goal is to predict, as the stream is consumed, which one of these minterms will be satisfied. Notice that, for every possible event, exactly one minterm is satisfied (the third one, $\neg(amountDiff > 0) \wedge amountDiff > 100$, is actually unsatisfiable). We use 75% of the original dataset (which amounts to 750.000 transactions) for training and the rest (250.000) for testing. We do not employ cross-validation, as the dataset is synthetic and the statistical properties of its folds would not differ.

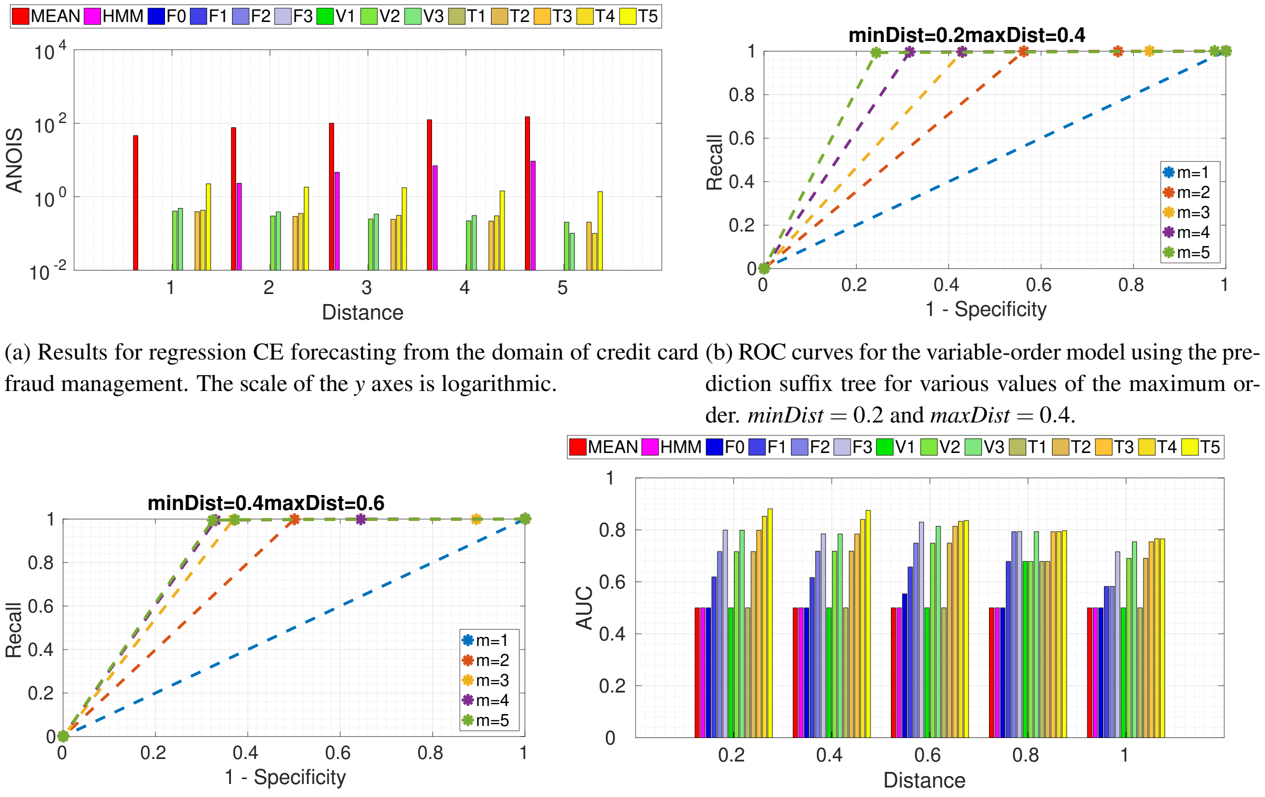
Figure 31a shows the average log-loss obtained for various models and orders *m* and Figure 31b shows the number of states for the full-order models or nodes for the variable-order models, which are prediction suffix trees. The best result is achieved with a variable-order Markov model of maximum order 7. The full-order Markov models are slightly better than their equivalent (same order) variable-order models. This is an expected behavior, since the variable-order models are essentially approximations of the full-order ones. We increase the order of the full-order models until $m = 3$, in which case the Markov chain has $\approx$ 750 states. We avoid increasing the order any further, because Markov chains with more than 1000 states become significantly difficult to manage in terms of memory usage (and in terms of the computational cost of estimating the waiting-time distributions for the experiments of CE forecasting that follow). Note that a Markov chain with 1000 states would require a transition matrix with 1000^2 entries. On the contrary, we can increase the maximum order of the variable-order model until we find the best one, i.e., the order after which the average log-loss starts increasing again. The size of the prediction suffix tree can be allowed to increase to more than 1000 nodes, since we are not required to build a transition matrix.

Figure 32a shows results from regression experiments on CE forecasting, as described in Sections 7.5.2, using Expression 42 and the metric of ANOIS (we avoid showing results for RMSE since they are very similar). The results are grouped by the distance (in number of events) between the detection of the complex event and the time the forecast was produced. Due to the significant differences in the scores, the y axes are on a logarithmic scale. Except for the MEAN method, which consistently performs poorly, all other methods have an ANOIS score that is close or below 1. Markov models seem to be able to achieve better scores than MEAN and HMM. Interestingly though, this is mostly true for low-order Markov models. Increasing the order typically leads to worse scores for variable-order models.

In order to explain this behavior, we will use as an example the automaton of Figure 33. This is a simplified version of the automaton that would be constructed from Expression 42. It only has two minterms and five states, corresponding to a fraud pattern with four transactions. For $m = 0$, the Markov model for this automaton would have exactly the same structure as the automaton itself and the transition probabilities would be the probabilities of encountering an event that satisfies a minterm, regardless of what has occurred in the past. Now, imagine that, at some point we are in state 2 and we attempt to emit a regression forecast about when the automaton is expected to reach state 4. Regardless of the exact transition probabilities, the most probable walk that the automaton can follow to reach state 4 is by first going to state 3 and then to state 4. All other walks leading to state 4 (e.g., by first going to state 0) are longer and contain this shortest walk. Hence, this shortest walk has the highest probability and the regression forecast would always be 2, which is correct. By increasing the order of the variable-order models, we essentially create more walks of variable length to state 4. As a result, walks that are longer than 2 transitions but with higher probability could be created and the regression forecast could be different than 2 in some cases. The distribution smoothing that is also performed with the variable-order models can also affect the choice of the most probable walk to a final state, if multiple such walks exist with similar probabilities.

We now move on to the classification experiments. Figure 32 shows the ROC curves that we obtain by running classification experiments with the variable-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 32b and for $minDist = 0.4$ and $maxDist = 0.6$ in Figure 32c. The more area a ROC curve covers, the better the corresponding model is assumed to be. Contrary to the regression experiments, we see here that increasing the maximum order does indeed lead to better results. Notice, however, that in Figure 32c, 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 32d 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 finally show throughput and training time results in Figure 34. Figure 34b, 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



(a) Results for regression CE forecasting from the domain of credit card fraud management. The scale of the *y* axes is logarithmic.

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

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

(d) AUC for ROC curves for all models.

Figure 32: 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* using an embedding. Tx stands for a Variable-order Markov Model of maximum order *x* using a prediction suffix tree. MEAN stands for the method of estimating the mean of “waiting-times”. HMM stands for Hidden Markov Model.

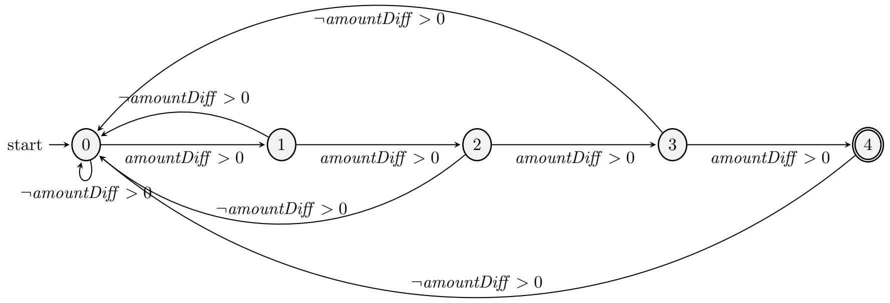
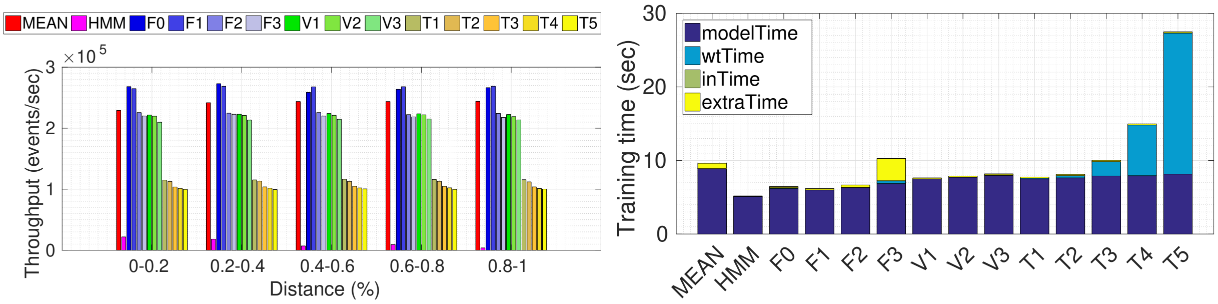


Figure 33: Example *DSFA* for the increasing trend patterns with 4 consecutive increases.



(a) Throughput.

(b) Training time.

Figure 34: 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 interval of all states from their waiting-time distributions. extraTime = time to determinize an automaton (+ disambiguation time for full-order models).

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 interval 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 the direct use of 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 7.4.6. The automata-based models (either full- or variable-order), 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 7.5.4 (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.

7.5.7 Maritime Monitoring

The second dataset that we used for our experiments is a real–world dataset coming from the field of maritime monitoring. Is 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 7.1.1. These trajectories can be analyzed, using the techniques of CER, in order to detect interesting patterns in the behavior of vessels [117]. 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 [123]. 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 [114]. 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 [117]. We further processed the dataset by interpolating between the critical points in order to produce trajectories where two consecutive points have a temporal distance of exactly 60 seconds. The reason for this pre-processing step is that AIS messages typically arrive at unspecified time intervals. These intervals can exhibit a very wide variation, depending on a lot of factors (e.g., human operators may turn on/off the AIS equipment), without any clear pattern that could be encoded by our probabilistic model. Consequently, running our experiments directly on the raw or compressed dataset may yield sub-optimal results.

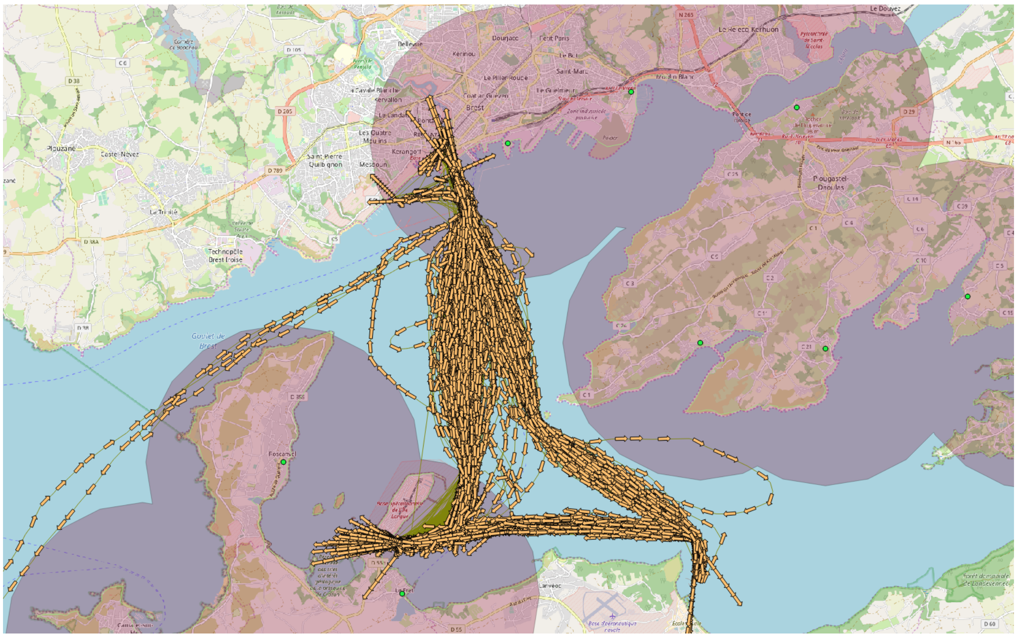


Figure 35: Trajectories of the vessel with the most matches for Pattern 43 around the port of Brest. Green points denote ports. Red, shaded circles show the areas covered by each port. They are centered around each port and have a radius of 5 km.

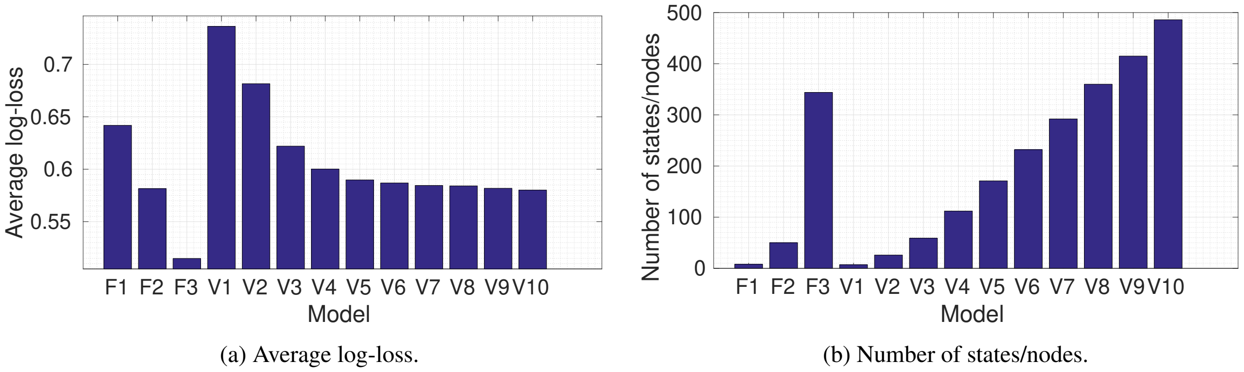
The pattern that we used is a movement pattern in which a vessel approaches the main port of Brest. The goal is to forecast when a vessel will enter the port. The symbolic regular expression for this pattern is the following:

$$\begin{aligned} R := \ & (\neg \mathit{InsidePort}(\mathit{Brest}))^{*} \cdot (\neg \mathit{InsidePort}(\mathit{Brest})) \cdot \\ & (\neg \mathit{InsidePort}(\mathit{Brest})) \cdot (\mathit{InsidePort}(\mathit{Brest})) \end{aligned} \tag{43}$$

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 the port. We use the Boolean operator of negation (¬), 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 43. 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 43. 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 43 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 $\approx$ 30.000. Figure 35 shows the isolated trajectories for this vessel.



(a) Average log-loss.

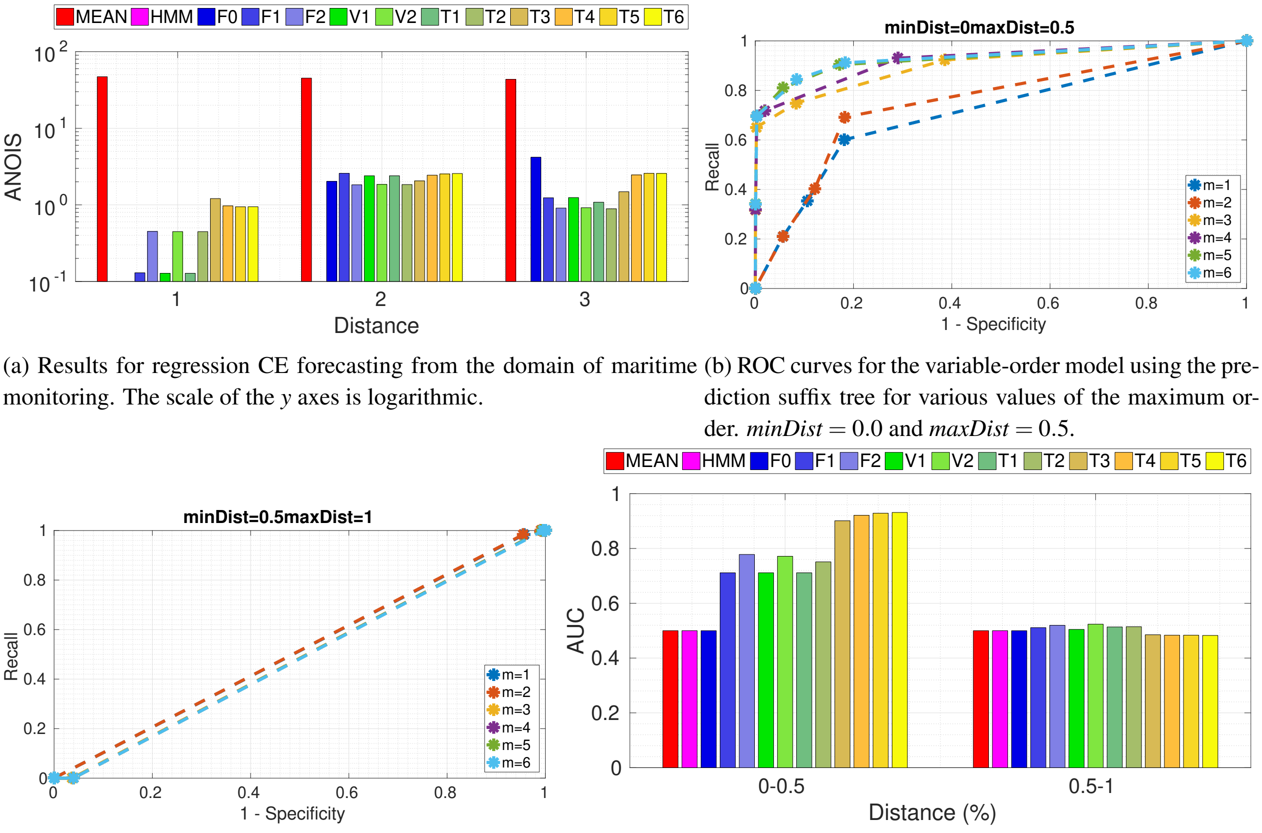
(b) Number of states/nodes.

Figure 36: Results for SE forecasting from the domain of maritime monitoring. Fx stands for a Full-order Markov Model of order x. Vx stands for a Variable-order Markov Model (a prediction suffix tree) of maximum order x.

Figure 36 shows results for SE forecasting. The best average log-loss is achieved with a full-order Markov model, with $m = 3$, and is ≈ 0.51. For the best hyper-parameter values out of those that we tested for the variable-order model, with $m = 10$, we can achieve an average log-loss that is ≈ 0.57. Contrary to the case of credit card data, increasing the order of the variable-order model does not allow us to achieve a better log-loss score than the best one achieved with a full-oder model. However, as we will show, this does not imply that the same is true for CE forecasting.

Figure 37a shows results for regression CE forecasting for 3 different distances. Except for MEAN, which is consistently worse than the others, the rest typically achieve low scores, below 1. We can again observe that increasing the order usually incurs a slight penalty (for an explanation, see the similar results for the dataset of credit card transaction in Section 7.5.6).

We now show results for classification CE forecasting in Figures 37b, 37c, 37d and 38. 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* $= 0\%$,



(a) Results for regression CE forecasting from the domain of maritime monitoring. The scale of the *y* axes is logarithmic.

(b) 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.

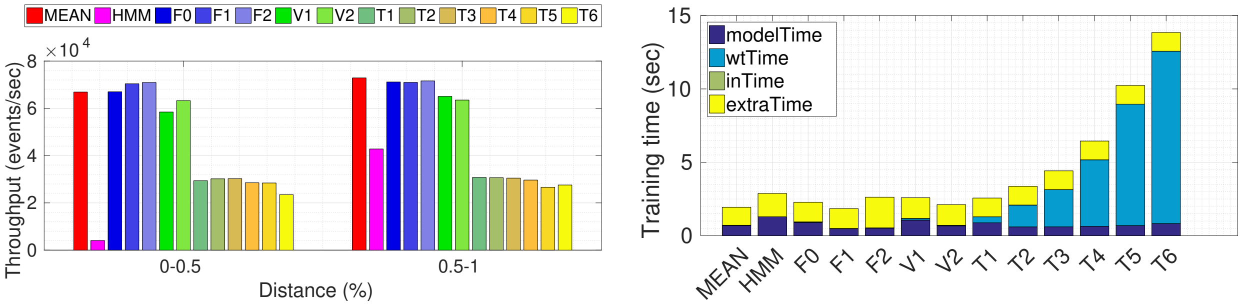
(c) ROC curves for the variable-order model using the prediction suffix tree for various values of the maximum order.*minDist* = 0.5 and *maxDist* = 1.0.

(d) AUC for ROC curves for all models.

Figure 37: Results for CE forecasting from the domain of maritime monitoring. Fx stands for a Full-order Markov Model of order *x*. Vx stands for a Variable-order Markov Model of maximum order *x* using an embedding. Tx stands for a Variable-order Markov Model of maximum order *x* using a prediction suffix tree. MEAN stands for the method of estimating the mean of “waiting-times”. HMM stands for Hidden Markov Model.

or all negative, where we have $Recall = 0\%$ and $Specificity = 100\%$ (see Figure 37c). Achieving higher scores with higher-order models comes at a cost though. The training time for variable-order tends to increase as we increase the order. This increase may be tolerated, given that the total training time is always less than 14 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.

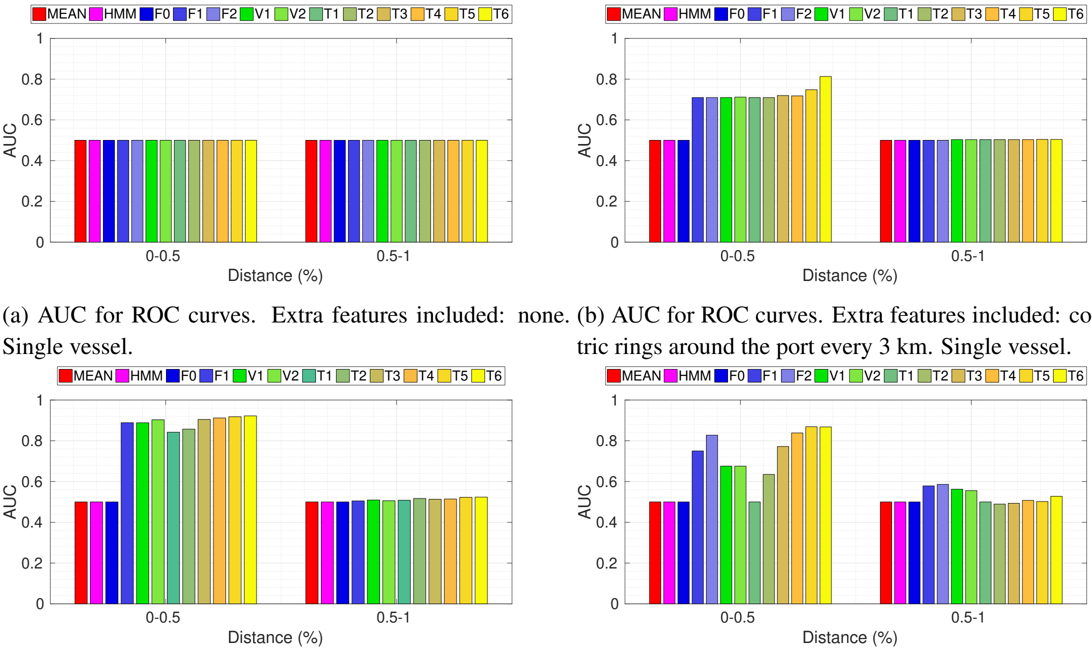
Finally, we test our proposed method in two more scenarios. First, we want to intvestigate how our approach behaves with different extra features (or none at all). Figures 39a, 39b and 39c show the relevant results for classification CE forecasting. Figure 39a shows results when no extra features are 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 39b 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



(a) Throughput.

(b) Training time.

Figure 38: Throughput and training time results for classification CE forecasting from the domain of maritime monitoring. modelTime = time to construct the model. wtTime = time to estimate the waiting-time distributions for all states. inTime = time to estimate the forecast interval of all states from their waiting-time distributions. extraTime = time to determinize an automaton (+ disambiguation time for full-order models).



(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.

Figure 39: Results for classification CE forecasting from the domain of maritime monitoring for various sets of extra features and for multiple vessels.

score achieved is still lower than the best score achieved with “rings” of 1 km (Figure 37d). “Rings” of 1 km are thus more appropriate as predictive features. Figure 39c 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 37d). 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 ≈ 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 39d shows the relevant classification results. The scores are very similar to the scores of the single-vessel model (Figure 37d). 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.

7.6 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.

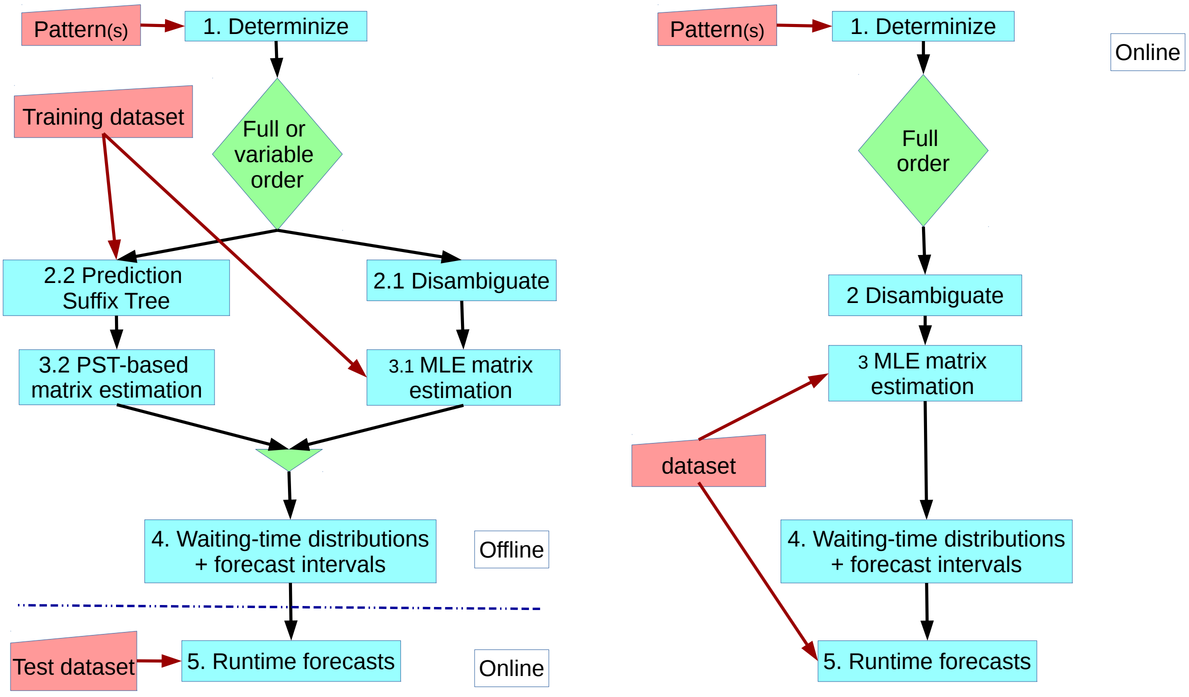
The weak point of our framework, with respect to user involvement, is its inability to automatically select the features that are the most informative. Currently, the user has to manually provide these features, but we intend to explore ways to automate this process in the future. The user also has to 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. Another research avenue that we are currently following concerns the extension of the expressive power of symbolic automata in order to allow n-ary predicates to be used in CE patterns. Some results towards this direction may be found in [10]. 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.

8 Distributed Parameter Estimation for Online Forecasting

In this section we describe our efforts, so far, to make the aforementioned architecture distributed and fully online. First, we talk about online learning and then move to the distribution of both the training and the forecasting phase. We use query based parallelization [62], but aim to implement more distribution techniques in the future (e.g., partition based). Then we move on to evaluate our implementation with a simple maritime example. Finally, we discuss integration to the INFORE platform.

8.1 Online Training

So far, the training of the Markov Chain (MC) was done in an offline manner (Figure 40). A portion of the input stream was used as a training dataset for the MC. When the training was complete the MC could be used to derive probabilistic forecasts from the events of the stream as they arrived (i.e., online).



(a) Original

(b) Online Training

Figure 40: Original and new workflow of the CEF module

When there are multiple patterns to forecast, multiple MCs are also created - one for each pattern. With the method described above, one MC needs to train on the whole stream, then the next MC on the whole stream and so on. This imposes difficulties for our cause, as in online training one does not have the whole stream beforehand to train one MC at a time.

To overcome this issue, we change the iteration from model based to event based. That is, each event in the dataset is processed by all MCs before moving to the next event. This is necessary to implement online learning. The algorithms in Listings 1 and 2 highlight this basic difference between the two approaches.

```
for (model in models ) {
        model.train(stream)
}
```

Listing 1: model iteration

```
for (event in stream) {
        models.train(event)
}
```

Listing 2: event iteration

For our next step, we set a fixed number of events *k* (e.g., 100K). For each *k* events that are fed into the training process we produce a fully functional MC ready to make forecasts (i.e., the MC is updated every *k* events).

This training process is chained to a forecasting process. Every event that is used to train a MC is then forwarded to the forecasting process to make forecasts. Whenever a new MC is produced the forecasting process receives it and uses this one instead. This means that for the first *k* events no forecasting is produced.

Finally, note that in Figure 40b only MLE matrix estimation is depicted as PST-based matrix estimation is not supported yet.

8.2 Flink Distribution

We use Apache's streaming platform Flink [27] to make our implementation distributed, in accordance to the implementation of the other components of the INFORE architecture.

Our first approach uses the query-based parallelization technique [62]. In this technique each query is assigned to a different processing unit and executed in parallel. In our case, queries are essentially patterns transformed to automata.

Figure 41 shows our pattern(query) based implementation. We use online learning described in the previous section. The difference here is that the number of patterns to forecast is equally divided to the number of available processing units, unlike before that there was a single process for forecasting and another one for training.

The flinkSource, which is either a Kafka topic or a file, creates a datastream of events. This stream is broadcast to all parallel instances of the training operator, since all the patterns work for the same stream. The training operator is essentially a RichMap function which contains the training process described before. Every parallel instance initializes the training process when it is created and processes the events for training as they arrive. It then forwards them to the next operator. Whenever *k* events are processed it also forwards the updated MC.

The next operator is the forecasting process and is another RichMap function. Each parallel instance receives the events and processes them for forecasting. Periodically, it also receives the updated MC and re-initializes the forecasting process. Again, we have to note that for the first *k* events no forecasting takes place. The resulting forecasts can be written back to a Kafka topic or locally to a log file. Finally, one implementation detail is that each forecasting instance also receives part of the patterns. This is not necessary as it could receive both the MC and the DFSA from the training operator. However, to reduce data transferring, the necessary information to create the DFSA is passed to the CEF process through the initialization of the Map function.

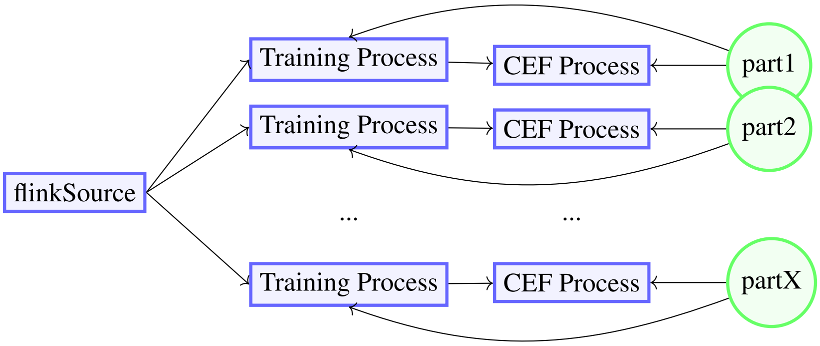


Figure 41: distributed forecasting and training

8.3 Evaluation

We have conducted two experiments to evaluate the efficiency of the distribution. We have conducted experiments on the maritime domain to evaluate the efficiency of the distribution. We have used a simple pattern of determining when a vessel is approaching the Brest port. That is, a match is detected when a vessel is more than 5km away from Brest for at least three events and then is less than 5km away for the following event. We have multiplied this pattern 48 times to test the execution time in a distributed environment.

We have used a single machine with 24 cores (2 threads per core), 264GB of memory and model name Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz. The stream is a subset of the Brest dataset and features 3,371,919 events. The experiment has been conducted a number of times with the parallelism set to one of 1, 2, 4, 8 and 16 each time. The patterns per worker node are 48, 24, 12, 6 and 3 respectively. Figure 43 showcases the results.

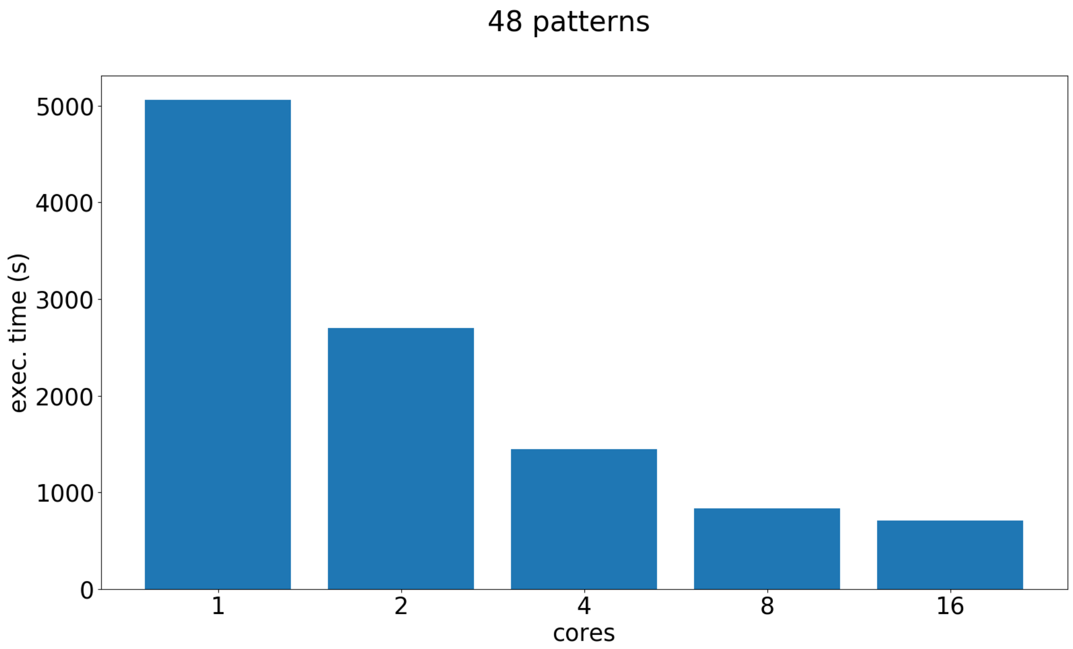


Figure 42: Execution time per workers used over the Brest dataset. There are 48 same patterns distributed among the worker nodes

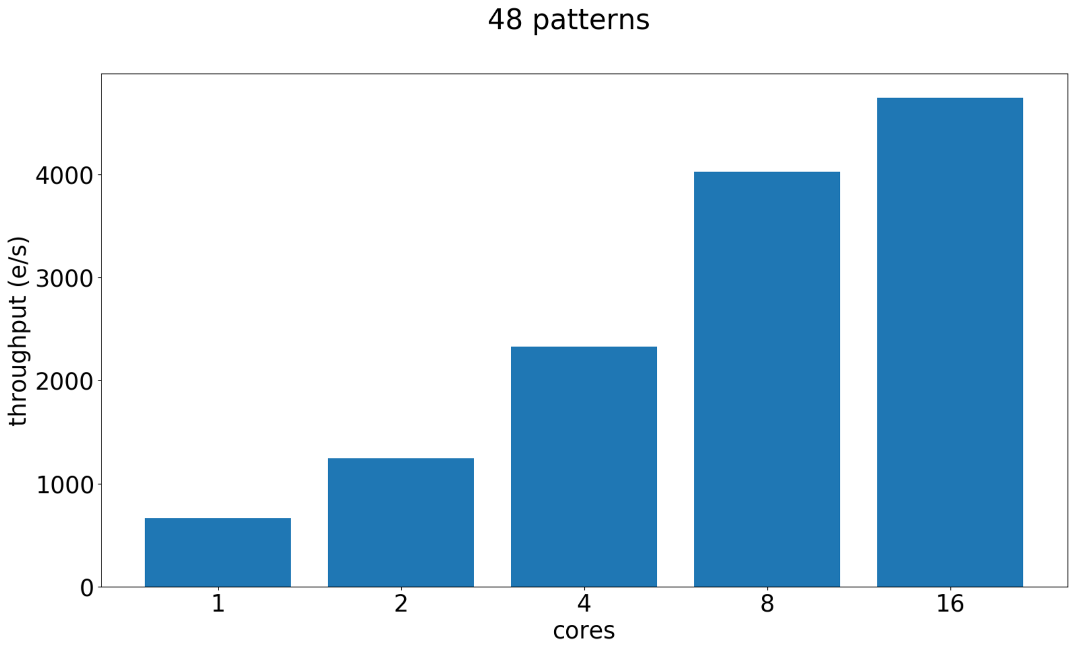


Figure 43: Throughput per workers used over the Brest dataset.

The Figure showcases great speed up from 1 to 16 cores (86% approximately). The speedup between 8 and 16 cores is insignificant (127 seconds). This could be attributed to either the replication of data (as the stream is broadcast to all workers) or to the small difference to the number of patterns per worker. That is, for 8 cores each node only has to deal with 3 patterns more.

For the second experiment we have used the same pattern multiplied 16 times. The dataset, this time, is taken from MarineTraffic and has undergone compression [116] retaining salient movement features only. It consists of 14,791,573 events, which is about 5 times the previous one in size.

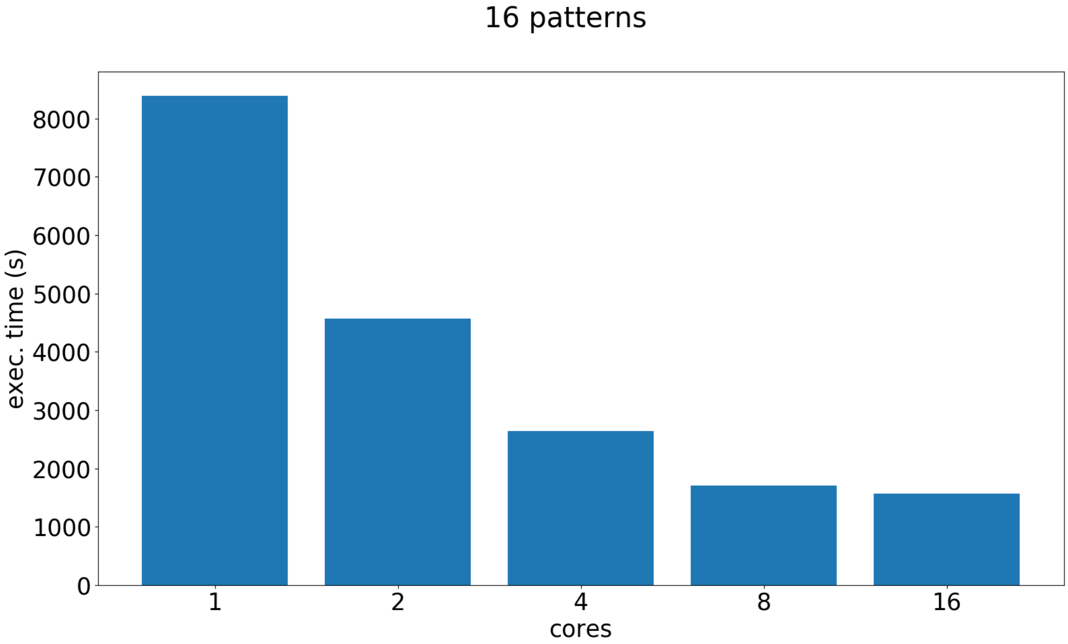


Figure 44: Execution time per workers used over the MarineTraffic dataset. There are 16 same patterns distributed among the worker nodes

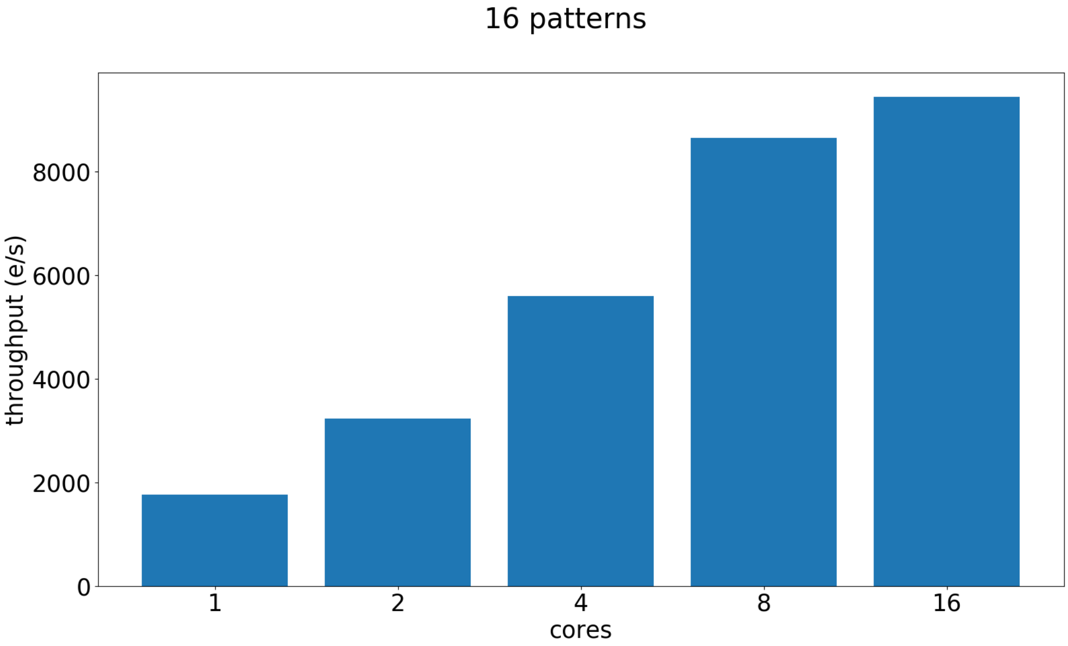


Figure 45: Throughput per workers used over the MarineTraffic dataset.

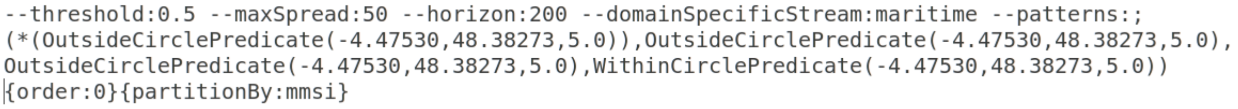
Similar results to the previous experiment are showcased in figure 45 with the speedup from 1 to 16 cores being approximately 81%. Again, we notice that the more cores we use the less gain we get. This notion is highlighted by the difference between 8 and 16 cores which is almost nonexistent. This makes sense, as with 16 cores each core has to work with 1 pattern, while with 8 cores with 2 patterns. That is, the difference in workload is very small.

One important detail is that execution times and throughput presented here are not calculated only for the forecasting phase. They also include time needed for training, producing the models every *k* events plus whatever extra overhead Flink asserts into the system. That is we just check how long a Flink job took to execute. We look to improve this in the future.

8.4 Integration with the INFORE Architecture

We mentioned earlier how the input and output of our module can either be local files or Kafka topics. For the purpose of integration to the Infore architecture we opt to use the Kafka topics. More specifically we use the ”DataTopic” and ”Forecast” as the input and output topics respectively, which are handled by the respective consumers and producers. The Kafka consumer is inside the Flink library as a FlinkKafkaConsumer flinkSource is used. For the producer, we used pure Kafka instead as it required less tweaking in the original code. That is, whenever a forecast is made, it is directly forwarded to the topic by a producer initialized inside the forecasting process.

The final topic is responsible for handling forecasting requests and is named ”ConfigTopic”. A consumer inside a loop is set up to listen to that topic and handles messages like the one shown in Figure 46. The value of the threshold, maxSpread and horizon for some forecast can be specified (e.g., −−threshold:<value>). The domainSpecificStream attribute specifies the type of the stream and is necessary for proper parsing of the data input messages.



```
--threshold:0.5 --maxSpread:50 --horizon:200 --domainSpecificStream:maritime --patterns:;
(*(OutsideCirclePredicate(-4.47530,48.38273,5.0)),OutsideCirclePredicate(-4.47530,48.38273,5.0),
OutsideCirclePredicate(-4.47530,48.38273,5.0),WithinCirclePredicate(-4.47530,48.38273,5.0))
{order:0}{partitionBy:mmsi}
```

Figure 46: Example of a forecasting request received from Kafka

Whenever a new request arrives it is parsed and then a forecasting module is fired asynchronously as a new process. The server then goes on to handle new incoming requests (if any). If a message arrives that is equal to ”terminateServer” the loop is broken and the consumer is closed, terminating the server process.

9 Progress Achieved Towards the INFORE objectives

The main objective that this deliverable addresses is the development of algorithms for real-time, interactive machine learning from massive, distributed data streams, as well as of techniques for complex event forecasting. Towards this objective, we have moved along the following directions:

- In Section 2, we presented an on-line interval-based probabilistic composite event recognition system. We have tested the proposed system on a surrogate dataset for human activity surveillance as a proof of concept for future use on maritime datasets.
- In Section 3 we presented an Answer Set Programming-based approach to online structure and weight learning of complex event patterns, that significantly improves previous work based on Markov Logic, both in terms of efficiency and of predictive performance. The latter is validated via experiments on the tasks of activity recognition, maritime surveillance and vehicle fleet management.
- In Section 4 we presented an improved approach to online semi-supervised learning of event rules, along experimental results on the maritime domain.
- In Section 5 we presented the OMLDM component of the INFORE platform, a high-performance engine for extreme scale mining of massive streams. We proposed an architecture following the parameter server pattern, which is easy to use and also easy to extend with new learning and mining algorithms. We discussed the current state of the implementation, focusing primarily on the algorithmic aspects, and described some of the implementation choices we made in developing it on top of Apache Flink.
- In Section 6 we presented two methods for exploring the parameter space of cancer simulations, one based on genetic algorithms and one based on random forests. In addition, towards learning patterns for complex event forecasting (as presented in Section 7), we introduce an approach for time-series discretization in order to create symbolic example sequences.
- In Section 7 we presented a complete theoretical framework for complex event forecasting. We additionally presented an initial implementation of this framework, along with experimental results from two domains.
- In Section 8 we made this initial implementation distributed and made its training phase online. We also made it capable of reading/writing from/to Kafka, as a first step towards its integration with the INFORE architecture. We showcased experiment results on the maritime domain regarding the efficiency of the distribution.

More concretely, WP6 has promised to deliver a fully expressive forecasting module that can accommodate any regular expression and has formal, probabilistic semantics. The work that we presented in Section 7 represents a significant step towards this direction. In Section 8 we have further expanded this framework by employing distribution techniques in order to be able to handle high-volume and high-velocity event streams. Besides the tasks of recognition and forecasting, we also showed the training/learning task of parameter estimation may be executed in a distributed manner. We intend to build upon this first step towards distributed complex event forecasting in order to develop a forecasting module that will be able to handle the data volumes of the INFORE use cases. With respect to Task 6.2 (Interactive, Online Learning and Data Mining), we presented a method for online structure and weight learning of complex event patterns that significantly improves previous work both in terms of efficiency and of predictive performance. We additionally presented an improved methods to online semi-supervised learning of event rules. Finally, we described a high-performance engine for extreme scale mining of massive streams, based on the parameter server. These methods will act as building blocks for the development of distributed learning methods in the remainder of the project.

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] Naoki Abe and Manfred K. Warmuth. On the computational complexity of approximating distributions by probabilistic automata. *Machine Learning*, 9:205–260, 1992.

[4] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *ACM Special Interest Group on Management of Data*, pages 147–160, 2008.

[5] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD Conference*, pages 207–216. ACM Press, 1993.

[6] Adnan Akbar, François Carrez, Klaus Moessner, and Ahmed Zoha. Predicting complex events for pro-active iot applications. In *WF-IoT*, pages 327–332. IEEE Computer Society, 2015.

[7] Massimiliano Albanese, Rama Chellappa, Naresh Cuntoor, Vincenzo Moscato, Antonio Picariello, V. S. Subrahmanian, and Octavian Udrea. PADS: A Probabilistic Activity Detection Framework for Video Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(12):2246–2261, 2010.

[8] Elias Alevizos, Alexander Artikis, and George Paliouras. Event forecasting with pattern markov chains. In *DEBS*, pages 146–157. ACM, 2017.

[9] Elias Alevizos, Alexander Artikis, and George Paliouras. Wayeb: a tool for complex event forecasting. In *LPAR*, volume 57 of *EPiC Series in Computing*, pages 26–35. EasyChair, 2018.

[10] Elias Alevizos, Alexander Artikis, and Georgios Paliouras. Symbolic automata with memory: a computational model for complex event processing. *CoRR*, abs/1804.09999, 2018.

[11] Elias Alevizos, Anastasios Skarlatidis, Alexander Artikis, and Georgios Paliouras. Probabilistic complex event recognition: A survey. *ACM Comput. Surv.*, 50(5):71:1–71:31, 2017.

[12] James F. Allen. Maintaining knowledge about temporal intervals. *Communication of the ACM*, 26(11):832–843, 1983.

[13] A. Artikis, C. Baber, Pedro Bizarro, C. Canudas-de-Wit, O. Etzion, F. Fournier, P. Goulart, A. Howes, J. Lygeros, G. Paliouras, A. Schuster, and I. Sharfman. Scalable proactive event-driven decision making. *IEEE Technol. Soc. Mag.*, 33(3):35–41, 2014.

[14] A. Artikis, M. Weidlich, A. Gal, V. Kalogeraki, and D. Gunopulos. Self-adaptive event recognition for intelligent transport management. In *Proceedings of Big Data*, pages 319–325. IEEE, 2013.

[15] Alexander Artikis, Nikos Katzouris, Ivo Correia, Chris Baber, Natan Morar, Inna Skarbovsky, Fabiana Fournier, and Georgios Paliouras. A prototype for credit card fraud management: Industry paper. In *DEBS*, pages 249–260. ACM, 2017.

[16] Alexander Artikis, Evangelos Makris, and Georgios Paliouras. A probabilistic interval-based event calculus for activity recognition. *Annals of Mathematics and Artificial Intelligence*, Aug 2019. `https://doi.org/10.1007/s10472-019-09664-4`.

[17] Alexander Artikis, Marek Sergot, and Georgios Paliouras. An event calculus for event recognition. *Knowledge and Data Engineering, IEEE Transactions on*, 27(4):895–908, 2015.

[18] Alexander Artikis, Marek J. Sergot, and Georgios Paliouras. An event calculus for event recognition. *IEEE Transactions on Knowledge and Data Engineering*, 27(4):895–908, 2015.

[19] Alexander Artikis, Anastasios Skarlatidis, François Portet, and Georgios Paliouras. Logic-based event recognition. *The Knowledge Engineering Review*, 27(04):469–506, 2012.

[20] Alexander Artikis, Anastasios Skarlatidis, François Portet, and Georgios Paliouras. Logic-based event recognition. *Knowledge Engineering Review*, 27(4):469–506, 2012.

[21] Duangtida Athakravi, Domenico Corapi, Krysia Broda, and Alessandra Russo. Learning through hypothesis refinement using answer set programming. In *Inductive Logic Programming*, pages 31–46. Springer, 2013.

[22] Ezio Bartocci and Radu Grosu. Monitoring with uncertainty. *Electronic Proceedings in Theoretical Computer Science*, 124, 08 2013.

[23] Ron Begleiter, Ran El-Yaniv, and Golan Yona. On prediction using variable order markov models. *J. Artif. Intell. Res.*, 22:385–421, 2004.

[24] Albert Bifet, Ricard Gavaldà, Geoff Holmes, and Bernhard Pfahringer. *Machine learning for data streams: with practical examples in MOA*. MIT Press, 2018.

[25] William Brendel, Alan Fern, and Sinisa Todorovic. Probabilistic event logic for interval-based event recognition. In *Proceedings of CVPR*, pages 3329–3336, 2011.

[26] Peter Bühlmann, Abraham J Wyner, et al. Variable length markov chains. *The Annals of Statistics*, 27(2):480–513, 1999.

[27] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[28] Olivier Chapelle, Bernhard Schlkopf, and Alexander Zien. *Semi-Supervised Learning*. MIT Press, 2006.

[29] Bo Chen, Hongwei Liu, Jing Chai, and Zheng Bao. Large margin feature weighting method via linear programming. *IEEE Trans. Knowl. Data Eng.*, 21(10):1475–1488, 2009.

[30] Cheng Siong Chin and Xi Ji. Adaptive online sequential extreme learning machine for frequency-dependent noise data on offshore oil rig. *Engineering Applications of Artificial Intelligence*, 74:226–241, September 2018.

[31] Chung-Wen Cho, Yi-Hung Wu, Show-Jane Yen, Ying Zheng, and Arbee L. P. Chen. On-line rule matching for event prediction. *VLDB J.*, 20(3):303–334, 2011.

[32] Maximilian Christ, Julian Krumeich, and Andreas W. Kempa-Liehr. Integrating predictive analytics into complex event processing by using conditional density estimations. In *EDOC Workshops*, pages 1–8. IEEE Computer Society, 2016.

[33] John G. Cleary and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. Communications*, 32(4):396–402, 1984.

[34] J. Cooley, P. Lewis, and P. Welch. The finite fourier transform. *IEEE Transactions on Audio and Electroacoustics*, 17:77–85, 6 1969.

[35] Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. Online Passive-Aggressive Algorithms. 7:551–585, 2006.

[36] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

[37] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, 2012.

[38] Loris D'Antoni and Margus Veanes. Extended symbolic finite automata and transducers. *Formal Methods in System Design*, 47(1):93–119, 2015.

[39] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In *CAV (1)*, volume 10426 of *Lecture Notes in Computer Science*, pages 47–67. Springer, 2017.

[40] Fabio Aurelio D'Asaro, Antonis Bikakis, Luke Dickens, and Rob Miller. Foundations for a probabilistic event calculus. In *Proceedings of LPNMR*, pages 57–63, 2017.

[41] Ingrid Daubechies. Orthonormal bases of compactly supported wavelets. *Communications on pure and applied mathematics*, pages 909–996, 10 1988.

[42] L. De Raedt, K. Kersting, S. Natarajan, and D. Poole. Statistical relational artificial intelligence: Logic, probability, and computation. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 10(2):1–189, 2016.

[43] Luc De Raedt. *Logical and relational learning*. Springer Science & Business Media, 2008.

[44] Pedro Domingos and Geoff Hulten. Mining High-Speed Data Streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pages 71–80, Boston, Massachusetts, USA, 2000. Association for Computing Machinery.

[45] Pedro M. Domingos and Geoff Hulten. Mining high-speed data streams. In *ACM SIGKDD*, pages 71–80, 2000.

[46] Florian Dörfler and Francesco Bullo. Kron reduction of graphs with applications to electrical networks. *IEEE Trans. on Circuits and Systems*, 60-I(1):150–163, 2013.

[47] C. Dousson and P. Le Maigat. Chronicle recognition improvement using temporal focusing and hierarchisation. In *Proceedings of IJCAI*, pages 324–329, 2007.

[48] Anton Dries and Luc De Raedt. Towards clausal discovery for stream mining. In *Proceedings of the 19th International Conference on Inductive Logic Programming (ILP)*, pages 9–16, 2009.

[49] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[50] Yagil Engel and Opher Etzion. Towards proactive event-driven computing. In *DEBS*, pages 125–136. ACM, 2011.

[51] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Company, 2010.

[52] R. Fagin, J. Halpern, and M. Vardi. What can machines know? On the properties of knowledge in distributed systems. *Journal of the ACM*, 39(2):328–376, 1992.

[53] Chiara Di Francescomarino, Chiara Ghidini, Fabrizio Maria Maggi, and Fredrik Milani. Predictive process monitoring methods: Which one suits me best? In *BPM*, volume 11080 of *Lecture Notes in Computer Science*, pages 462–479. Springer, 2018.

[54] James C Fu and WY Wendy Lou. *Distribution theory of runs and patterns and its applications: a finite Markov chain imbedding approach*. World Scientific, 2003.

[55] Lajos Jeno Fülöp, Árpád Beszédes, Gabriella Toth, Hunor Demeter, László Vidács, and Lóránt Farkas. Predictive complex event processing: a conceptual framework for combining complex event processing and predictive analytics. In *BCI*, pages 26–31. ACM, 2012.

[56] Avigdor Gal and Nicolo Rivetti. Uncertainty in streams. In *Encyclopedia of Big Data Technologies*. 2019.

[57] João Gama. *Knowledge Discovery from Data Streams*. Chapman and Hall / CRC Data Mining and Knowledge Discovery Series. CRC Press, 2010.

[58] João Gama, Raquel Sebastião, and Pedro Pereira Rodrigues. On evaluating stream learning algorithms. *Machine Learning*, 90(3):317–346, March 2013.

[59] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[60] Lise Getoor and Ben Taskar. *Introduction to Statistical Relational Learning*. MIT Press, 2007.

[61] Ahmadreza Ghaffarizadeh, Randy Heiland, Samuel H. Friedman, Shannon M. Mumenthaler, and Paul Macklin. Physicell: An open source physics-based cell simulator for 3-d multicellular systems. In *PLoS Computational Biology*, 2018.

[62] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos N. Garofalakis. Complex event recognition in the big data era: a survey. *VLDB J.*, 29(1):313–352, 2020.

[63] Matthew L. Ginsberg. Multivalued logics: a uniform approach to reasoning in artificial intelligence. *Computational Intelligence*, 4:265–316, 1988.

[64] Tilmann Gneiting and Adrian E Raftery. Strictly proper scoring rules, prediction, and estimation. *Journal of the American Statistical Association*, 102(477):359–378, 2007.

[65] Alejandro Grez, Cristian Riveros, and Martín Ugarte. A formal framework for complex event processing. In *ICDT*, volume 127 of *LIPIcs*, pages 5:1–5:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

[66] Victor Guimarães, Aline Paes, and Gerson Zaverucha. Online probabilistic theory revision from examples with proppr. *Mach. Learn.*, 108(7):1165–1189, 2019.

[67] Ulrich Hedtstück. *Complex event processing: Verarbeitung von Ereignismustern in Datenströmen*. Springer Vieweg, Berlin, 2017.

[68] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, pages 278–282, Montreal, Quebec, Canada, 1995. IEEE.

[69] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

[70] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: Theory and applications. *Neurocomputing*, 70(1):489–501, December 2006.

[71] Tuyen N Huynh and Raymond J Mooney. Max-margin weight learning for markov logic networks. In *ECML-2009*, pages 564–579. Springer, 2009.

[72] Tuyen N Huynh and Raymond J Mooney. Online max-margin weight learning for markov logic networks. In *SDM*, pages 642–651. SIAM, 2011.

[73] Koray Inki, Ismail Ari, and Hasan Sözer. Runtime verification of iot systems using complex event processing. *Proceedings of ICNSC*, pages 625–630, 2017.

[74] Yogi Joshi, Guy Martin Tchamgoue, and Sebastian Fischmeister. Runtime verification of ltl on lossy traces. In *Proceedings of SAC*, page 13791386. ACM, 2017.

[75] Michael Kamp, Linara Adilova, Joachim Sicking, Fabian Hüger, Peter Schlicht, Tim Wirtz, and Stefan Wrobel. Efficient decentralized deep learning by dynamic model averaging. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 393–409. Springer, 2018.

[76] Michael Kamp, Mario Boley, Daniel Keren, Assaf Schuster, and Izchak Sharfman. Communication-efficient distributed online prediction by dynamic model synchronization. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 623–639. Springer, 2014.

[77] Michael Kamp, Mario Boley, Olana Missura, and Thomas Gärtner. Effective parallelisation for machine learning. In *Advances in Neural Information Processing Systems*, pages 6477–6488, 2017.

[78] Nikos Katzouris, Alexander Artikis, and Georgios Paliouras. Incremental learning of event definitions with inductive logic programming. *Machine Learning*, 100(2-3):555–585, 2015.

[79] Nikos Katzouris, Alexander Artikis, and Georgios Paliouras. Online learning of event definitions. *TPLP*, 16(5-6):817–833, 2016.

[80] Nikos Katzouris, Evangelos Michelioudakis, Alexander Artikis, and Georgios Paliouras. Online learning of weighted relational rules for complex event recognition. In *ECML-PKDD 2018*, pages 396–413, 2018.

[81] Nikos Katzouris, Evangelos Michelioudakis, Alexander Artikis, and Georgios Paliouras. Online learning of weighted relational rules for complex event recognition. In *Proceedings of European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 396–413, 2018.

[82] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and Information Systems*, 3, 01 2002.

[83] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[84] A. Kimmig, B. Demoen, L. De Raedt, V. Santos Costa, and R. Rocha. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming*, 11:235–262, 2011.

[85] Nicolas Kourtellis, Gianmarco De Francisci Morales, and Albert Bifet. Large-Scale Learning from Data Streams with Apache SAMOA. In Moamar Sayed-Mouchaweh, editor, *Learning from Data Streams in Evolving Environments: Methods and Applications*, Studies in Big Data, pages 177–207. Springer International Publishing, Cham, 2019.

[86] Robert A. Kowalski and Marek J. Sergot. A logic-based calculus of events. *New Generation Comput.*, 4(1):67–95, 1986.

[87] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[88] Mark Law, Alessandra Russo, and Krysia Broda. Inductive learning of answer set programs from noisy examples. *Advances in Cognitive Systems*, 2018.

[89] Srivatsan Laxman, Vikram Tankasali, and Ryen W. White. Stream prediction using a generative model based on frequent episodes in event sequences. In *KDD*, pages 453–461. ACM, 2008.

[90] Joohyung Lee, Samidh Talsania, and Yi Wang. Computing LPMLN using ASP and MLN solvers. *Theory Pract. Log. Program.*, 17(5-6):942–960, 2017.

[91] Joohyung Lee and Yi Wang. Weighted rules under the stable model semantics. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR*, pages 145–154. AAAI Press, 2016.

[92] Gaelle Letort, Arnau Montagud, Gautier Stoll, Randy Heiland, Emmanuel Barillot, Paul Macklin, Andrei Zinovyev, and Laurence Calzone. Physiboss: a multi-scale agent-based modelling framework integrating physical dimension and cell signalling. *Bioinformatics*, pages 1188–1196, 04 2019.

[93] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598. USENIX Association, 2014.

[94] Jessica Lin, Eamonn Keogh, Li Wei, and Stefano Lonardi. Experiencing sax: A novel symbolic representation of time series. *Data Min. Knowl. Discov.*, 15:107–144, 08 2007.

[95] T. List, J. Bins, J. Vazquez, and R. B. Fisher. Performance evaluating the evaluator. In *Proceedings of International Conference on Computer Communications and Networks*, pages 129–136. IEEE Computer Society, 2005.

[96] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM)*, pages 413–422, 2008.

[97] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2001.

[98] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov.*, 1(3):259–289, 1997.

[99] Alfonso Eduardo Márquez-Chamorro, Manuel Resinas, and Antonio Ruiz-Cortés. Predictive monitoring of business processes: A survey. *IEEE Trans. Services Computing*, 11(6):962–977, 2018.

[100] E. Michelioudakis, A. Skarlatidis, G. Paliouras, and A. Artikis. Online structure learning using background knowledge axiomatization. In *Proceedings of European Conference on Machine Learning and Knowledge Discovery in Databases*, volume 1, pages 242–237, 2016.

[101] E. Michelioudakis, A. Skarlatidis, G. Paliouras, and A. Artikis. Osla: Online structure learning using background knowledge axiomatization. In *ECML*, pages 232–247. Springer, 2016.

[102] Evangelos Michelioudakis, Alexander Artikis, and Georgios Paliouras. Semi-supervised online structure learning for composite event recognition. *Machine Learning*, 108(7):1085–1110, 2019.

[103] Evangelos Michelioudakis, Alexander Artikis, and Georgios Paliouras. Semi-supervised online structure learning for composite event recognition. *Machine Learning*, 108(7):1085–1110, 2019.

[104] Evangelos Michelioudakis, Anastasios Skarlatidis, Georgios Paliouras, and Alexander Artikis. OSLα: Online structure learning using background knowledge axiomatization. In *Proceedings of ECML-PKDD*, pages 232–247, 2016.

[105] Douglas C Montgomery, Cheryl L Jennings, and Murat Kulahci. *Introduction to time series analysis and forecasting*. John Wiley & Sons, 2015.

[106] Vlad I. Morariu and Larry S. Davis. Multi-agent event recognition in structured scenarios. In *Proceedings of CVPR*, pages 3289–3296, 2011.

[107] E. Mueller. Event calculus and temporal action logics compared. *Artificial Intelligence*, 170(11):1017–1029, 2006.

[108] Erik T. Mueller. *Commonsense Reasoning*. Morgan Kaufmann, 2006.

[109] Erik T Mueller. *Commonsense reasoning: an event calculus based approach*. Morgan Kaufmann, 2014.

[110] Vinod Muthusamy, Haifeng Liu, and Hans-Arno Jacobsen. Predictive publish/subscribe matching. In *DEBS*, pages 14–25. ACM, 2010.

[111] Shan-Hwei Nienhuys-Cheng. Distance Between Herbrand Interpretations: A Measure for Approximations to a Target Concept. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, pages 213–226. Springer-Verlag, 1997.

[112] J. Ozik, Nicholson Collier, Randy Heiland, Gary An, and Paul Macklin. Learning-accelerated discovery of immune-tumour interactions. *Molecular Systems Design and Engineering*, 4:747–760, 08 2019.

[113] Suraj Pandey, Surya Nepal, and Shiping Chen. A test-bed for the evaluation of business process prediction techniques. In *CollaborateCom*, pages 382–391. ICST / IEEE, 2011.

[114] K Patroumpas, D Spirelis, E Chondrodima, H Georgiou, Petrou P, Tampakis P, Sideridis S, Pelekis N, and Theodoridis Y. Final dataset of Trajectory Synopses over AIS kinematic messages in Brest area (ver. 0.8) [Data set], 10.5281/zenodo.2563256, 2018.

[115] Kostas Patroumpas, Elias Alevizos, Alexander Artikis, Marios Vodas, Nikos Pelekis, and Yannis Theodoridis. Online event recognition from moving vessel trajectories. *GeoInformatica*, 21(2):389–427, 2017.

[116] Kostas Patroumpas, Elias Alevizos, Alexander Artikis, Marios Vodas, Nikos Pelekis, and Yannis Theodoridis. Online event recognition from moving vessel trajectories. *GeoInformatica*, 21(2):389–427, 2017.

[117] Kostas Patroumpas, Elias Alevizos, Alexander Artikis, Marios Vodas, Nikos Pelekis, and Yannis Theodoridis. Online event recognition from moving vessel trajectories. *GeoInformatica*, 21(2):389–427, 2017.

[118] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[119] Manolis Pitsikalis, Alexander Artikis, Richard Dreo, Cyril Ray, Elena Camossi, and Anne-Laure Jousselme. Composite event recognition for maritime monitoring. In *Proceedings of DEBS*, pages 163–174, 2019.

[120] Gordon D. Plotkin. *Automatic Methods of Inductive Inference*. PhD thesis, Edinburgh University, 1971.

[121] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[122] Luc De Raedt. *Logical and Relational Learning: From ILP to MRDM (Cognitive Technologies)*. Springer, 2008.

[123] Cyril RAY, Richard DRO, Elena CAMOSSI, and Anne-Laure JOUSSELME. Heterogeneous Integrated Dataset for Maritime Intelligence, Surveillance, and Reconnaissance, February 2018.

[124] Oliver Ray. Nonmonotonic abductive inductive learning. *Journal of Applied Logic*, 7(3):329–340, 2009.

[125] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1–2):107–136, 2006.

[126] Dana Ron, Yoram Singer, and Naftali Tishby. The power of amnesia. In *NIPS*, pages 176–183. Morgan Kaufmann, 1993.

[127] Dana Ron, Yoram Singer, and Naftali Tishby. The power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning*, 25(2-3):117–149, 1996.

[128] Adam Sadilek and Henry A. Kautz. Location-Based Reasoning about Complex Multi-Agent Behavior. *Journal of Artificial Intelligence Research (JAIR)*, 43:87–133, 2012.

[129] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.

[130] Vasilis Samoladas and Minos Garofalakis. Functional Geometric Monitoring for Distributed Streams. In *EDBT2019*, Lisbon, Portugal.

[131] Joseph Selman, Mohamed R. Amer, Alan Fern, and Sinisa Todorovic. PEL-CNF: Probabilistic event logic conjunctive normal form for video interpretation. In *Proceedings of ICCVW*, pages 680–687. IEEE, 2011.

[132] Vinay D. Shet, Jan Neumann, Visvanathan Ramesh, and Larry S. Davis. Bilattice-based logical reasoning for human detection. *Proceedings of CVPR*, pages 1–8, 2007.

[133] Jeffrey Mark Siskind. Grounding the lexical semantics of verbs in visual perception using force dynamics and event logic. *The Journal of Artificial Intelligence Research (JAIR)*, 15:31–90, 2001.

[134] Anastasios Skarlatidis, Alexander Artikis, Jason Filipou, and Georgios Paliouras. A probabilistic logic programming event calculus. *Theory and Practice of Logic Programming*, 15(2):213–245, 2015.

[135] Anastasios Skarlatidis and Evangelos Michelioudakis. Logical Markov Random Fields (LoMRF): an open-source implementation of Markov Logic Networks, 2014.

[136] Anastasios Skarlatidis, Georgios Paliouras, Alexander Artikis, and George A. Vouros. Probabilistic event calculus for event recognition. *ACM Transactions on Computational Logic*, 16(2), 2015.

[137] Anastasios Skarlatidis, Georgios Paliouras, Alexander Artikis, and George A Vouros. Probabilistic event calculus for event recognition. *ACM Transactions on Computational Logic (TOCL)*, 16(2):11, 2015.

[138] Ashwin Srinivasan and Michael Bain. An empirical study of on-line models for relational data streams. *Machine Learning*, 106(2):243–276, 2017.

[139] Gautier Stoll, Barthelemy Caron, Eric Viara, Aurelien Dugourd, Andrei Zinovyev, Aurlien Naldi, Guido Kroemer, Emmanuel Barillot, and Laurence Calzone. Maboss 2.0: an environment for stochastic boolean modeling. *Bioinformatics*, 1-3, 03 2017.

[140] Kai Ming Ting, Guang-Tong Zhou, Fei Tony Liu, and Swee Chuan Tan. Mass estimation. *Machine Learning*, 90(1):127–160, 2013.

[141] Kai Ming Ting, Ye Zhu, Mark J. Carman, Yue Zhu, Takashi Washio, and Zhi-Hua Zhou. Lowest probability mass neighbour algorithms: relaxing the metric constraint in distance-based neighbourhood algorithms. *Machine Learning*, 108(2):331–376, 2019.

[142] Wil Van Der Aalst. *Process mining: discovery, conformance and enhancement of business processes*. Springer, 2011.

[143] Wil M. P. van der Aalst, M. H. Schonenberg, and Minseok Song. Time prediction based on process mining. *Inf. Syst.*, 36(2):450–475, 2011.

[144] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *ICST*, pages 498–507. IEEE Computer Society, 2010.

[145] Ricardo Vilalta and Sheng Ma. Predicting rare events in temporal domains. In *ICDM*, pages 474–481. IEEE Computer Society, 2002.

[146] Tal Wagner, Sudipto Guha, Shiva Prasad Kasiviswanathan, and Nina Mishra. Semi-supervised learning on data streams via temporal label propagation. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 5082–5091, 2018.

[147] Jue Wang and Pedro Domingos. Hybrid Markov Logic Networks. In *Proceedings of AAAI*, pages 1106–1111. AAAI Press, 2008.

[148] Kilian Q. Weinberger and Lawrence K. Saul. Distance metric learning for large margin nearest neighbor classification. *J. Mach. Learn. Res.*, 10:207–244, 2009.

[149] Frans M. J. Willems, Yuri M. Shtarkov, and Tjalling J. Tjalkens. The context-tree weighting method: basic properties. *IEEE Trans. Information Theory*, 41(3):653–664, 1995.

[150] Fuzhen Zhang. *The Schur Complement and Its Applications*. Springer, 2005.

[151] Haopeng Zhang, Yanlei Diao, and Neil Immerman. Recognizing patterns in streams with imprecise timestamps. *Proceedings of VLDB*, 3(1-2):244–255, 2010.

[152] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: A New Data Clustering Algorithm and Its Applications. *Data Mining and Knowledge Discovery*, 1(2):141–182, June 1997.

[153] Cheng Zhou, Boris Cule, and Bart Goethals. A pattern based predictor for event streams. *Expert Syst. Appl.*, 42(23):9294–9306, 2015.

[154] Xiaojin Zhu, Zoubin Ghahramani, and John D. Lafferty. Semi-supervised learning using gaussian fields and harmonic functions. In *Proceedings of the 20th International Conference on Machine Learning*, pages 912–919. AAAI Press, 2003.

[155] Xiaojin Zhu, Andrew B. Goldberg, Ronald Brachman, and Thomas Dietterich. *Introduction to Semi-Supervised Learning*. Morgan and Claypool Publishers, 2009.

[156] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory*, 24(5):530–536, 1978.