Offline and Online Learning of Signal Temporal Logic Formulae Using Decision Trees

GIUSEPPE BOMBARA and CALIN BELTA, Boston University, USA

In this article, we focus on inferring high-level descriptions of a system from its execution traces. Specifically, we consider a classification problem where system behaviors are described using formulae of *Signal Temporal Logic* (STL). Given a finite set of pairs of system traces and labels, where each label indicates whether the corresponding trace exhibits some system property, we devised a decision-tree-based framework that outputs an STL formula that can distinguish the traces. We also extend this approach to the online learning scenario. In this setting, it is assumed that new signals may arrive over time and the previously inferred formula should be updated to accommodate the new data. The proposed approach presents some advantages over traditional machine learning classifiers. In particular, the produced formulae are interpretable and can be used in other phases of the system's operation, such as monitoring and control. We present two case studies to illustrate the effectiveness of the proposed algorithms: (1) a fault detection problem in an automotive system and (2) an anomaly detection problem in a maritime environment.

CCS Concepts: • **Computing methodologies** → *Logical and relational learning*; **Classification and regression trees**; • **Theory of computation** → *Modal and temporal logics*;

Additional Key Words and Phrases: Signal temporal logic, logic inference, specification mining, formal methods, decision trees, impurity measure, classification, supervised learning, anomaly detection, online learning

ACM Reference format:
Giuseppe Bombara and Calin Belta. 2021. Offline and Online Learning of Signal Temporal Logic Formulae Using Decision Trees. *ACM Trans. Cyber-Phys. Syst.* 5, 3, Article 22 (March 2021), 23 pages.
https://doi.org/10.1145/3433994

1 INTRODUCTION

In recent years, there has been a cross-fertilization between the fields of machine learning and formal methods. For example, formal verification techniques have been applied to provide guarantees on the behavior of machine learning components, such as neural networks [30]. In this article, we focus on *learning* high-level descriptions of a system from its execution traces. The system operation is described using Signal Temporal Logic (STL), a specification language used in the field of formal methods to define the behaviors of dynamical systems [14]. The inferred formulae can be employed directly for classification or, more generally, for monitoring and controlling

This work was partially supported by DENSO CORPORATION, and by the NSF under Grants No. CNS-1446607, No. CBET-0939511, and No. NSF IIS-1723995.
Authors' addresses: G. Bombara, Boston University, 8 Saint Mary's Street, Boston, MA 02215; email: gbombara@bu.edu; C. Belta, Boston University, 110 Cummington Mall, Boston, MA 02215; email: cbelta@bu.edu.

2378-962X/2021/03-ART22 $15.00
https://doi.org/10.1145/3433994

the system. This approach, while retaining many qualities of traditional classifiers, addresses some of the limitations of current formalisms. First, as opposed to most classifiers, STL formulae have precise meaning and allow for a rich specification of the behaviors that is *interpretable* by humans experts. Second, machine learning methods commonly applied to time series data are either model-based, i.e., they require a *good* model of the system under analysis [22], or based on black-box models, such as deep neural networks [23]. Third, classical machine learning methods are often overly specific to the task. That is, they focus exclusively on solving the problem at hand but offer no other insight on the system where they have been applied (*knowledge discovery*).

In this article, we first focus on the so-called *two-class classification problem*. In this setting, our goal is to build a temporal logic formula that can distinguish traces belonging to one of two possible classes. The dataset is given as a finite set of pairs of system traces, also called *signals*, and *labels*. Each label indicates whether the respective trace exhibits some desired system behavior, e.g., an engine is working correctly (*supervised learning*). To construct a discriminating formula, we propose a novel, decision-tree-based framework. In this approach, each node of the tree contains a test associated with the satisfaction of a simple formula, optimally tuned from a predefined set of primitive formulae. Our framework produces a binary tree that can be translated to an equivalent STL formula and used for classification purposes. We refer to our approach as *framework*, because we are not just proposing a single algorithm but a family of algorithms.

Later, we turn our attention to the *online learning problem*. In this scenario, it is assumed that new data arrives over time and the inference system should be updated to accommodate it. This is in contrast with the classical (or offline) scenario, where only a single batch of data is available at the beginning and no further data can be considered. The online learning approach presents some major advantages. First, it provides a formula early during the signal collection process and then can refine it progressively when more signals become available. Second, it removes the usual separation between the building phase and the deployment phase of classifiers. The key insight we use to solve this problem *efficiently* is to create a new node in the decision tree only when we can be *reasonably* sure that the decision made holds for future data [12]. This is achieved through a probabilistic assessment among the possible options available for a node.

The offline and online algorithms were first introduced in References [6] and [5], respectively. In this article, we refine and extend the previous results by (1) utilizing a new optimization algorithm for the node construction (at the core of both algorithms), (2) performing a comprehensive evaluation of the various proposed impurity measures and primitive sets, (3) introducing a post-completion pruning procedure for the batch algorithm (to output simpler and more interpretable formulae), and (4) comparing and contrasting our work with recent related approaches. Moreover, the software tool was improved and is available online.[1]

We present two case studies to illustrate the effectiveness of the framework. The first is an anomaly detection problem in a maritime environment. The second is a fault detection problem in an automotive powertrain system. Comparisons with some related work are also included.

2 RELATED WORK

In this section, we focus on papers related to learning STL formulae from data. Two major areas can be identified.

The first area is concerned with finding the optimal parameters for a formula when a formula structure is given [1, 2, 9, 19, 24, 26]. That is, a designer provides a formula template such as "The engine speed settles below v m/s within τ seconds" and an optimization procedure finds values for v and τ. The given structure reflects the domain knowledge of the designer on the system and its

[1] LoTuS Toolbox—http://sites.bu.edu/hyness/lotus/.

properties of interest. This problem is called *parameter mining*, and the parameters for the formula are selected so that the resulting formula barely satisfies the input signals [19, 26], or strongly satisfies them [2] (in the sense of the robustness degree). These approaches essentially differ in the way the underlying objective function is formulated and the optimization strategy employed. It is worth mentioning that in References [2, 9, 19, 26], the parameter optimization problem is cast within a more general active learning framework, where the original system is queried for new signals if deemed necessary.

The second area tackles the *supervised two-class classification problem* and the goal is to construct a formula, *both* structure and parameters, that can *distinguish* between two sets of signals [3, 28, 34]. In Reference [28], the authors first defined a fragment of STL, called inference parametric signal temporal logic (iPSTL), and showed that this fragment admits a partial order among formulae, in the sense of language inclusion, and with respect to the robustness degree. This implies that iPSTL formulae can be organized in an infinite directed acyclic graph (DAG) capturing their ordering. This result is used to formulate the classification problem as an optimization problem, whose objective function involves the robustness degree, and solve it in two cyclic steps: (1) optimize the formula structure by exploring the DAG, pruning and growing it, and (2) optimize the formula parameters, for a fixed structure, using a nonlinear optimization algorithm. This approach presents two major limitations. First, the parameter optimization routine has a high computational cost. This is due to its nonlinear nature. Finding the optimal valuation becomes more and more challenging as the algorithm proceeds, because the dimension of the parameter space grows at each iteration. Second, the DAG is built using an ordering on the language accepted by PSTL formulae. This has adverse effects on performance. Specifically, the algorithm aims to optimize for the overall formula structure, i.e., for all valuations the structure should be good, which is too conservative. References [3, 8] also tackled the two-class problem. Their approach can be divided into two separate steps. First, they build two generative models, one for each class. The models have to be in the form of stochastic systems and are used to compute the probability of satisfaction of a formula. Second, a discriminative formula is obtained by searching a formula that maximizes the odds of being true for the first model and false for the other model. As with other approaches, the formula structure and parameters are optimized separately. In particular, the formula structure is constructed through heuristics [3] or with a genetic algorithm [8], whereas the parameter space is explored through statistical model checking. This approach presents some disadvantages. Primarily, it needs to build models of the system under analysis. This requires a domain expert and a certain amount of data. The parameter optimization process, based on extensive simulations of the models, is expensive. Nenzi et al. [34] proposed another approach based on genetic algorithms. Here, however, the structure construction uses primitives and the parameter mining uses a Gaussian Process-based optimization scheme that attempts to maximize the gap between robustness of normal traces and robustness of anomalous traces.

To conclude, there is a considerable amount of research on Boolean Logic (for an overview, see [27]), and some recent efforts on mining Linear Temporal Logic (LTL) formulae [33] and mining timed regular expressions (TRE) [32]. [16] used a learning procedure for formulae defined in particular spatial superposition logic. This logic was developed for describing patterns in images without a time component. Every image is represented with a multi-resolution format using a fixed height quad-tree data structure (which should not be confused with a decision tree).

3 SIGNAL TEMPORAL LOGIC

A temporal logic is a system of rules and symbols used for reasoning about propositions whose truth values change over time. LTLand Computation Tree Logic (CTL) are the most commonly used temporal logics [10]. Signal Temporal Logic (STL) has emerged recently as a generalization

of LTL, where time is continuous and the predicates are defined over real values [31]. STL has found significant applications in formal verification of hybrid systems where it is used to state and monitor requirements. In this section, we briefly review the syntax and the semantics of this logic.

Let $\mathbb{R}$ be the set of real numbers. For $t \in \mathbb{R}$, we denote the interval $[t, \infty)$ by $\mathbb{R}_{\geq t}$. We use $\mathcal{S} = \{s : \mathbb{R}_{\geq 0} \to \mathbb{R}^n\}$ with $n \in \mathbb{N}$ to denote the set of all continuous parameterized curves in the n-dimensional Euclidean space $\mathbb{R}^n$. In this article, an element of $\mathcal{S}$ is called a *signal* and its parameter is interpreted as *time*. Given a signal s, the components of s are denoted by s_i, $i \in \{1, \ldots, n\}$. The set $\mathcal{F}$ contains the projection operators from a signal s to one of its components s_i, that is $\mathcal{F} = \{f_i : \mathbb{R}^n \to \mathbb{R}, f_i(s) = s_i, i = \{1, \ldots, n\}\}$.[2] The *suffix* at time $t \geq 0$ of a signal is denoted by $s[t] \in \mathcal{S}$, and it represents the signal s shifted forward in time by t time units, i.e., $s[t](\tau) = s(\tau + t)$ for all $\tau \in \mathbb{R}_{\geq 0}$.

The syntax of STL is defined as follows [31]:

$$\phi ::= \top \mid f(x) \sim \mu \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \mathbf{U}_{[a,b)} \phi_2,$$

where $\top$ is the Boolean *true* constant ($\bot$ for *false*); $f(x) \sim \mu$ is a predicate over $\mathbb{R}^n$ defined by a function $f \in \mathcal{F}$, a real number $\mu \in \mathbb{R}$, and an order relation $\sim \in \{\leq, >\}$; $\neg$ and $\wedge$ are the Boolean operators negation and conjunction; and $\mathbf{U}_{[a,b)}$ is the bounded temporal operator *until*.

The semantics of STL is defined over signals in $\mathcal{S}$ as [31]

$$\begin{aligned}
s[t] &\models \top && \Leftrightarrow \ \top, \\
s[t] &\models f(x) \sim \mu && \Leftrightarrow \ f(s(t)) \sim \mu, \\
s[t] &\models \neg\phi && \Leftrightarrow \ \neg(s[t] \models \phi), \\
s[t] &\models (\phi_1 \wedge \phi_2) && \Leftrightarrow \ (s[t] \models \phi_1) \wedge (s[t] \models \phi_2), \\
s[t] &\models (\phi_1 \mathbf{U}_{[a,b)} \phi_2) && \Leftrightarrow \ \exists t_u \in [t + a, t + b) \text{ s.t. } \Big(s[t_u] \models \phi_2\Big) \\
& && \qquad \wedge \Big(\forall t_1 \in [t, t_u)\ s[t_1] \models \phi_1\Big).
\end{aligned}$$

A signal $s \in \mathcal{S}$ is said to satisfy an STL formula ϕ if and only if $s[0] \models \phi$. Other Boolean operations, such as disjunction, implication, and equivalence, are defined in the usual way. The temporal operators *eventually* and *globally* are defined, respectively, as

$$\mathbf{F}_{[a,b)}\phi \equiv \top \mathbf{U}_{[a,b)} \phi\,, \qquad \mathbf{G}_{[a,b)}\phi \equiv \neg \mathbf{F}_{[a,b)} \neg\phi.$$

In addition to the Boolean semantics defined above, some *quantitative semantics* have been proposed for STL [14, 15]. These semantics are formalized through the introduction of a real valued function called *robustness*, which quantifies the *degree* of satisfaction of a signal with respect to a formula.

Proposition 3.1. *Let $s \in \mathcal{S}$ be a signal and ϕ an STL formula such that $r(s, \phi) > 0$. All signals $s' \in \mathcal{S}$ such that $\|s - s'\|_\infty < r(s, \phi)$ satisfy the formula ϕ, i.e., $s' \models \phi$.*

Parametric Signal Temporal Logic (PSTL) was introduced in Reference [1] as an extension of STL where formulae are parameterized. A PSTL formula is similar to an STL formula, however all the time bounds in the time intervals associated with the temporal operators and all the constants in the inequality predicates are replaced by free parameters. These two types of parameters are called *time* and *space* parameters, respectively. If ψ is a PSTL formula, then every parameter assignment $\theta \in \Theta$ (where Θ is the parameter space of ψ) induces a corresponding STL formula $\phi = \psi(\theta)$, where all the space and time parameters of ψ have been fixed according to θ. This assignment is also

[2] A more general definition of the set $\mathcal{F}$ is used in Reference [31].

referred to as valuation θ of ψ. For example, given $\psi = \mathbf{F}_{[a,b)}(f_1(x) > \pi)$ and $\theta = [1.1, 2.3, 3.7]$, we obtain the STL formula $\psi(\theta) = \mathbf{F}_{[1.1,2.3)}(f_1(x) > 3.7)$.

Even though STL is defined using a dense-time semantics and natively supports predicates over reals, in practice its monitoring algorithms work with *sampled* data and assume that the signals are piece-wise constant (or piece-wise linearly interpolated) [13]. The sampling rate does not have to be constant.

4 SIGNAL CLASSIFICATION

4.1 Problem Formulation

We want to find an STL formula that separates traces produced by a system that exhibit some desired property, such as behaving correctly, from other traces of the same system. The normal working conditions are often referred to as *targets*, or *positives*, whereas the non-conforming patterns are usually referred to as *anomalies*, or *negatives*. Let $C = \{C_p, C_n\}$ be the set of classes, with C_p standing for the positive class and C_n for the negative class. Let $s^i \in \mathcal{S}$ be an n-dimensional signal, and let $l^i \in C$ be its label. We consider the following problem:

Problem 1 (Two-Class Classification). *Given a dataset of labeled signals $S_{\text{ds}} = \{(s^i, l^i)\}_{i=1}^N$, we want to find an STL formula ϕ^* such that the misclassification rate* $\text{MCR}(\phi, S_{\text{ds}})$ *is minimized, where the misclassification rate is defined as*

$$\text{MCR}(\phi, S_{\text{ds}}) := \frac{\left|\left\{s^i \mid (s^i \models \phi,\ l^i = C_n) \text{ or } (s^i \not\models \phi,\ l^i = C_p)\right\}\right|}{|S_{\text{ds}}|}.$$

In the above formula, $(s^i \models \phi,\ l^i = C_n)$ represents a *false positive*, while $(s^i \not\models \phi,\ l^i = C_p)$ represents a *false negative*.

4.2 STL Formulae and Decision Trees

Our key insight to tackle Problem 1 is that it is possible to build a map between a fragment of STL and decision trees. Consequently, we can exploit and extend the decision tree learning literature [7, 35, 36] to build a decision tree that classifies signals and map the constructed tree to an STL formula.

A decision tree is a tree-structured sequence of questions about the data used to make predictions about the data's labels. In a tree, we define: the root as the initial node; the depth of a node as the length of the path from the root to that node; the parent of a node as the neighbor whose depth is one less; the children of a node as the neighbors whose depths are one more. A node with no children is called a leaf, all other nodes are called non-terminal nodes. We focus on *binary* trees, where every non-terminal node splits the data in two children nodes and every leaf predicts a class.

Unfortunately, the space of all possible decision trees for a given classification problem is very large, and it is known that the problem of learning the optimal decision tree is NP-complete [20]. Most decision-tree learning algorithms are based on *greedy* approaches, where locally optimal decisions are taken at each node. These greedy induction algorithms can be stated in a simple recursive fashion, starting from the root node, and require two core components: (1) a list of possible ways to split the signals reaching a node; and (2) an optimality criterion to select the best split. Several learning algorithms can be created by selecting different components, called here *meta-parameters*. That is, once the user fixes the meta-parameters, a specific algorithm is *instantiated*. We propose to split the signals using a simple formula at each node, chosen from a finite set of PSTL formulae, called *primitives* (Section 4.3). The optimality of each candidate formula for a node is assessed using an appropriately defined *impurity measure*, which captures how well it splits the

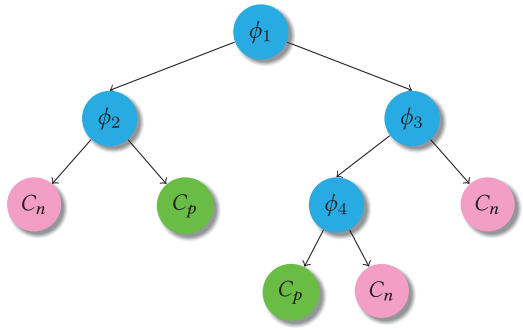

Fig. 1. The formula associated with the tree is $\phi_{tree} = (\phi_1 \wedge \neg\phi_2) \vee (\neg\phi_1 \wedge (\phi_3 \wedge \phi_4))$ and can be obtained using Algorithm 2.

signals reaching that node (Section 4.4). Since we are not just proposing a single algorithm but a class of algorithms, we refer to this approach as "decision tree learning framework for temporal logic inference."

The induction procedure is presented in detail in Section 4.5 and a resulting tree can be mapped to the equivalent STL formula using the simple algorithm described in Section 4.6. Figure 1 shows a tree and its corresponding STL formula. In Section 4.8, we discuss the link between depth of the tree and formula complexity. We define some termination conditions for the tree induction algorithm along with a post-completion pruning strategy. We conclude in Section 4.9 with an analysis of the computational complexity of the main algorithm.

4.3 PSTL Primitives

In the decision tree literature, a finite list of simple splitting rules is considered at each node [36]. The aim is to progressively explain the data with a combination of explainable functions. For our problem, we propose to use simple PSTL formulae, called *primitives*, to split the data. In particular, we define two types of primitives:

Definition 4.1 (First-Level Primitives). Let $\mathcal{S}$ be the set of signals with values in $\mathbb{R}^n$. We define the set of first-level primitives as follows:

$$\mathcal{P}^1 = \Big\{ \mathbf{F}_{[\tau_1,\tau_2]}(f_i(x) \sim \mu) \text{ or } \mathbf{G}_{[\tau_1,\tau_2]}(f_i(x) \sim \mu) \\ \mid i \in \{1,\dots,n\},\ \sim \in \{\leq, >\} \Big\}.$$

The parameters for the PSTL formulae in $\mathcal{P}^1$ are (μ, τ_1, τ_2) and the respective space of parameters is $\Theta^1 = \{(\mu, \tau_1, \tau_2) \mid \mu \in \mathbb{R},\ \tau_1 < \tau_2,\ \tau_1, \tau_2 \in \mathbb{R}_{\geq 0}\}$.

Definition 4.2 (Second-Level Primitives). Let $\mathcal{S}$ be the set of signals with values in $\mathbb{R}^n$. We define the set of second-level primitives as follows:

$$\mathcal{P}^2 = \Big\{ \mathbf{G}_{[\tau_1,\tau_2]}\mathbf{F}_{[0,\tau_3]}(f_i(x) \sim \mu) \text{ or } \mathbf{F}_{[\tau_1,\tau_2]}\mathbf{G}_{[0,\tau_3]}(f_i(x) \sim \mu) \\ \mid i \in \{1,\dots,n\},\ \sim \in \{\leq, >\} \Big\}.$$

The parameters for the PSTL formulae in $\mathcal{P}^2$ are $(\mu, \tau_1, \tau_2, \tau_3)$ and the respective space of parameters is $\Theta^2 = \{(\mu, \tau_1, \tau_2, \tau_3) \mid \mu \in \mathbb{R},\ \tau_1 < \tau_2,\ \tau_1, \tau_2, \tau_3 \in \mathbb{R}_{\geq 0}\}$.

The meaning of first-level primitives is straightforward. The primitive $\mathbf{F}_{[\tau_1,\tau_2]}(f_i(x) \sim \mu)$ is used to express that the predicate $f_i(x) \sim \mu$ must be true for at least one time instance in the interval $[\tau_1, \tau_2]$, while the primitive $\mathbf{G}_{[\tau_1,\tau_2]}(f_i(x) \sim \mu)$ expresses that $f_i(x) \sim \mu$ must be true for all time in

the interval. Similarly, the formulae in $\mathcal{P}^2$ can be interpreted in natural language. For example, the primitive $\mathbf{F}_{[\tau_1,\tau_2]}\mathbf{G}_{[0,\tau_3]}(f_i(x) \sim \mu)$ specifies that "the predicate $(f_i(x) \sim \mu)$ must hold true for τ_3 seconds and its start time is in the interval $[\tau_1, \tau_2]$." Both first- and second-level primitives may be thought as specifications for bounded reachability and safety with varying degrees of flexibility.

Remark 4.1. We choose the primitives in $\mathcal{P}^1$ and $\mathcal{P}^2$ as building blocks for constructing longer formulae, because they are generic and easy to interpret. It is important to stress, however, that the proposed PSTL primitives are not the only possible ones. A user may define other primitives, either generic ones, like the first- and second- level primitives, or specific ones, guided by the particular nature of the learning problem at hand.

Remark 4.2. Let $\mathcal{P}$ be the set of PSTL primitives. The fragment of STL that is mapped with decision trees corresponds to the Boolean closure of the valuations from $\mathcal{P}$. We denote this fragment with $\text{STL}_{\mathcal{P}}$. In other words, each decision tree constructed with the set of primitives $\mathcal{P}$ is mapped to an STL formula belonging to the $\text{STL}_{\mathcal{P}}$ fragment.

4.4 Impurity Measures

In the previous section, we defined a list of ways to split the data using a set of primitives $\mathcal{P}$. It is also necessary to define a criterion to select which primitive best splits the data at each node. Intuitively, a good split leads to children that are *pure*, that is, they contain mostly objects belonging to the same class. This concept has been formalized in literature with *impurity measures* [7, 35], and the goal is to obtain children *purer* than their parents.

Definition 4.3 (Impurity Measures). Let S be a finite set of signals and ϕ an STL formula. The following partition weights are introduced to describe how the signals s^i are distributed according to their labels l^i and the formula ϕ:

$$p_\top = \frac{|S_\top|}{|S|},\ p_\bot = \frac{|S_\bot|}{|S|},\ p_{C_p} = \frac{\left|S_{C_p}\right|}{|S|},\ p_{C_n} = \frac{|S_{C_n}|}{|S|}, \tag{1}$$

where $S_\top = \{(s^i, l^i) \in S \mid s^i \models \phi\}$, $S_\bot = \{(s^i, l^i) \in S \mid s^i \not\models \phi\}$, $S_{C_p} = \{(s^i, l^i) \in S \mid l^i = C_p\}$, and $S_{C_n} = \{(s^i, l^i) \in S \mid l^i = C_n\}$. In other words, $p_\top$ and $p_\bot$ represent the fraction of signals from S present in $S_\top$ and $S_\bot$, respectively, whereas p_{C_p} and p_{C_n} represent the fraction of signals in S belonging to class C_p and C_n, respectively.

The impurity measures are defined as [7, 35] follows:

— *Information gain (IG)*

$$\begin{aligned} IG(S, \phi) &= H(S) - \sum_{\otimes \in \{\top, \bot\}} p_\otimes \cdot H(S_\otimes), \\ H(S) &= -p_{C_p} \log p_{C_p} - p_{C_n} \log p_{C_n}; \end{aligned} \tag{2}$$

— *Misclassification gain (MG)*

$$\begin{aligned} MG(S, \phi) &= MR(S) - \sum_{\otimes \in \{\top, \bot\}} p_\otimes \cdot MR(S_\otimes), \\ MR(S) &= \min(p_{C_p}, p_{C_n}). \end{aligned} \tag{3}$$

Intuitively, a positive value for one of the impurity measure, such as the Information gain $IG(S, \phi)$, means that we have *reduced the impurity* by splitting the set S with the formula ϕ (or, equivalently, we have *gained purity*).

4.5 Parameterized Learning Algorithm

In Algorithm 1, we present the parameterized procedure for inferring temporal logic formulae from data. The *meta-parameters* of Algorithm 1 are (1) a set of PSTL primitives $\mathcal{P}$ and (2) an impurity measure J, which were defined in the previous sections. A set of stopping conditions *stop* is also necessary for determining the algorithm termination. We discuss them in Section 4.8.

ALGORITHM 1: Parameterized Decision Tree Construction—$buildTree(\cdot)$

Meta-Parameter: $\mathcal{P}$—set of PSTL primitives
Meta-Parameter: J—impurity measure
Parameter: *stop*—set of stopping criteria
Input: $S = \{(s^i, l^i)_{i=1}^N\}$—set of labeled signals
Input: h—the current depth level
Output: a (sub)-tree

1 **if** $stop(h, S)$ **then**
2 $\quad t \leftarrow leaf(\arg\max_{c \in C}\{p_c\})$
3 $\quad$ **return** t
4 $\phi^* \leftarrow \arg\max_{\psi \in \mathcal{P}, \theta \in \Theta} J(S, \psi(\theta))$
5 $t \leftarrow non\_terminal(\phi^*)$
6 $S^*_\top, S^*_\bot \leftarrow partition(S, \phi^*)$
7 $t.left \leftarrow buildTree(S^*_\top, h+1)$
8 $t.right \leftarrow buildTree(S^*_\bot, h+1)$
9 **return** t

Algorithm 1 is recursive and takes as input arguments the set of data S that reaches the current node, and the current depth level h. At the beginning, the stopping conditions are checked (line 1). If they are met, then the algorithm returns a single leaf node marked with the label $c \in C$. The label c is chosen according to the majority vote over data reaching that leaf (line 2). If the stopping conditions are not met (line 4), then the algorithm proceeds to find the optimal STL formula among all the valuations of PSTL formulae from the set of primitives $\mathcal{P}$. The cost function used in the optimization is the impurity measure J, which assesses the quality of the partition induced by PSTL primitives valuations. At line 5, a new non-terminal node is created and associated with the optimal STL formula ϕ^*. Next, the partition induced by the formula ϕ^* is computed (line 6). For each outcome of the split, the *buildTree*() procedure is called recursively to construct the left and right subtrees (lines 7 and 8). The corresponding data partition are passed. The depth level is increased by one.

The parameterized family of algorithms uses three procedures: (a) $leaf(c)$ creates a leaf node marked with the label $c \in C$, (b) $non\_terminal(\phi)$ creates a non-terminal node associated with the valuation of a PSTL primitive from $\mathcal{P}$, and (c) $partition(S, \phi)$ splits the set of signals S into satisfying and non-satisfying signals with respect to ϕ.

By fixing the meta-parameters ($\mathcal{P}$, J) and a set of stopping conditions (*stop*), a particular algorithm is *instantiated*. For each possible instance, a decision tree is obtained by executing $buildTree(S_{\text{ds}}, 0)$ on a batch set of labeled signals S_{ds}. Clearly, the returned tree depends on both the input data S_{ds} and the particular instance chosen.

4.6 Tree to STL Formula

A decision tree obtained by an instantiation of Algorithm 1 can be used directly for classification or converted to an equivalent STL formula using Algorithm 2. This algorithm recursively traverses the tree, starting from the root, and only keeps track of the paths reaching leaves associated with

ALGORITHM 2: Tree to formula—*Tree2STL*(·)

Input: t—node of a tree
Output: ϕ—STL Formula

1 **if** *t is a leaf and class associated with t is C_p* **then**
2 | **return** $\top$
3 **if** *t is a leaf and class associated with t is C_n* **then**
4 | **return** $\bot$
5 $\phi_l = (t.\phi \wedge \mathit{Tree2STL}(t.\mathit{left}))$
6 $\phi_r = (\neg t.\phi \wedge \mathit{Tree2STL}(t.\mathit{right}))$
7 **return** $\phi_l \vee \phi_r$

the positive class C_p. At each node, the formula is obtained by (1) conjunction of the node's formula with its left subtree's formula, (2) conjunction of the negation of the node's formula with its right subtree's formula, (3) disjunction of (1) and (2). Figure 1 shows a simple tree and its corresponding formula obtained by applying Algorithm 2.

4.7 Local Node Optimization

The cost function used in the local node optimization (line 4 of Algorithm 1) is one of the impurity measures defined in the Section 4.4. The local node optimization strategy presents some advantages. The optimization is performed over the chosen set of PSTL primitives $\mathcal{P}$ and their valuations Θ, Therefore, the optimization problem is always decomposed into $|\mathcal{P}|$ problems over a fixed and small number of real-valued parameters. In other terms, the complexity of the optimization problem does *not* depend on the length of the overall formula. The second advantage is due to the divide-and-conquer nature of Decision Trees. In Algorithm 1, the signals are partitioned between the children of the currently processed node and, consequently, the optimization becomes easier as the depth of the tree increases (fewer data to process).

The local optimization problems may be solved using any global non-linear optimization algorithm. In Reference [6], we used Simulated Annealing [21] and Differential Evolution [39]. However, we found that Particle Swarm Optimization [38], delivers superior performance. To use any these numerical optimization algorithms, we need to define finite bounds for the parameters of the primitive formulae. These bounds may easily be inferred from data, but may also be application-specific, if expert knowledge is available.

4.8 Stop Conditions, Pruning, and Formula Complexity

Due to the recursive nature of decision trees, Algorithm 1 can achieve perfect classification accuracy on the *training* data, if the maximum depth of the tree is unconstrained. This trivially occurs when we keep splitting the data until the current node contains only one signal (assuming there are no two exact signals with different labels).

This strategy is undesirable for several reasons. First, by fitting the training data perfectly, we are likely to model the noise contained in the data. In this scenario, the resulting formula performance on unseen signals will be poor for a lack of generalization ability (over-fitting). Inducing a deep tree will also lead to a longer execution time. Moreover, since in our approach there is a direct connection between depth of the tree and length of the corresponding STL formula (Section 4.6), a deeper tree will result in less interpretable formula.

To deal with these problems, two approaches are possible: (1) introduce more restrictive stopping conditions and (2) prune (merge back) unnecessary parts of the tree after its construction.

Several stopping criteria can be set for Algorithm 1. For instance, stop if the vast majority of the signals belong to the same class, either positive or negative, e.g., stop if 99% of signals belong to

the same class. Another common strategy is to stop if the algorithm has reached a certain, fixed, depth. These conditions also provide a faster termination of the algorithm.

For post-completion pruning, a popular method is cost-complexity pruning [17]. In this approach, a progressive sequence of shallower trees (corresponding to simpler formulae) is derived from the initial tree. Later, an independent set of data is used to pick the best tree in this sequence.

Specifically, a cost $C(t, \alpha)$ is associated to a tree t,

$$C(t, \alpha) = R(t) + \alpha L(t),$$

where $R(t)$, called resubstitution error, denotes the fraction of signals in the training set that are misclassified by t, and $\alpha L(t)$ is a penalty term that depends on the number of leaves $L(t)$ in t and a weight $\alpha \geq 0$. Depending on the value of α, a deep tree that makes no errors may have a higher cost C than a shallow tree that makes some classification errors.

Proposition 4.1. *If t_1 is an initial decision tree (like the one produced by Algorithm 1), then when a value for α is fixed, there exists only* one *subtree t_α of t_1 that minimizes the cost $C(t, \alpha)$* and *also has smallest $\alpha L(t)$ [17].*

Even though α can assume any value greater or equal zero, there is only a finite number of subtrees of t_1. Exploiting Prop. 4.1, we can construct a sequence of subtrees $t_1, t_2, \ldots, t_m$, so that (1) t_i is the smallest optimal subtree for $\alpha \in [\alpha_i, \alpha_{i+1}]$ and (2) t_{i+1} is a subtree of t_i. In other words, we can construct the next tree in the sequence t_{i+1} simply by pruning the current t_i, up until we reach a t_m, a tree composed only by the root node of t_1. To obtain t_{i+1}, we explore every node n_k of t_i (in a top-down fashion) and compute the value of α_{ik} at which t_i pruned at n_k becomes better than t_i (has lower cost C). Then, we prune all nodes n_k for which α_{ik} is minimum.

Once the sequence of trees $\{t_i\}_1^m$ has been constructed, the easiest way to pick the final tree (corresponding to a value of *alpha*), is to pick the tree that has minimum classification error on a hold-out (validation) set of signals. Alternatively, the optimal value for α can be determined within a cross-validation procedure.

4.9 Computational Complexity

In this section, we provide a worst-case and average-case complexity analysis of Algorithm 1 in terms of the complexity of the local optimization procedure (Algorithm 1, line 4). This complexity analysis assumes that just the sufficient stopping conditions are set. Let $C(N)$ and $g(N)$ be the complexity of Algorithm 1 and of the local optimization algorithm, respectively, where N is the number of signals to be processed by the algorithms. Trivially, we have $g(N) = \Omega(N)$, where $\Omega(\cdot)$ is the asymptotic notation for lower bound [11], because the algorithm must at least check the labels of all signals. The worst-case complexity of Algorithm 1 is attained when at each node the optimal partition has size $(1, N-1)$. In this case, the complexity satisfies the recurrence $C(N) = C(N-1) + C(1) + g(N)$, which implies $C(N) = \Theta(N + \sum_{k=2}^{N} g(k))$, where $\Theta(\cdot)$ is the two-sided asymptotic notation for complexity bound [11]. However, the worst-case scenario is not likely to occur in large datasets. Therefore, we consider the average case where at least a fraction $\gamma \in (0, 1)$ of the signals are in one set of the partition. The recurrence relation becomes $C(N) = C(\gamma N) + C((1-\gamma)N) + g(N)$, which implies the following complexity bound:

$$C(N) = \Theta\left(1 \cdot \left(1 + \int_1^N \frac{g(u)}{u} \mathrm{d}u\right)\right), \quad \forall 0 < \gamma < 1,$$

obtained using the Akra-Bazzi method [11]. Finally, note that the hidden constants in the complexity bounds above depend on the cardinality of the set of primitives considered and the size of their parameterization.

ALGORITHM 3: Online Decision Tree Construction

Input: (s, l)—a new labeled signal
Data: T—a tree

1 **if** T *does not exist* **then**
2 $\quad T \leftarrow emptyLeaf()$
3 $L, c \leftarrow locateLeaf(s, l)$
4 **if** $l \neq c$ **then**
5 $\quad updateLeaf(L)$

5 ONLINE LEARNING

In the batch algorithm proposed in Section 4.5, a *greedy* recursive procedure is followed to construct the tree, with all the data available S_{ds}, starting from the root. The data is partitioned as new nodes are created using locally optimal decisions on the signals reaching each node. The decision on which primitive to pick and which parameters to use is made by optimizing an impurity measure J on a set of primitives $\mathcal{P}$ and its space of parameter Θ.

In this section, we tackle the *online learning problem*. Here, new signals may arrive over time and the inference system should be updated to accommodate it. A trivial solution to the online problem would be, every time a new instance arrives, to use the offline learner of Section 4.5 from scratch on the whole data accumulated so far. Clearly, this is highly inefficient as any formula discovered, and any associated data structure, would be thrown away. Therefore, the focus of this section is to devise a method that *builds* and *updates* an STL formula used for data classification in an efficient manner.

As discussed in Reference [40], updating a decision tree when new data arrives is not an easy task. Specifically, if the arrival of a new signal causes the change of the best primitive in a node, in the sense of the impurity measure, then that node and all its children should be pruned and reconstructed. A different perspective to deal with this problem has emerged from the study of *data streams* (i.e., a data source that generates ordered sequence of instances, usually at a high rate [12, 25]). Instead of creating a node immediately, based on the data currently available, the key idea is to *defer* its creation until we can be *reasonably* sure that the decision made will hold in the future [12]. In particular, if an infinite amount of data was available, then we would (theoretically) be able to pick the best formula to use at each node. With a finite amount of data, it is not possible to be sure about which formula is the overall best, however a decision on the primitive to pick can be made using probabilistic arguments [12, 25, 37].

We present the online algorithm for constructing the tree in Section 5.1, and we describe the details of the decision process behind the creation of a new non-terminal node in Section 5.2.

5.1 Online Learning Algorithm

In Algorithm 3, we report the online procedure for inferring temporal logic formulae using high-level pseudocode. The algorithm can be executed whenever new signals are available. Algorithm 3 operates on a data structure T representing the tree and takes as input a new labelled signal (s, l) to be processed. At the beginning, the algorithm checks if the tree exists (line 1). If it does not, then it creates a tree with a single leaf at the root (line 2). When a tree exists, the new labelled signal (s, l) is sorted through the tree to the leaf L where it belongs and the label $c \in C$ of this leaf is examined (line 3). The label c associated with a leaf corresponds simply to the majority of the labels of the signals falling in that leaf. If the new signal is misclassified (line 4), that is $l \neq c$, then the procedure $updateLeaf()$ is invoked on leaf L (line 5).

ALGORITHM 4: Update a Leaf—*updateLeaf*$(\cdot)$

Meta-Parameter: $\mathcal{P}$—set of PSTL primitives
Meta-Parameter: J—impurity measure
Parameter: δ—confidence threshold
Parameter: $N_{\max}$—maximum number of signals
Input: L—a leaf of tree T
Data: S—set of signals contained in leaf L
Data: $\tilde{\mathcal{P}}$—set of candidate primitives for leaf L

1 $\theta_i \leftarrow \arg\max_{\theta \in \Theta} J(S, \psi_i(\theta_i)),\ \forall \psi_i \in \tilde{\mathcal{P}}$
2 $\tilde{\mathcal{P}}, \phi_{\text{bst1}}, createNode \leftarrow evalLeafStatus(S, \tilde{\mathcal{P}}, \{\theta_i\}_1^{|\tilde{\mathcal{P}}|})$
3 **if** $createNode == True$ **then**
4 $\quad N \leftarrow non\_terminal(\phi_{\text{bst1}})$
5 $\quad N.left \leftarrow emptyLeaf()$
6 $\quad N.right \leftarrow emptyLeaf()$
7 $\quad S_\top, S_\bot \leftarrow partition(S,\ \phi_{\text{bst1}})$
8 $\quad storeInLeaf(N.left, S_\top)$
9 $\quad storeInLeaf(N.right, S_\bot)$

The procedure *updateLeaf*(), reported in Algorithm 4, operates on a single leaf of the tree and performs three major steps. First, it finds the optimal parameters for each primitive in the set $\tilde{\mathcal{P}} \subseteq \mathcal{P}$ according to the impurity measure J (line 1). Second, it evaluates the status of the leaf to decide if it should be kept as a leaf or if a new non-terminal node can be created in its place (line 2). This part is discussed in the next section. Third, if the conditions are met (line 3), the leaf is transformed into a non-terminal node and it is associated with the optimal formula ϕ_{bst1} (line 4). Two empty leaves are initially added as children of this new node (lines 5 and 6). Finally, the signals S are partitioned according to ϕ_{bst1} (line 7), and for each outcome of the split the corresponding partition is passed to the appropriate leaf (lines 8 and 9).

Algorithms 3 and 4 use several functions: (a) *emptyLeaf*() creates a leaf with no signals in it and initializes the set of primitives to analyze $\tilde{\mathcal{P}}$ to $\mathcal{P}$; (b) *locateLeaf*(s, l) locates and stores a signal s with label l in the leaf L where it belongs according to the decision tree; (c) *evalLeafStatus*() tests the status of the candidate primitives in the leaf under analysis and checks the conditions to create a new node; (d) *storeInLeaf*(L, S) stores the signals S in leaf L.

Remark 5.1. Algorithm 4 operates on a single leaf of the tree at the time and only the signals belonging to that specific leaf are required to be in memory. Therefore, this approach can potentially handle large datasets. Alternately, if all accumulated signals can be stored in memory, the addition of new signals can be parallelized over different leaves.

5.2 Primitive Evaluation and Node Creation

Hypothetically, if an infinite set of signals S_∞ was available at a leaf, we would be able to pick the *best* formula to split the signals, both in terms of primitive and its parameters, with respect to the impurity measure Equation (3). Assume that ϕ_{bst1} $(= \psi_{\text{bst1}}(\theta_{\text{bst1}}))$ is this formula, corresponding to the primitive ψ_{bst1} with optimal valuation θ_{bst1}. Assume also that ϕ_{bst2} is the best formula obtainable with any other primitive, say ψ_{bst2}, we would obviously have

$$J(S_\infty, \phi_{\text{bst1}}) - J(S_\infty, \phi_{\text{bst2}}) > 0. \tag{4}$$

That is, primitive ψ_{bst1} provides an overall higher impurity reduction (purity gain) than ψ_{bst2}. With a finite amount of data, it is not possible to be sure about which formula is the best. However, some probabilistic guarantees on the best overall primitive to pick can be obtained using the finite set of signals S collected so far in the leaf. Following the idea initially proposed in Reference [12], a bound is derived on the difference of purity gains (using just the signals S available), such that,

$$\text{if} \qquad J(S, \phi_{\text{bst1}}) - J(S, \phi_{\text{bst2}}) > \epsilon(S, \delta), \tag{5}$$

$$\text{then} \qquad \Pr(\Delta J(S_\infty, \phi_{\text{bst1}}, \phi_{\text{bst2}}) > 0) \geq 1 - \delta. \tag{6}$$

In other words, if, on the S signals available, the difference between the purity gain of the best formula ϕ_{bst1}, obtained with the best primitive ψ_{bst1}, and the purity gain of the formula ϕ_{bst2}, obtained with the second best primitive ψ_{bst2}, is greater than a certain ϵ, then, with probability greater than $1 - \delta$, ψ_{bst1} is indeed better then ψ_{bst2} (as if we had access to infinite signals S_∞). Moreover, if Equation (5) holds and since there are $|\mathcal{P}|$ primitives, then we have that ψ_{bst1} is the *best* primitive with probability $(1 - \delta)^{(|\mathcal{P}|-1)}$.

In literature [12, 25, 37], several approaches have been pursued to obtain a value for ϵ in Equation (5) in order to guarantee Equation (6). They vary on the impurity measure used and on the concentration inequality (Hoeffding, McDiarmid, etc.) or probabilistic approximation employed. In this article, we investigate only the Misclassification Gain ($J = MG$) and use the bound recently derived in Reference [37] using a Gaussian Approximation approach[3]:

$$\epsilon_{MG}(S, \delta) = z_{1-\delta} \frac{1}{\sqrt{2|S|}}, \tag{7}$$

where $z_{1-\delta}$ is $(1 - \delta)$-th quantile of the normal distribution. In general, ϵ depends on the confidence threshold δ and on the cardinality of S. ϵ grows as δ approaches 0 (i.e., we want more confidence) and becomes smaller as the number of signals acquired increases (i.e., we collected more evidence).

Remark 5.2. It may happen that two primitives are almost equally good in terms of impurity reduction. In this scenario, a large number of signals would be required to assess the best one. To avoid a long decision time, the split of a leaf can be forced when more than $N_{\max}$ signals have been collected (tie breaking). Even in this scenario, the probabilistic bound is useful to speed up the computation, because it can be employed to progressively eliminate all the non-promising primitives from further analysis, that is, the primitives that compared with the current best satisfy the condition in (5).

The arguments previously discussed are used in the implementation of the function *evalLeafStatus*() for Algorithm 4. This function takes as input arguments the set of signals S collected so far in the leaf, the current set of candidate PSTL primitives $\tilde{\mathcal{P}} \subseteq \mathcal{P}$, and the optimal parameters for each primitive in $\tilde{\mathcal{P}}$, that is $\{\theta_i\}_1^{|\tilde{\mathcal{P}}|}$. It returns the updated set of PSTL primitives to consider in the future $\tilde{\mathcal{P}}$, the best splitting formula ϕ_{bst1}, and a Boolean *createNode* that indicates whether the leaf should become a new non-terminal node. *evalLeafStatus*() performs three major actions. First, it finds the best primitive, that is, the one associated with the highest purity gain. Second, it removes all the non-promising primitives from set $\tilde{\mathcal{P}}$ by checking them against the best primitive using Equation (5). Third, it sets *createNode* = *True* if only one primitive is left in set $\tilde{\mathcal{P}}$, or if the number of signals in the leaf has exceeded the maximum $|S| > N_{\max}$.

The specific values of the probability confidence threshold δ in Equation (6) and of the maximum number of signals $N_{\max}$ are parameters of Algorithm 4 and can be decided by the user.

[3]In this approach, each gain $MG(S_\infty, \phi_i)$ is a fixed unknown and the respective $MG(S, \phi_i)$ is its empirical estimator using S signals. The behavior of $MG(S, \phi_i)$ is characterized as a binomial random variable function of $|S|$ i.i.d signals. Later, it is approximated with a Gaussian r.v.

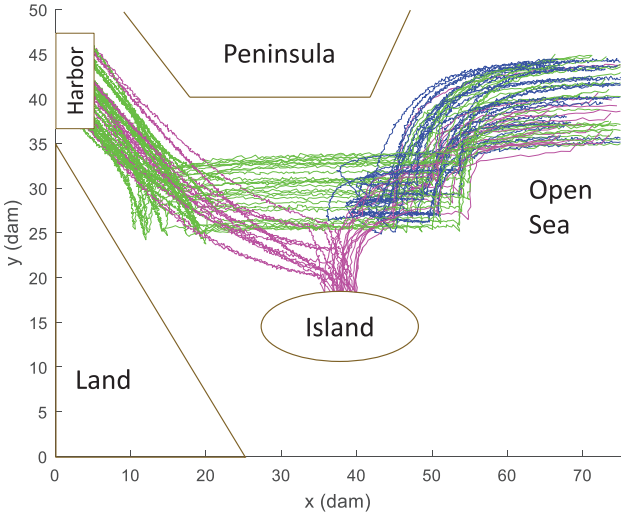

Fig. 2. Maritime surveillance dataset. The vessels behaving normally are shown in green. The magenta and blue trajectories represent two types of anomalous paths (human trafficking and terrorism, respectively).

Remark 5.3. Further considerations are possible during the leaf evaluation. For example, even if a good primitive has been found, a node creation can be deferred when the leaf contains mostly signals belonging to the same class.

6 CASE STUDIES

In this section, we present two case studies that illustrate the usefulness and the computational advantages of the algorithms. The first is an anomalous trajectory detection problem in a maritime environment. The second is a fault detection problem in an automotive powertrain system. The automotive application is particularly appealing because the systems involved are getting more and more sophisticated. In a modern vehicle, several highly complex dynamical systems are interconnected and the methods present in literature may fail to cope with this complexity.

6.1 Maritime Surveillance

This synthetic dataset emulates a maritime surveillance problem, where the goal is to detect suspicious vessels approaching the harbor from the open sea by looking at their trajectories. It was developed in Reference [28], based on the scenarios described in Reference [29], for evaluating their inference algorithms.

The trajectories are represented with planar coordinates $x(t)$ and $y(t)$ and were generated using a Dubins' vehicle model with additive Gaussian noise. Three types of scenarios, one normal and two anomalous, were considered. In the normal scenario, a vessel approaching from sea heads directly towards the harbor. In the first anomalous scenario, a ship veers to the island and heads to the harbor next. This scenario is compatible with human trafficking. In the second anomalous scenario, a boat tries to approach other vessels in the passage between the peninsula and the island and then veers back to the open sea. This scenario is compatible with terrorist activity. Some sample traces are shown in Figure 2. The dataset is composed of 2,000 total traces, with 61 sample points per trace. There are 1,000 normal traces and 1,000 anomalous.

6.2 Fuel Control System

We investigate a fuel control system for a gasoline engine. A model for this system is provided as a built-in example in Simulink, and we modified it for our purposes. This model was initially used for Bayesian statistical model checking [43] and has been recently proposed as a benchmark for the hybrid systems community [18]. We selected this model because it includes all the complexities of real-world industrial models, but is still quick to simulate, i.e., it is easy to obtain a large number of traces.

The key quantity in the model is the *air-to-fuel ratio*, that is, the ratio between the mass of air and the mass of fuel in the combustion process. The goal of the control system is to keep it close to the "ideal" stoichiometric value for the combustion process. For this system, the target air-fuel ratio is 14.6, as it provides a good compromise between power, fuel economy, and emissions. The system has one main output, the air-to-fuel ratio, one control variable, the fuel rate, and two inputs, the engine speed and the throttle command. The system estimates the correct fuel rate to achieve the target stoichiometric ratio by taking into account four sensor readings. Two are related directly to the inputs: the engine speed and the throttle angle. The remaining two sensors provide crucial feedback information: the EGO sensor reports the amount of residual oxygen present in the exhaust gas, and the MAP sensor reports the (intake) manifold absolute pressure. The EGO value is related to the air-to-fuel ratio, whereas the MAP value is related to the air mass rate. The Simulink diagram is made of several subsystems with different kinds of blocks, both continuous and discrete, among which there are look-up tables and a hybrid automaton. Due to these characteristics, this model can exhibit a rich and diverse number of output traces, thus making it an interesting candidate for our investigation.

The base model, that is, the one included in Simulink, includes a very basic fault detection scheme and fault injection mechanism. The fault detection scheme is a simple threshold crossing test (within a Stateflow chart) and is only able to detect single off range values. For avoiding the overlap of two anomaly detection schemes, the built-in one has been removed. In the base model, the faults are injected by simply reporting an incorrect and fixed value for a sensor's reading. Moreover, these faults are always present from the beginning of the simulation. We replaced this simple fault injection mechanism with a more sophisticated unit. The new subsystem is capable of inducing faults in both the EGO and MAP sensors with a *random* arrival time and with a *random* value. Specifically, the faults can manifest at any time during the execution (uniformly at random) and the readings of the sensors affected are offset by a value that *varies* at every execution. Finally, independent Gaussian noise signals, with zero mean and variance $\sigma^2 = 0.01$, have been added at the output of the sensors.

For the fuel control system, 1,200 total simulations were performed. In all cases, the throttle command provides a periodic triangular input, and the engine speed is kept constant at 300 rad/s (2,865 RPM). The simulation time is 60 s. In details, we obtained: 600 traces where the system was working normally; 200 traces with a fault in the EGO sensor; 200 traces with a fault in the MAP sensor; 200 traces with faults in both sensors. For every trace, we collected 200 samples of the EGO and MAP sensors' readings. The average simulation time to obtain a single trace was roughly 1 second.

7 RESULTS

We implemented and tested the algorithms described in Sections 4 and 5 using MATLAB. To assess the performance of the proposed methods, we used a fivefold cross-validation scheme. One round of cross-validation entails partitioning the whole dataset into two complementary subsets, performing the training on one subset, and the evaluation of the error on the other (testing). The

Table 1. Maritime Surveillance: Analysis of Classification Accuracy as a Function of Impurity Measure (J), Primitive Set ($\mathcal{P}$), and Maximum Tree Depth (D)

$(J, \mathcal{P})$	MCR ($D = 1$)	MCR ($D = 3$)	MCR ($D = 5$)
$(MG, \mathcal{P}^1)$	19.36% (0.46%)	1.31% (0.28%)	0.55% (0.33%)
$(MG, \mathcal{P}^2)$	19.24% (0.31%)	1.44% (0.18%)	0.45% (0.20%)
$(IG, \mathcal{P}^1)$	23.46% (1.42%)	0.81% (0.11%)	0.31% (0.46%)
$(IG, \mathcal{P}^2)$	21.17% (2.18%)	0.80% (0.18%)	0.07% (0.17%)

Every cell of the table contains the average train MCR and its standard deviation obtained with the cross-validation procedure.

Table 2. Fuel Control System: Analysis of Classification Accuracy as a Function of Impurity Measure (J), Primitive Set ($\mathcal{P}$), and Maximum Tree Depth (D)

$(J, \mathcal{P})$	MCR ($D = 1$)	MCR ($D = 3$)	MCR ($D = 5$)
$(MG, \mathcal{P}^1)$	23.31% (0.41%)	1.06% (0.32%)	0.35% (0.16%)
$(MG, \mathcal{P}^2)$	23.29% (0.59%)	1.02% (0.17%)	0.60% (0.23%)
$(IG, \mathcal{P}^1)$	23.50% (0.41%)	0.96% (0.39%)	0.31% (0.31%)
$(IG, \mathcal{P}^2)$	23.33% (0.56%)	1.02% (0.31%)	0.46% (0.19%)

Every cell of the table contains the average train MCR and its standard deviation obtained with the cross-validation procedure.

results for each round are averaged to obtain a single estimate of the misclassification rate, its standard deviation, and the average execution time. We ran our experiments on a Windows PC, with an Intel 5930K CPU and 16 GB ram.

7.1 Offline Learning Results

As discussed in Section 4.2, Algorithm 1 represents a *family* of algorithms and several possible algorithms can be created depending on the chosen impurity measure and primitive set (the *meta-parameters*). There is a large number of possible combinations, and we attempt to investigate at least some of them in Tables 1 and 2 for the maritime surveillance and fuel control case studies, respectively. As mentioned in Section 4.8, all meta-parameter combinations can achieve a good accuracy if the maximum depth of the tree is unconstrained. Therefore, it is more insightful to study how their accuracy behaves as a function of the maximum depth allowed (D) on the training data.

The tables show that information gain (IG) has a small edge over the misclassification gain (MG) as the depth of the tree increases. Especially for the naval surveillance case study, MG is better than IG at depth 1; however, at depths 3 and 5, IG gains the lead. This phenomenon is known in literature and can be intuitively explained with the capability of IG of preparing the data for a better division later, sometimes at the expense of the first splits [17].

In both case studies, the primitive sets $\mathcal{P}^1$ and $\mathcal{P}^2$ achieve very similar results. This indicates that the greater expressivity of $\mathcal{P}^2$ is not needed to describe these case studies. The execution time for $\mathcal{P}^2$ is higher than $\mathcal{P}^1$ (about 5 times). Even though both sets have the same number of elements, $\mathcal{P}^2$ involves a more complicated optimization problem. Specifically, primitives in $\mathcal{P}^2$ have 4 free parameters, whereas primitives in $\mathcal{P}^1$ have only 3 free parameters. Moreover, evaluating the primitives in $\mathcal{P}^2$ is computationally more demanding (due to the presence of nested temporal operators).

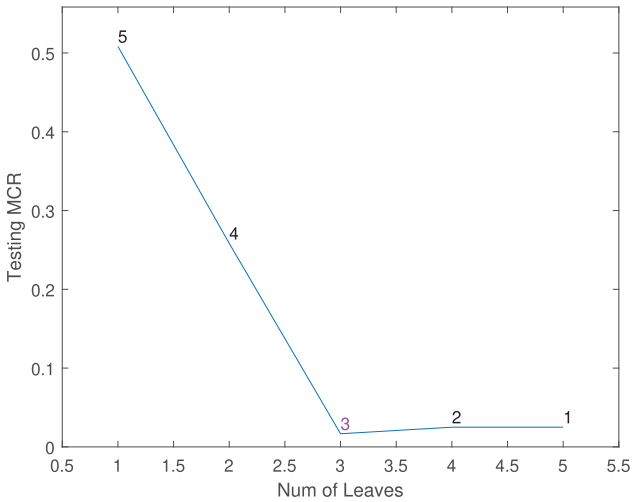

Fig. 3. Fuel Control: Test MCR as a function of the number of leaves in the tree sequence produced by the cost-complexity pruning algorithm.

The low variance preset in all results indicates a strong consistency in the produced formulae during every cross-validation round.

Following the discussion in Section 4.8, it is often advisable to induce a deeper tree from the training data and then use independent data to determine the right size.

For the automotive case study, if we use the impurity function IG with primitive set $\mathcal{P}^1$ and terminate the algorithm only when a maximum depth of 5 has been reached, we obtain the following formula (x_1 and x_2 stand for EGO and MAP, respectively):

$$\phi = \phi_1 \wedge (\phi_2 \wedge ((\phi_3 \wedge \phi_4) \vee \phi_5)), \tag{8}$$

$$\begin{aligned} &\phi_1 = \mathbf{G}_{[47.7,59.7]}(x_1 > 0.285) \quad \phi_2 = \mathbf{G}_{[8.45,59.5]}(x_1 < 0.937), \\ &\phi_3 = \mathbf{G}_{[46.8,59.7]}(x_2 > 0.392) \quad \phi_4 = \mathbf{F}_{[37,39.9]}(x_1 < 0.358), \\ &\phi_5 = \mathbf{F}_{[46.8,59.7]}(x_1 < 0.392), \end{aligned} \tag{9}$$

with an associated MCR of 0.42% on the training data and 2.50% on the testing data (execution time 30 s). Figure 3 shows the testing error for the sequence of trees constructed by the post-completion pruning procedure. By selecting the third tree in the sequence, we simplify it to

$$\phi = \mathbf{G}_{[47.7,59.7]}(x_1 > 0.285) \wedge \mathbf{G}_{[8.45,59.5]}(x_1 < 0.937),$$

with an associated MCR of 2.50% on the training data and 1.67% on the testing data. This formula establishes tight thresholds for the EGO sensor values. Therefore, we not only obtained a more interpretable formula but also one that is performing better on unseen data.

7.2 Online Learning Results

We used again a cross-validation scheme. In this case, however, the training signals are presented to Algorithm 3 one at the time. We used the MG as impurity measure and the $\mathcal{P}^1$ as primitive set.

For the maritime surveillance case study, we obtained a mean MCR of 1.95% (STD 0.54%). The mean runtime (to process *all* the signals in the training set) was about 140 s per cross-validation

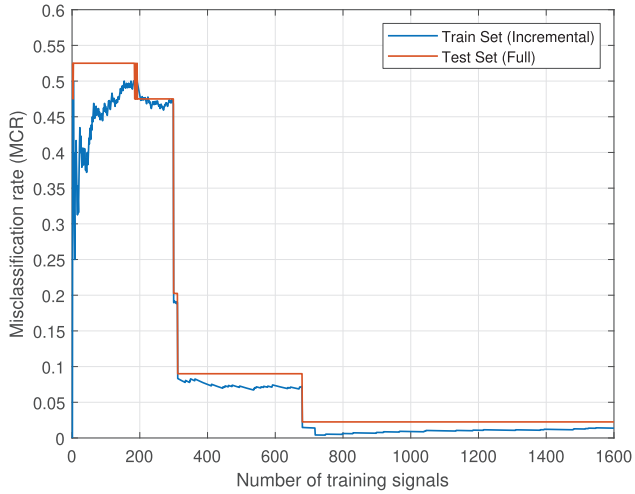

Fig. 4. Maritime Surveillance: MCR as function of the number of signal processed. In blue, evolution of MCR computed on training signals (as seen by the algorithm). In red, evolution of MCR computed on an independent test set.

round. A sample formula, obtained in one of the rounds after processing 1,600 signals, is

$$\phi = \left(\neg\phi_1 \wedge \phi_3\right) \vee \left(\phi_1 \wedge \neg\phi_2\right) \vee \left(\phi_1 \wedge \phi_2 \wedge \phi_4\right), \tag{10}$$
$$\phi_1 = \mathbf{F}_{[191,267]}(x > 20.1), \qquad \phi_2 = \mathbf{F}_{[99.9,294]}(y > 32.2),$$
$$\phi_3 = \mathbf{F}_{[58.9,186]}(x > 43.2), \qquad \phi_4 = \mathbf{G}_{[68.3,300]}(y > 30.1).$$

For the fuel control case study, we obtained a mean misclassification rate of 1.92% with a standard deviation of 0.96%. The mean runtime was 90 s per round. A sample formula obtained during one of the cross-validation round is reported in Equation (12).

It is interesting to analyze how Algorithm 3 performs as signals are incrementally processed. Figure 4 displays how two error rates evolve as training signals from the maritime dataset are presented to the algorithm. The first, in blue, is the misclassification rate on the set of *training* signals seen *so far* by the algorithm, while the second, in red, is the misclassification rate respect an independent (fixed) *test* set. As expected, the functions are flat for ample intervals with jumps at the points where the algorithm creates a new node in the tree. At these points, the corresponding formula becomes more complex (longer) and, generally, the misclassification rate decreases.

The evolution of the errors on the training and test data provides valuable information. For instance, if the training error decreases while the test error remains stable or increases, the formula inferred is getting overly specific to the training data and will not generalize well on unseen data (overfitting). From Figure 4, it also clear that after a certain number of signals has been processed, adding more signals, while still increasing the complexity of the formula, does not improve the classification accuracy significantly. This information can be exploited to stop early the learning process (that is, *before* the whole dataset is processed) and focus on a reasonably accurate and more interpretable formula. For example in the maritime surveillance case study, by looking at Figure 4, if we stop after 420 signals have been processed (after around 30 s), we obtain a simpler formula

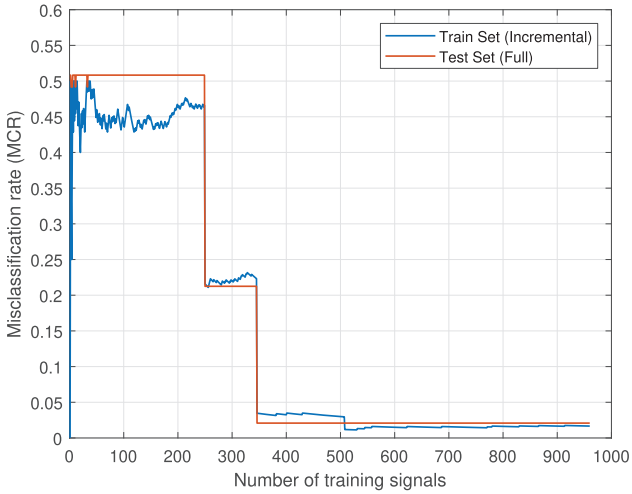

Fig. 5. Fuel Control System: MCR as function of the number of signal processed. In blue, evolution of MCR computed on training signals (as seen by the algorithm). In red, evolution of MCR computed on an independent test set.

(compare with Equation (10)):

$$\phi = \left(\neg\phi_1 \wedge \phi_3\right) \vee \left(\phi_1 \wedge \neg\phi_2\right), \tag{11}$$
$$\phi_1 = \mathbf{F}_{[191,267]}(x > 20.1), \qquad \phi_2 = \mathbf{F}_{[99.9,294]}(y > 32.2),$$
$$\phi_3 = \mathbf{F}_{[58.9,186]}(x > 43.2),$$

with an associated MCR of 2.25%. This is a form of *online evaluation* of a formula's performance and complexity as opposed to the post-completion pruning procedure described in Section 4.8. Notice also the insight we can gain from the English translation of Equation (11): Normal vessels' x coordinate is eventually past 43.2 in the middle part of the trajectory (i.e., the ships progress toward the port), and is globally less than 20.1 during the final stages (for the ships that dock in the port). Moreover, normal vessels' y coordinate stays below 23.7 for the ships that dock outside the port.

Likewise, for the automotive case study, by stopping the learning process after 350 signals, we get the formula (Figure 5)

$$\phi = \mathbf{G}_{[0,59.7]}(x_1 > 0.008) \wedge \mathbf{G}_{[7.41,59.1]}(x_1 \leq 0.931), \tag{12}$$

with an associated MCR of 2.00% on the testing data.

Remark 7.1. We did not test the online learning algorithm using the *IG*. Switching from *MG* to *IG* as impurity measure is not as straightforward as in the offline algorithm. The core inequality in Equation (5) depends on the impurity measure used. Therefore, as discussed at the end of Section 5.2, Equation (7) is tailored for the misclassification gain. In literature, some bounds have been derived for *IG* but they are not as tight as the one for *MG*. A comparison between *MG* to *IG* in the online algorithm would be skewed by this factor.

7.3 Discussion

Given the same meta-parameters, Algorithms 1 and 3 are able to produce similar formulae. However, comparing them is not straightforward due to the different problem setting they address. Specifically, Algorithm 1 is an offline algorithm, that is, it processes the whole dataset in a single batch and produce a formula, whereas the Algorithm 3 processes the dataset one signal at a

time and produces a new formula after each addition. In terms of classification performance, the accuracy of the online algorithm is on par with the results of the offline algorithm. However, to elaborate the *whole* maritime surveillance dataset, Algorithm 3 is around 2.5 times slower than Algorithm 1. In this regard, it is worth noting that the signal addition time of online algorithm is not constant. For the maritime dataset, it goes from less than 1 second to around 5 s, depending on whether the algorithm decides to (1) do nothing, (2) only reoptimize the primitives' parameters, or (3) create a new node. Moreover, as mentioned in previous section, it is often *not* necessary to process the whole dataset available.

As noted in Section 4.6, in this article, we employ a different optimization algorithm for the local node optimization. This has led to an average speedup of 25× with respect to our previous work [6]. For example, on the maritime surveillance dataset, the average cross-validation fold execution time, with hyper-parameters (MG, $\mathcal{P}^1$) and maximum depth 5, was 16 min in Reference [6] and 35 s now. Likewise, for the fuel control system, the average cross-validation fold execution time was 18 min in Reference [6] and 40 s now. The speedup is even greater when using $\mathcal{P}^2$, since in Reference [6], we attempted to use another optimization algorithm (different evolution) that performed poorly.

The maritime surveillance case study was also investigated in References [28, 34]. Reference [28] uses an SVM-inspired optimization coupled with heuristics for constructing the formula structure. [34] uses a Gaussian Process inspired optimization coupled with a genetic algorithm on primitives for the formula structure. Unfortunately, it is not easy to make a formal comparison between the formulae learned by our approach and the ones in Reference [28]. This is due to the fact that iPSTL, defined in Reference [28], and $\text{STL}_{\mathcal{P}}$ do not represent the same STL fragment. Likewise, Reference [34] uses a slightly different set of primitives (for example, they explicitly include the until operator). Nevertheless, it is always possible to make a judgment in terms of sheer classification performance and, to a less extent, execution time. With respect to Reference [28], we improve the misclassification rate by a factor of 20 while being around 25× faster. In Reference [34], the authors report a perfect accuracy and average execution time of 73 s (on their machine). Some heuristics within the training algorithm are present to promote simpler formulae.

As mentioned in related work (Section 2), most approaches involve two major decisions: how parameters are optimized and how the formula structure is induced (including the *granularity* at which it is induced with *primitives*). Assuming everything else being equal, genetic algorithms support more flexible formula structures, however, the price to pay is a more challenging parameter optimization problem, with more parameters to be optimized and always the same amount of data at every step. The performance advantage of our method is due to the "divide and conquer" nature of growing STL formulae represented by special binary trees. For the offline algorithm, the problem of finding optimal primitives becomes progressively easier as the tree is constructed. This follows from the fact that a node's optimization problem has always a fixed number of parameters and the data is partitioned between the two children of the node. For the online algorithm, a new node is created only when some conditions on the overall best primitive to pick are attained. This strategy avoids any pruning and can handle large datasets.

Another non-trivial advantage of our approach is that it is easily adaptable to the multi-class scenario, i.e., where there are multiple possible labels for a signal, not just *positive* and *negative*, and one formula for each class should be produced. In our case, it is sufficient to adapt the impurity measures in Definition 4.3 and execute Algorithm 1 once. Approaches based on heuristics [28] or genetic algorithms [34] can be employed in a multi-class scenario by running the learning algorithm multiple times using one-vs.-all partitions. This process, other than being more time consuming, does not guarantee mutual-exclusivity among the formulae produced.

8 CONCLUSION

We presented an inference framework of timed temporal logic properties from time-series data. The framework defines customizable decision-tree algorithms that output STL formulae as classifiers. In particular, two induction algorithms are proposed, one for offline learning (a batch of signals to be processed) and one for the online scenario (signals incrementally available). These algorithms may be customized by providing (a) a set of primitive properties of interest and (b) an impurity measure that captures the node's homogeneity. The resulting procedures are model-free and are suitable for inferring properties for problems such as anomaly detection and monitoring and in application domains as diverse as automotive and maritime security. We showed that the algorithms are able to capture relevant timed properties in both case studies with good classification accuracy. Finally, in this article, we addressed the link between formula complexity and its accuracy. For the offline supervised case, we employed a post-completion pruning procedure to avoid over-fitting and produce more interpretable formulae. In the online case, we argued that the evolution of the misclassification rate (as signals are processed by the algorithm) provides information that can be exploited to interrupt the data collection process if a satisfactory solution has already been found.

This work can be extended in several directions. We believe that the parameter optimization problem, at the core of our approach and several others, can be further studied and improved. An interesting idea is present in Reference [24], where the authors try to frame this problem so that smooth optimization techniques can be used (such as gradient descent). The optimization formulation can also play a role in making the produced formulae more resilient to noise in the training data. In this article, we used the formula length as a proxy for interpretability. This is a simple approach, but not the only possible one. Future work should investigate the construction of a better metric to assess the formula complexity and rank candidate formulae accordingly. Another research direction is *unsupervised learning*, where only a set of *unlabeled* signals is available and a formula (or set of formulae) should be inferred to describe them. Some results have been obtained with a grid-based approach [41], using hierarchical clustering [4], and by projecting the signals in the space of parameters of formula [42]. Finally, even though this line of research was motivated by the primary goal of producing formulae and not achieve the highest accuracy, it would be interesting to compare this body of work against the state of the art of signal classification.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for the valuable feedback provided.

REFERENCES

[1] Eugene Asarin, Alexandre Donzé, Oded Maler, and Dejan Nickovic. 2012. Parametric identification of temporal properties. In *Proceedings of the Conference on Runtime Verification (RV'11) (Lecture Notes in Computer Science)*, Sarfraz Khurshid and Koushik Sen (Eds.), Vol. 7186. Springer, Berlin, 147–160. DOI: https://doi.org/10.1007/978-3-642-29860-8_12

[2] Ezio Bartocci, Luca Bortolussi, Laura Nenzi, and Guido Sanguinetti. 2015. System design of stochastic models using robustness of temporal properties. *Theor. Comput. Sci.* 587 (July 2015), 3–25. DOI: https://doi.org/10.1016/j.tcs.2015.02.046

[3] Ezio Bartocci, Luca Bortolussi, and Guido Sanguinetti. 2014. Data-driven statistical learning of temporal logic properties. In *Formal Modeling and Analysis of Timed Systems*. Springer, 23–37.

[4] Giuseppe Bombara and Calin Belta. 2017. Signal clustering using temporal logics. In *Runtime Verification (Lecture Notes in Computer Science)*. Springer, Cham, 121–137. DOI: https://doi.org/10.1007/978-3-319-67531-2_8

[5] Giuseppe Bombara and Calin Belta. 2018. Online learning of temporal logic formulae for signal classification. In *Proceedings of the European Control Conference*.

[6] Giuseppe Bombara, Cristian-Ioan Vasile, Francisco Penedo, Hirotoshi Yasuoka, and Calin Belta. 2016. A decision tree approach to data classification using signal temporal logic. In *Proceedings of the 19th International Conference on*

Hybrid Systems: Computation and Control (HSCC'16). ACM, New York, NY, 1–10. DOI: https://doi.org/10.1145/2883817.2883843

[7] Leo Breiman, Jerome Friedman, Charles J. Stone, and Richard A. Olshen. 1984. *Classification and Regression Trees*. CRC Press.

[8] Sara Bufo, Ezio Bartocci, Guido Sanguinetti, Massimo Borelli, Umberto Lucangelo, and Luca Bortolussi. 2014. Temporal logic based monitoring of assisted ventilation in intensive care patients. In *Leveraging Applications of Formal Methods, Verification and Validation*. Number 8803 in Lecture Notes in Computer Science. Springer, 391–403. DOI: https://doi.org/10.1007/978-3-662-45231-8_30

[9] G. Chen, Z. Sabato, and Z. Kong. 2016. Active learning based requirement mining for cyber-physical systems. In *Proceedings of the IEEE 55th Conference on Decision and Control (CDC'16)*. 4586–4593. DOI: https://doi.org/10.1109/CDC.2016.7798967

[10] E. M. Clarke, Orna Grumberg, and Doron Peled. 1999. *Model Checking*. MIT Press.

[11] Thomas H. Cormen. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.

[12] Pedro Domingos and Geoff Hulten. 2000. Mining high-speed data streams. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'00)*. ACM, New York, NY, 71–80. DOI: https://doi.org/10.1145/347090.347107

[13] Alexandre Donzé, Thomas Ferrere, and Oded Maler. 2013. Efficient robust monitoring for STL. In *Computer Aided Verification*. Springer, 264–279.

[14] Alexandre Donzé and Oded Maler. 2010. Robust satisfaction of temporal logic over real-valued signals. In *Formal Modeling and Analysis of Timed Systems*, Krishnendu Chatterjee and Thomas A. Henzinger (Eds.). Number 6246 in Lecture Notes in Computer Science. Springer, Berlin, 92–106.

[15] Georgios E. Fainekos and George J. Pappas. 2009. Robustness of temporal logic specifications for continuous-time signals. *Theor. Comput. Sci.* 410, 42 (Sept. 2009), 4262–4291. DOI: https://doi.org/10.1016/j.tcs.2009.06.021

[16] Radu Grosu, Scott A. Smolka, Flavio Corradini, Anita Wasilewska, Emilia Entcheva, and Ezio Bartocci. 2009. Learning and detecting emergent behavior in networks of cardiac myocytes. *Commun. ACM* 52, 3 (2009), 97–105.

[17] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2016. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* (2nd ed.). Springer, New York, NY.

[18] Bardh Hoxha, Houssam Abbas, and Georgios Fainekos. 2014. Benchmarks for temporal logic requirements for automotive systems. *Proc. Appl. Verificat. Cont. Hybrid Syst.* (2014).

[19] Bardh Hoxha, Adel Dokhanchi, and Georgios Fainekos. 2018. Mining parametric temporal logic properties in model-based design for cyber-physical systems. *Int. J. Softw. Tools Technol. Transfer* 20, 1 (Feb. 2018), 79–93. DOI: https://doi.org/10.1007/s10009-017-0447-4

[20] Laurent Hyafil and Ronald L. Rivest. 1976. Constructing optimal binary decision trees is NP-complete. *Inform. Process. Lett.* 5, 1 (May 1976), 15–17. DOI: https://doi.org/10.1016/0020-0190(76)90095-8

[21] Lester Ingber. 1996. Adaptive simulated annealing (ASA): Lessons learned. *Control Cybernet.* 25 (1996), 33–54.

[22] Rolf Isermann. 2006. *Fault-Diagnosis Systems*. Springer.

[23] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. 2019. Deep learning for time series classification: A review. *Data Min. Knowl. Discov.* 33, 4 (July 2019), 917–963. DOI: https://doi.org/10.1007/s10618-019-00619-1

[24] Susmit Jha, Ashish Tiwari, Sanjit A. Seshia, Tuhin Sahai, and Natarajan Shankar. 2017. TeLEx: Passive STL learning using only positive examples. In *Runtime Verification (Lecture Notes in Computer Science)*. Springer, Cham, 208–224. DOI: https://doi.org/10.1007/978-3-319-67531-2_13

[25] Ruoming Jin and Gagan Agrawal. 2003. Efficient decision tree construction on streaming data. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'03)*. ACM, New York, NY, 571–576. DOI: https://doi.org/10.1145/956750.956821

[26] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy Deshmukh, and Sanjit A. Seshia. 2015. Mining requirements from closed-loop control models. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 99, 34 (2015), 1–1. DOI: https://doi.org/10.1109/TCAD.2015.2421907

[27] Michael J. Kearns and Umesh Virkumar Vazirani. 1994. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA.

[28] Z. Kong, A. Jones, and C. Belta. 2017. Temporal logics for learning and detection of anomalous behavior. *IEEE Trans. Automat. Control* 62, 3 (Mar. 2017), 1210–1222. DOI: https://doi.org/10.1109/TAC.2016.2585083

[29] K. Kowalska and L. Peel. 2012. Maritime anomaly detection using Gaussian process active learning. In *Proceedings of the 15th International Conference on Information Fusion (FUSION'12)*. 1164–1171.

[30] Changliu Liu, Tomer Arnon, Christopher Lazarus, Clark Barrett, and Mykel J. Kochenderfer. 2019. Algorithms for verifying deep neural networks. Retrieved from https://arXiv:1903.06758

[31] Oded Maler and Dejan Nickovic. 2004. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, Yassine Lakhnech and Sergio Yovine (Eds.). Number 3253 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 152–166.

[32] Apurva Narayan, Greta Cutulenco, Yogi Joshi, and Sebastian Fischmeister. 2018. Mining timed regular specifications from system traces. *ACM Trans. Embed. Comput. Syst.* 17, 2 (Jan. 2018), 46:1–46:21. DOI: https://doi.org/10.1145/3147660

[33] Daniel Neider and Ivan Gavran. 2018. Learning linear temporal properties. In *Proceedings of the Formal Methods in Computer Aided Design (FMCAD'18)*. 1–10. DOI: https://doi.org/10.23919/FMCAD.2018.8603016

[34] Laura Nenzi, Simone Silvetti, Ezio Bartocci, and Luca Bortolussi. 2018. A robust genetic algorithm for learning temporal specifications from data. In *Quantitative Evaluation of Systems (Lecture Notes in Computer Science)*, Annabelle McIver and Andras Horvath (Eds.). Springer International Publishing, 323–338.

[35] J. Ross Quinlan. 2014. *C4.5: Programs for Machine Learning*. Elsevier.

[36] Brian D. Ripley. 1996. *Pattern Recognition and Neural Networks*. Cambridge University Press.

[37] L. Rutkowski, M. Jaworski, L. Pietruczuk, and P. Duda. 2015. A new method for data stream mining based on the misclassification error. *IEEE Trans. Neural Netw. Learn. Syst.* 26, 5 (May 2015), 1048–1059. DOI: https://doi.org/10.1109/TNNLS.2014.2333557

[38] Y. Shi and R. Eberhart. 1998. A modified particle swarm optimizer. In *Proceedings of the IEEE International Conference on Evolutionary Computation Proceedings*. 69–73. DOI: https://doi.org/10.1109/ICEC.1998.699146

[39] Rainer Storn and Kenneth Price. 1997. Differential evolution—A simple and efficient heuristic for global optimization over continuous spaces. *J. Global Optimiz.* 11, 4 (Dec. 1997), 341–359. DOI: https://doi.org/10.1023/A:1008202821328

[40] Paul E. Utgoff, Neil C. Berkman, and Jeffery A. Clouse. 1997. Decision tree induction based on efficient tree restructuring. *Mach. Learn.* 29, 1 (Oct. 1997), 5–44. DOI: https://doi.org/10.1023/A:1007413323501

[41] P. Vaidyanathan, R. Ivison, G. Bombara, N. A. DeLateur, R. Weiss, D. Densmore, and C. Belta. 2017. Grid-based temporal logic inference. In *Proceedings of the IEEE 56th Annual Conference on Decision and Control (CDC'17)*. 5354–5359. DOI: https://doi.org/10.1109/CDC.2017.8264452

[42] Marcell Vazquez-Chanlatte, Jyotirmoy V. Deshmukh, Xiaoqing Jin, and Sanjit A. Seshia. 2017. Logical clustering and learning for time-series data. In *Computer Aided Verification (Lecture Notes in Computer Science)*. Springer, Cham, 305–325. DOI: https://doi.org/10.1007/978-3-319-63387-9_15

[43] Paolo Zuliani, André Platzer, and Edmund M. Clarke. 2013. Bayesian statistical model checking with application to Stateflow/Simulink verification. *Formal Methods Syst. Design* 43, 2 (Aug. 2013), 338–367. DOI: https://doi.org/10.1007/s10703-013-0195-3

Received May 2020; revised October 2020; accepted November 2020