Innovations in Systems and Software Engineering (2022) 18:427–442
https://doi.org/10.1007/s11334-022-00444-8

S.I. : ATVA 2021

MaxSAT-based temporal logic inference from noisy data

Jean-Raphaël Gaglione[1] · Daniel Neider[2] · Rajarshi Roy[2] · Ufuk Topcu[1] · Zhe Xu[3]

Received: 24 October 2021 / Accepted: 19 February 2022 / Published online: 6 April 2022

Abstract
We address the problem of inferring descriptions of system behavior using temporal logic from a finite set of positive and negative examples. In this paper, we consider two formalisms of temporal logic that describe linear time properties: Linear Temporal Logic over finite horizon (LTL_f) and Signal Temporal Logic (STL). For inferring formulas in either of the formalism, most of the existing approaches rely on predefined templates that guide the structure of the inferred formula. On the other hand, the approaches that can infer arbitrary formulas are not robust to noise in the data. To alleviate such limitations, we devise two algorithms for inferring concise formulas even in the presence of noise. Our first approach to infer minimal formulas involves reducing the inference problem to a problem in maximum satisfiability and then using off-the-shelf solvers to find a solution. To the best of our knowledge, we are the first to incorporate the usage of MaxSAT/MaxSMT solvers for inferring formulas in LTL_f and STL. Our second approach relies on the first approach to derive a decision tree over temporal formulas exploiting standard decision tree learning algorithm. We have implemented our approaches and verified their efficacy in learning concise descriptions in the presence of noise.

Keywords Linear Temporal Logic · Signal Temporal Logic · Decision tree · Specification mining · Explainable AI

1 Introduction

Explaining the behavior of complex systems in a form that is interpretable to humans has become a central problem in Artificial Intelligence. Applications where having concise system descriptions are essential include debugging [13,29,30,38], reverse engineering [37], motion planning [12,44], specification mining for formal verification [21,35], to name just a few examples.

For inferring descriptions of a system, we rely on a set of positive examples and a set of negative examples generated from the underlying system. Given such data, the objective is to infer a concise model in a suitable formalism that is consistent with the data; that is, the model must satisfy the positive examples and not satisfy the negative ones.

Most of the data representing AI systems consist of sequences since, more often than not, the properties of these systems evolve over time. For representing data consisting of sequences, temporal logic has emerged as a successful and popular formalism. Such logic, in addition to having resemblance to natural language, eliminates the ambiguities existing in natural language through mathematical rigor.

Linear Temporal Logic (LTL), developed by Pnueli [28], is one such temporal logic that describes properties of systems over discrete time intervals. To this end, LTL relies on temporal operators such as **F** ("finally"), **G** ("globally"), **U** ("until"), and several others to capture temporal properties of systems. In recent years, especially in AI-related applications (e.g., robot motion planning [8], inverse reinforcement learning [9]), Linear Temporal Logic over finite horizon [14] (LTL_f in short) has gained popularity. In this logic, one can describe the properties such as "the robot should reach the goal and not touch a wall or step into the water in the process" using the LTL_f formula $(\neg\texttt{water} \wedge \neg\texttt{wall})\ \mathbf{U}\ \texttt{goal}$.

✉ Jean-Raphaël Gaglione
jr.gaglione@utexas.edu

Daniel Neider
neider@mpi-sws.org

Rajarshi Roy
rajarshi@mpi-sws.org

Ufuk Topcu
utopcu@utexas.edu

Zhe Xu
xzhe1@asu.edu

1 University of Texas at Austin, Austin, TX, USA

2 Max Planck Institute for Software Systems, Kaiserslautern, Germany

3 Arizona State University, Tempe, AZ, USA

For describing continuous-time properties especially for cyber-physical systems [1,32], Signal Temporal Logic [22] is often used. STL, which is essentially an extension of LTL, reasons about signals which are real-valued finite or infinite time series. STL, thus, relies on temporal operators involving intervals of time to describe continuous-time properties. For instance, the property "for the first 60 seconds, the speed of the vehicle should be less than 30 km/h, and the steering angle should be less than 60°" for an autonomous vehicle, can be described using the STL formula $\mathbf{G}_{[0,60]}(\texttt{speed} < 30 \wedge \texttt{angle} < 60)$.

The task of inferring temporal logic formulas consistent with a given data has been studied extensively for both LTL and STL [5,20,39,40]. Most of the existing inference approaches, however, typically impose syntactic restrictions on the inferred formula using handcrafted templates. Such methods have several drawbacks. First, handcrafting templates may not be a straightforward task since it requires adequate knowledge about the underlying system. Second, by restricting the structure of inferred formulas, we potentially increase the size of the inferred formula. This makes the formulas difficult to comprehend by humans and also amplifies the computation efforts required to find a formula.

Nevertheless, there are approaches [10,27] that avoid the use of templates. These approaches reduce the learning problem to satisfiability problems in propositional logic and use highly optimized constraint solvers to systematically search for solutions. This results in effective algorithms that infer formulas that perfectly classify the input data. However, such exact algorithms suffer from the limitation that they are susceptible to failure in the presence of noise which is ubiquitous in real-world data. Furthermore, trying to infer formulas that perfectly classify a noisy sample often results in complex formulas, hampering interpretability.

To alleviate the limitation of the earlier approaches, in this paper we present two novel algorithmic frameworks[1] for inferring temporal logic formulas from a sample having system traces labeled as positive and negative. We exploit these frameworks to devise algorithms for inferring formulas in both LTL_f and STL. While our presented algorithms infer temporal logic formulas over finite horizon, they can be seamlessly extended to also infer formulas over infinite horizon with minor modifications.

The general goal of algorithms is to infer concise (and thus, interpretable) formulas that achieve a low *loss* on the sample, where loss $l(S, \varphi)$ refers to the fraction of examples in the sample S that the inferred formula φ misclassified. Precisely, the problem solved by the algorithms is the following: given a sample S and a threshold κ, find a minimal formula φ that has $l(S, \varphi) \leq \kappa$.

[1] Based on the conference version of this paper [18].

Our algorithmic frameworks derive ideas from the SAT-based learning algorithms introduced by Neider and Gavran [27]. Our first framework reduces the problem of formula inference to problems in maximum satisfiability. Roughly speaking, in this framework, we first encode the inference problem using formulas having appropriate weights assigned to various clauses. Then, we search for such assignments to the formulas that maximize the total weight of the satisfied clauses. Finally, using an assignment that maximizes the weights of the satisfied clauses, we construct an appropriate formula minimizing loss in a straightforward manner.

The first framework constructs a series of monolithic formulas to encode the inference problem and is, thus, often inefficient for inferring larger formulas. Our second framework solves the inference problem by dividing the problem into smaller subproblems based on a decision tree learning algorithm. Instead of finding formulas that achieve a loss of less than κ in one step, we exploit algorithms from the first framework to infer small formulas in LTL_f or STL for each decision node in the tree.

We have implemented a prototype of our algorithms in a publicly available tool. We have also verified the efficacy of our tool on synthetic as well as real-world data. From our observations, we conclude that our algorithms are effective in inferring concise LTL_f and STL formulas, particularly from the samples that contain noise.

Outline In Sect. 2, we introduce the necessary background. In Sect. 3, we formally introduce the LTL_f inference problem and, then, discuss a MaxSAT algorithm based on the first algorithmic framework to solve it. In Sect. 4, we formally introduce the STL inference problem and discuss a MaxSMT algorithm, also based on the first framework. In Sect. 5, we discuss our second algorithmic framework which is based on decision tree learning. In Sect. 6, we discuss the implementation of our algorithms and their performance on synthetic and real-world examples. In Sect. 7, we discuss the related works. Finally, in Sect. 8, we conclude and provide the possible future works.

2 Preliminaries

Propositional logic Let *Var* be a set of propositional variables, which take Boolean values {0, 1} (0 represents *false*, 1 represents *true*). Formulas in propositional logic—denoted by capital Greek letters—are defined inductively as follows:

$$\Phi := x \in \mathit{Var} \mid \neg\Phi \mid \Phi \vee \Phi$$

As syntax sugar, we allow the formulas *true*, *false*, $\Phi \wedge \Psi$, $\Phi \rightarrow \Psi$ and $\Phi \leftrightarrow \Psi$ which are defined in the standard manner.

An assignment is a mapping $v\colon Var \mapsto \{0,1\}$, which maps propositional variables to Boolean values. Now, we define the semantics of propositional logic using a valuation function $V(\Phi, v)$ that is inductively defined as follows: $V(x,v) = v(x)$, $V(\neg\Psi, v) = 1 - V(\Psi, v)$, and $V(\Psi \vee \Phi, v) = \max\{V(\Psi, v), V(\Phi, v)\}$. We say that v satisfies Φ if $V(\Phi, v) = 1$. A propositional formula Φ is satisfiable if there exists an assignment v that satisfies Φ.

First-order logic In this paper, we only consider a specific fragment of first-order logic—quantifier-free Linear Real Arithmetic (LRA)—and thus, we only define this fragment here.

First, let $\mathcal{X} = \{x_0, x_1, \ldots\}$ be a set of *variables*, which range over values in $\mathbb{R}$. Then, we define *terms* as follows: a term is either a constant $c \in \mathbb{R}$, a variable $x \in \mathcal{X}$, or a function application $t_1 \circ t_2$, where $\circ \in \{+, \cdot\}$ and t_1, t_2 are two terms. For instance, 5, x, and $3 \cdot x + 2 \cdot y$ are terms. To reflect the usual notation, we often drop the multiplication sign.

An *atomic formula* is a predicate symbol applied to terms. In LRA, we allow the usual binary predicates $<, \leq, =, \geq$, and $>$. For example, $3x+2y > 5$ is an atomic formula. Moreover, a *formula* is inductively defined as follows: a formula is either an atomic formula, the negation $\neg\varphi$ of a formula φ, or the disjunction $\varphi_1 \vee \varphi_2$ of two formulas φ_1, φ_2. We also add syntactic sugar and allow the formulas $\varphi_1 \wedge \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, and $\varphi_1 \leftrightarrow \varphi_2$, which are defined as usual.

To assign meaning to formulas, similar to propositional logic, we have assignments. An assignment, in this case, is a mapping $v\colon \mathcal{X} \rightarrow \mathbb{R}$, which assigns to each variable a real value. Assignments can easily be lifted to terms in the usual way, and we write $v(t)$ for the value of the term t under v. Finally, we can define when an assignment v *satisfies* a formula φ, which we denote by $v \models \varphi$: we have $v \models t_1 \diamond t_2$ for $\diamond \in \{<, \leq, =, \geq, >\}$ if and only if $v(t_1) \diamond v(t_2)$ is true, $v \models \neg\varphi$ if $v \not\models \varphi$, and $v \models \varphi_1 \vee \varphi_2$ if and only if $v \models \varphi_1$ or $v \models \varphi_2$. We say that a formula φ is *satisfiable* if an assignment v with $v \models \varphi$ exists.

3 Learning minimal $\mathrm{LTL_f}$ formulas

In this section, we first formally introduce the various ingredients of the $\mathrm{LTL_f}$ learning problem. Then we state the problem and finally describe our solution using the first algorithmic framework.

Finite traces Formally, a *trace* over a set $\mathcal{P}$ of propositional variables (which represent interesting system properties) is a finite sequence of symbols $u = a_0 a_1 \ldots a_n$, where $a_i \in 2^{\mathcal{P}}$ for $i \in \{0, \ldots, n\}$. For instance, $\{p,q\}\{p\}\{q\}$ is a trace over the propositional variables $\mathcal{P} = \{p, q\}$. The empty trace, denoted by ϵ, is an empty sequence. The length of a trace is given by $|u|$ (note $|\epsilon| = 0$). Moreover, given a trace u and $i < |u| \in \mathbb{N}$, we use $u[i]$ to denote the symbol at position i (counting starts from 0). Finally, we denote the set of all traces by $(2^{\mathcal{P}})^*$.

Linear Temporal Logic Linear Temporal Logic (LTL) is a logic that enables reasoning about sequences of events by extending propositional Boolean logic with temporal modalities. Given a finite set $\mathcal{P}$ of propositional variables, formulas in LTL—denoted by small greek letters—are defined inductively by:

$$\varphi := p \in \mathcal{P} \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\,\varphi \mid \varphi\,\mathbf{U}\,\varphi$$

As syntactic sugar, we allow the use of additional constants and operators used in propositional logic. Additionally, we include temporal operators $\mathbf{F}$ ("finally") and $\mathbf{G}$ ("globally") by $\mathbf{F}\,\varphi := \mathit{true}\,\mathbf{U}\,\varphi$ and $\mathbf{G}\,\varphi := \neg\,\mathbf{F}\,\neg\varphi$. The set of all operators is defined as $\Lambda = \{\neg, \vee, \wedge, \rightarrow, \mathbf{X}, \mathbf{U}, \mathbf{F}, \mathbf{G}\} \cup \mathcal{P}$ (propositional variables are considered to be nullary operators). We define the size $|\varphi|$ of an LTL formula φ to be the number of its unique subformulas. For instance, the size of formula $\varphi = (p\,\mathbf{U}\,\mathbf{X}\,q) \vee \mathbf{X}\,q$ is 5, since the distinct subformulas of φ are p, q, $\mathbf{X}\,q$, $p\,\mathbf{U}\,\mathbf{X}\,q$ and $(p\,\mathbf{U}\,\mathbf{X}\,q) \vee \mathbf{X}\,q$.

We interpret LTL over finite traces[2] as is done in several applications related to AI [4]. We define the semantics of $\mathrm{LTL_f}$ based on the definition by Giacomo and Vardi [14]. For the semantics, we use a valuation function V, that maps a formula, a finite trace and a position in the trace to a Boolean value. Formally we define V as follows:

$$\begin{aligned}
V(p,u,i) &= 1 \text{ if and only if } p \in u[i] \\
V(\neg\varphi,u,i) &= 1 - V(\varphi,u,i) \\
V(\varphi \vee \psi,u,i) &= \max\{V(\varphi,u,i), V(\psi,u,i)\} \\
V(\mathbf{X}\,\varphi,u,i) &= \min\{i < |u| - 1, V(\varphi,u,i+1)\} \\
V(\varphi\,\mathbf{U}\,\psi,u,i) &= \max_{i \leq j < |u|} \{\min\{V(\psi,u,j), \\
&\qquad \min_{i \leq k < j} \{V(\varphi,u,k)\}\}\}
\end{aligned}$$

We say that a trace $u \in (2^{\mathcal{P}})^*$ satisfies a formula φ if $V(u,\varphi,0) = 1$. For the sake of brevity, we use $V(u,\varphi)$ to denote $V(u,\varphi,0)$.

3.1 The learning problem

Problem input The input for this problem is provided as a sample $S \subset (2^{\mathcal{P}})^* \times \{0,1\}$ consisting of labeled traces. Precisely, sample S is a set of pairs (u,b), where $u \in (2^{\mathcal{P}})^*$ is a trace and $b \in \{0,1\}$ is its classification label. The traces labeled 1 are called positive traces, while the ones labeled 0 are called negative traces. We assume that in a sample

[2] LTL, when interpreted over finite traces, is sometimes referred to as $\mathrm{LTL_f}$.

$(u, b_1) = (u, b_2)$ implies $b_1 = b_2$, indicating that no trace can be both positive and negative. Further, we denote the size of S, that is, the number of traces in a sample, by $|S|$.

We define a *loss* function which assigns a real value to a given sample S and an LTL_f formula φ. Intuitively, the function evaluates how "well" the LTL_f formula φ classifies a sample. While there are numerous ways of defining it (e.g., quadratic loss function, regret, etc.), we use the definition:

$$l(S, \varphi) = \sum_{(u,b)\in S} \frac{|V(\varphi, u) - b|}{|S|}, \tag{1}$$

which calculates the fraction of traces in S which the LTL_f formula φ misclassified.

Having defined the setting, we now formally describe the problem we solve:

Problem 1 *Given a sample $S \subset (2^{\mathcal{P}})^* \times \{0, 1\}$ and threshold $\kappa \in [0, 1]$, find an LTL_f formula φ such that $l(S, \varphi) \leq \kappa$.*

Intuitively, the margin on the achieved loss κ allows for a bounded fraction of the traces to be considered as noise. We refer the readers to Appendix 4 for additional theoretical observations.

Generally speaking, the above problem is trivial if no constraint is imposed on the size of the output formula, since one can always find a large LTL_f formula with zero loss on a given sample, as indicated by the following remark.

Remark 1 Given sample S, there exists an LTL_f formula φ such that $l(S, \varphi) = 0$.

One can construct such a formula by enumerating the differences in the positive and negative traces using a sequence of $\mathbf{X}$ and appropriate propositions (see Appendix 1 for the exact formula). Such a formula, however, is large in size (of the order of $|S|^2 \times max_{(u,b)\in S}|u|$), and it does not help toward the goal of inferring a concise description of the data.

In the next section, thus, we present an algorithm to infer minimal LTL_f formulas based on maximum satisfiability, which is our first algorithmic framework.

3.2 The learning algorithm

Our solution to Problem 1 relies on MaxSAT solvers which we introduce next.

MaxSAT MaxSAT—a variant of the Boolean satisfiability problem (commonly known as SAT)—is the problem of finding an assignment that maximizes the number of satisfied clauses in a given propositional formula provided in CNF. For solving our problem, we use a more general variant of MaxSAT, known as Partial Weighted MaxSAT. In this variant, a weight function $w\colon \mathcal{C} \mapsto \mathbb{R} \cup \{\infty\}$ assigns a weight to every clause in the set of clauses $\mathcal{C}$ of a propositional formula. The problem is to then find a valuation v that maximizes $\Sigma_{C_i\in\mathcal{C}} w(C_i) \cdot V(v, C_i)$.

While the MaxSAT problem and its variants can be solved using dedicated solvers, standard SMT solvers like Z3 [25] are also able to handle such problems. According to terminology derived from the theory behind such solvers, clauses C_i for which $w(C_i) = \infty$ are termed as *hard* constraints, while clauses C_i for which $w(C_i) < \infty$ are termed as *soft* constraints. Given a propositional formula with weights assigned to clauses, MaxSAT solvers try to find a valuation that satisfies all the hard constraints and maximizes the total weight of the soft constraints that can be satisfied.

Given that we are using MaxSAT solvers that possess the capability of handling Partial Weighted MaxSAT problems, we can solve a stronger version of Problem 1. In this stronger version, the loss based on which we search for LTL_f formulas takes the following form:

$$wl(S, \varphi, \Omega) = \sum_{(u,b)\in S} \Omega(u)|V(\varphi, u) - b|,$$

where Ω is a function that assigns a positive real-valued weight to each u in the sample in such a way that $\sum_{(u,b)\in S} \Omega(u) = 1$. Observe that by considering $\Omega(u) = 1/|S|$ for all traces in the sample, we have exactly $wl(S, \varphi, \Omega) = l(S, \varphi)$ which is used in Problem 1. In this section, we will solve the stronger version, since not only does it enable us to solve Problem 1 but also makes our algorithmic framework versatile enough to assist the decision trees learning algorithm, described in Sect. 5.

For solving this problem, we devise an algorithm based on ideas from the learning algorithm of Neider and Gavran for inferring LTL_f formulas that perfectly classify a sample. Following their algorithm, we translate the problem of inferring LTL_f formulas into problems in Partial Weighted MaxSAT and then use an optimized MaxSAT solver to find a solution. More precisely, we construct a propositional formula Φ_n^S and assign weights to its clauses in such a way that an assignment v of Φ_n^S that satisfies all the hard constraints, satisfies two properties:

1. Φ_n^S contains sufficient information to extract an LTL_f formula φ_v of size n, and
2. the sum of weights of the soft constraints satisfied by it is equal to $1 - wl(S, \varphi_v, \Omega)$.

To obtain a complete algorithm, we increase the value of n (starting from 1) until we find an assignment v of Φ_n^S that satisfies the hard constraints and ensures that the sum of weights of the soft constraints is greater than $1 - \kappa$. The termination of this algorithm is guaranteed by the existence of an LTL_f formula with zero loss on the sample (see Remark 1).

Algorithm 1: Learning algorithm based on maximum satisfiability

Input: A sample S, Ω function, Threshold κ
$n \leftarrow 0$
repeat
  $n \leftarrow n + 1$
  Construct formula $\Phi_n^S = \Phi_n^{str} \wedge \Phi_n^{stf}$
  Assign weights to soft constraints in Φ_n^S:
    $w(y_{n,0}^u) = \Omega(u)$ for$(u, 1) \in S$, and $w(\neg y_{n,0}^u) = \Omega(u)$ for $(u, 0) \in S$
  Find assignment v using MaxSAT solver
until *Sum of weights of soft constraints* $\geq 1 - \kappa$
return φ_v

On a technical level, the formula Φ_n^S in Algorithm 1 is the conjunction $\Phi_n^S = \Phi_n^{str} \wedge \Phi_n^{stf}$, where Φ_n^{str} encodes the *structure* of the prospective LTL_f formula (of size n) and Φ_n^{stf} tracks the satisfaction of the prospective LTL_f formula with traces in S. We now explain each of the conjuncts in greater detail.

Structural constraints For designing the formula Φ_n^{str}, we rely on a canonical syntactic representation of LTL_f formulas, which we refer to as *syntax DAGs*. A syntax DAG is essentially a syntax tree (i.e., the unique tree that arises from the inductive definition of an LTL_f formula) in which common subformulas are shared. As a result, the number of the unique subformulas of an LTL_f formula coincides with the number of nodes, which we term as the size of its syntax DAG.

In a syntax DAG, to uniquely identify the nodes, we assign identifiers $1, \ldots, n$ in such a way that the root node is always indicated by n and every node has an identifier larger than that of its children, if it has any. An example of a syntax DAG is shown in Fig. 1.

To encode the structure of a syntax DAG using propositional logic, we introduce the following propositional variables: $x_{i,\lambda}$ for $i \in \{1, \ldots, n\}$ and $\lambda \in \Lambda$, which encode that Node i is labeled by operator λ (includes propositional variables); and $l_{i,j}$ and $r_{i,j'}$, for $i \in \{2, \ldots, n\}$ and $j, j' \in \{1, \ldots, i-1\}$, which encode that the left and right child of Node i is Node j and Node j', respectively. For instance, we must have variables $x_{6,\wedge}$, $l_{6,4}$, and $r_{6,5}$ to be true in order to obtain a syntax DAG where Node 6 is labeled with $\wedge$, has the left child to be Node 4, and the right child to be Node 5 (similar to the syntax DAG in Fig. 1).

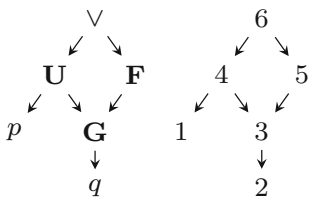


Fig. 1 Syntax DAG and identifiers of the formula $(p \,\mathbf{U}\, \mathbf{G}\, q) \vee \mathbf{F}\, \mathbf{G}\, q$

We now introduce constraints on the variables to ensure that they encode a valid syntax DAG. First, we ensure that each node of the syntax DAG has a unique label using the following constraint:

$$\Big[\bigwedge_{1\leq i\leq n}\bigvee_{\lambda\in\Lambda} x_{i,\lambda}\Big] \wedge \Big[\bigwedge_{1\leq i\leq n}\bigwedge_{\lambda\neq\lambda'\in\Lambda} \neg x_{i,\lambda} \vee \neg x_{i,\lambda'}\Big] \tag{2}$$

Next, we need constraints to ensure that each node of a syntax DAG has a unique left and right child, which can be done similar to Formula 2. Moreover, we must ensure that Node 1 is labeled by a propositional variable; we refer the readers to Appendix 1 for the remaining structural constraints. The overall formula Φ_n^{str} is obtained by taking the conjunction of all the structural constraints discussed above.

Observe that from a valuation v satisfying Φ_n^{str} one can extract a unique syntax DAG describing an LTL_f formula φ_v as follows: label Node p of the syntax DAG with the unique λ for which $v(x_{p,\lambda}) = 1$; assign Node n to be the root node; and assign edges from a node to its children based on the values of $l_{p,q}$ and $r_{p,q}$.

Semantic constraints Toward the definition of the formula Φ_n^{stf}, we define propositional formulas Φ_u^n for each trace u that tracks the valuation of the LTL_f formula encoded by Φ_n^{str} on u. These formulas are built using variables $y_{i,\tau}^u$, where $i \in \{1, \ldots, n\}$ and $\tau \in \{1, \ldots, |u| - 1\}$, that corresponds to the value of $V(\varphi_i, u, \tau)$ (φ_i is the LTL_f formula rooted at Node i). Now, to make sure that these variables have the desired meaning, we impose constraints based on the semantics of the LTL_f operators. For instance, for the **X**-operator, we impose the following constraint:

$$\bigwedge_{\substack{1<i\leq n\\ 1\leq j<i}} [x_{i,\mathbf{X}} \wedge l_{i,j}] \rightarrow \Big[\bigwedge_{0\leq\tau\leq|u|-1} \Big[y_{i,\tau}^u \leftrightarrow y_{j,\tau+1}^u\Big]\Big] \tag{3}$$

This constraint states that if Node i is labeled with **X** and its left child is Node j, then the satisfaction of the formula rooted at Node i at time τ (i.e., $y_{i,\tau}^u$) equals the satisfaction of the subformula rooted at Node j at time $\tau + 1$ (i.e., $y_{j,\tau+1}^u$). The constraints for the remaining operators can again be found in Appendix 1. The formula Φ_u^n is the conjunction of all such semantic constraints

We now define Φ_n^{stf} to be:

$$\Phi_n^{stf} = \bigwedge_{(u,b)\in S} \Phi_u^n \wedge \bigwedge_{(u,1)\in S} y_{n,0}^u \wedge \bigwedge_{(u,0)\in S} \neg y_{n,0}^u \tag{4}$$

Weight assignment For assigning weights to the clauses of Φ_n^S, we first convert the formulas Φ_n^{str} and Φ_n^{stf} into CNF. Toward this, we simply exploit the Tseitin transformation

[36] which converts a formula into an equivalent formula in CNF whose size is linear in the size of the original formula.

We now assign weights to constraints starting with the hard constraints as follows: $w(\Phi_n^{str}) = \infty, w(\Phi_u^n) = \infty$ for all $(u, b) \in S$. Here, $w(\Phi) = w$ is a shorthand to denote $w(C_i) = w$ for all clauses C_i in Φ. The constraint Φ_n^{str} is a hard one since it ensures that we obtain a valid syntax DAG of an LTL_f formula. Φ_u^n ensures that the prospective LTL_f formula is evaluated on the trace u according to the semantics of LTL_f and thus, also needs to be a hard constraint.

The soft constraints are the ones that enforce correct classification and we assign them weights as follows: $w(y_{n,0}^u) = \Omega(u)$ for all $(u, 1) \in S$ and $w(\neg y_{n,0}^u) = \Omega(u)$ for all $(u, 0) \in S$. Recall that Ω refers to the function assigning weights to the traces.

To prove the correctness of our learning algorithm, we first ensure that the formula Φ_n^S along with the weight assigned to its clauses serves our purpose.

Lemma 1 *Let S be a sample, Ω the weight function, $n \in \mathbb{N} \setminus \{0\}$ and Φ_n^S the formula with the associated weights as defined above. Then,*

1. *The hard constraints are satisfiable; and*
2. *If v is an assignment that satisfies the hard constraints and maximizes the sum of weight of the satisfied soft constraints, then φ_v is an LTL_f formula of size n, such that $wl(S, \varphi_v, \Omega) \leq wl(S, \varphi, \Omega)$ for all LTL_f formulas φ of size n.*

The termination and the correctness of Algorithm 1, which is established using the following theorem, is a consequence of Lemma 1.

Theorem 1 *Given a sample S and threshold $\kappa \in \mathbb{R}$, Algorithm 1 computes an LTL_f formula φ that has $wl(S, \varphi, \Omega) \leq \kappa$ and is the minimal in size among all LTL_f formulas that have $wl(S, \varphi, \Omega) \leq \kappa$.*

The proof of the results in this section can be found in Appendix 1.

4 Learning minimal STL formulas

In this section, we formally introduce the ingredients for the STL learning problem, followed by the problem. We then present the STL learning algorithm based on the first framework. In particular, we pinpoint the differences between this algorithm and the one in Sect. 3.

Signals A signal is a time series that indicate the evolution of system features over time. Unlike traces, however, features assume real values here. Formally, a signal $u : \mathbb{T} \to \mathbb{R}^m$ is a function defined over a time domain $\mathbb{T}$. In this paper, we assume a discrete and finite time domain $\mathbb{T} = \{0, \ldots, n\}$. Moreover, given a signal u and $i \in \mathbb{T}$, we use $u[i]$ to denote the value of u at time i, and $u_j[i]$ to denote the value of its j^{th} coordinate. Since we use discrete time, we can define the length of a signal by $|u| = |\mathbb{T}|$. Finally, we denote the set of all signals by $(\mathbb{R}^m)^*$.

Signal Temporal Logic Signal Temporal Logic (STL) is an extension of LTL_f defined over signals [3,23], which branches out LTL_f in two directions: it employs temporal operators defined over time intervals, and it is interpreted over signals [7]. Formulas in STL—denoted by small greek letters—are defined inductively by:

$$\varphi := \pi \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \, \mathbf{U}_I \, \varphi$$

Here, π is a predicate of the form $f_\pi(u) \geq \theta$ with $f_\pi : \mathbb{R}^m \to \mathbb{R}$ being a function over the signal value, and $\theta \in \mathbb{R}$ a threshold. I is a time interval of the form $I := [a, b)$, with $0 \leq a < b$ two integers. The extended set of all operators is defined as $\Lambda = \{\neg, \vee, \wedge, \to, \mathbf{U}_I, \mathbf{F}_I, \mathbf{G}_I\} \cup \{\pi, \ldots\}$, where $\mathbf{U}_I$ is parameterized with $[a, b)$, and each π is parameterized with θ.

We interpret STL over final signals. We redefine the valuation function V from Sect. 3 to define the semantics of STL formulas as follows:

$$V(\pi, u, i) = 1 \text{ if and only if } f_\pi(u[i]) \geq \theta$$

$$V(\varphi \, \mathbf{U}_{[a,b)} \, \psi, u, i) = \max_{i+a \leq j < \min(i+b, |u|)} \{\min\{ V(\psi, u, j), \min_{i+a \leq k < j} \{V(\varphi, u, k)\}\}\}$$

Here, the value of θ is an attribute of the evaluated STL formula and can differ for each subformula.

4.1 The learning problem

Problem input As the input of this problem, in addition to a sample $S \subset (\mathbb{R}^m)^* \times \{0, 1\}$ consisting of labeled signals, we have a finite set of predicates Π. The set of predicates consists of the atoms for the prospective STL formulas. While for each predicate in Π, users need to specify the function used, the threshold θ need not be specified.

Apart from the additional set of predicates, the problem setting remains identical to that of Problem 1. In particular, in a sample $(u, b_1) = (u, b_2)$ implies $b_1 = b_2$. Also, the loss function $l(S, \varphi)$ has the same definition as in Eq. 1.

We are now ready to define the STL learning problem.

Problem 2 *Given S, Π, find a minimal STL formula φ using predicates from Π such that $l(S, \varphi) \leq \kappa$.*

Unlike Problem 1, the existence of a solution to Problem 2 is not always guaranteed. This is because the existence of an

STL formula with zero loss depends on the input predicates. Thus, in order to guarantee the existence of a solution, we restrict the set of predicates to have a specific structure. In particular, we propose the following set of predicates: $\Pi = \{u_j \geq \theta | 1 \leq j \leq m\}$. Note that such a restriction is required only for the theoretical guarantees. Our algorithm in fact works for any arbitrary set of predicates, if there exists an appropriate STL formula using them.

The restriction discussed above provides us with the following guarantee:

Remark 2 Given sample S and predicates $\Pi = \{u_j \geq \theta | 1 \leq j \leq m\}$, there exist an STL formula φ using predicates from Π such that $l(S, \varphi) = 0$.

The construction of an STL formula with zero loss is similar to the one for LTL_f and can be found in Appendix 1.

4.2 The learning algorithm

Our solution to the learning problem relies on MaxSMT solvers which we introduce next.

MaxSMT Unlike SAT problems, SMT (Satisfiability Modulo Theories) deals with the satisfiability of first-order formulas over background theories. Similar to MaxSAT, MaxSMT is the problem of finding assignments that maximize the number of satisfiable clauses [34]. The formal problem definition remains the same as in the case of MaxSAT. For our algorithm, we will exploit the Partial Weighted MaxSMT for the theory of Linear Real Arithmetic (LRA). Standard SMT solvers like Z3 [25] can handle such problems.

The algorithm for learning STL formulas follows the same framework as that for learning LTL_f formulas.

However, the syntax and semantics of STL being different from LTL_f, we modify the construction of the propositional formula Φ_n^S. In particular, the structural constraint Φ_n^{str}, additionally, encodes the temporal bounds for $\mathbf{U}$ and the value of the thresholds θ for the predicates. The semantic constraints Φ_u^n change to ensure that proper semantics of STL is used.

Structural constraints To include the features of STL in the structure of the syntax DAG, we introduce the following additional variables: $a_i \in \mathbb{N}$ and $b_i \in \mathbb{N}$ for $i \in \{1, \ldots, n\}$, which encode that the temporal bounds of Node i are $[a_i, b_i)$ when the operator labeling Node i uses temporal bounds (i.e., is $\mathbf{U}_I$), and $\theta_i \in \mathbb{R}$ for $i \in \{1, \ldots, n\}$, which encode the value of the parameterized threshold of Node i when a predicate is labeling Node i. Φ_n^{str} is a conjunction of the constraints specified in Sect. 3, with the additional constraint $0 \leq a_i < b_i$ for $i \in \{1, \ldots, n\}$.

The formula Φ_n^{str} constrains the variables $x_{i,\lambda}, l_{i,j}, r_{i,j}, a_i, b_i$ and θ_i to encode a valid syntax DAG, such that a valuation v of these variables satisfying Φ_n^{str} describes an STL formula φ_v. A unique φ_v can be extracted from v as for STL, where we also assign interval $[a_p, b_p)$ and parameter θ_p to Node p when labeled with some λ that expect, respectively, an interval and a parameter.

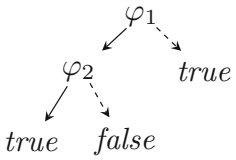


Fig. 2 A decision tree over LTL_f formulas

Semantic constraints We define Φ_u^n, which tracks the valuation of the STL formula encoded by Φ_n^{str} on u, as the conjunction of Formulas 5–8. Φ_n^{stf} is then defined as in Formula 4.

$$\bigwedge_{1 \leq i \leq n} \bigwedge_{\pi \in \Pi} x_{i,\pi} \rightarrow \left[\bigwedge_{0 \leq \tau < |u|} y_{i,\tau}^u \leftrightarrow f_\pi(u[\tau]) \geq \theta_i \right] \tag{5}$$

$$\bigwedge_{\substack{1 \leq i \leq n \\ 1 \leq j < i}} x_{i,\neg} \wedge l_{i,j} \rightarrow \left[\bigwedge_{0 \leq \tau < |u|} \left[y_{i,\tau}^u \leftrightarrow \neg y_{j,\tau}^u \right] \right] \tag{6}$$

$$\bigwedge_{\substack{1 \leq i \leq n \\ 1 \leq j, j' < i}} x_{i,\vee} \wedge l_{i,j} \wedge r_{i,j'} \rightarrow \left[\bigwedge_{0 \leq \tau < |u|} \left[y_{i,\tau}^u \leftrightarrow y_{j,\tau}^u \vee y_{j',\tau}^u \right] \right] \tag{7}$$

$$\bigwedge_{\substack{1 \leq i \leq n \\ 1 \leq j, j' < i}} x_{i,\mathbf{U}_I} \wedge l_{i,j} \wedge r_{i,j'} \rightarrow \left[\bigwedge_{0 \leq \tau < |u|} \left[y_{i,\tau}^u \leftrightarrow \bigvee_{\tau + a_i \leq \tau' < \min(\tau + b_i, |u|)} \left[y_{j',\tau'}^u \wedge \bigwedge_{\tau + a_i \leq t < \tau'} y_{j,t}^u \right] \right] \right] \tag{8}$$

The correctness of the algorithm adapted to learn STL formulas follows from the correctness of the formula Φ_n^S.

Theorem 2 *Given a sample S, predicates Π as indicated in Remark 2 and threshold $\kappa \in \mathbb{R}$, the MaxSMT-based STL learning algorithm terminates and outputs an STL formula φ that has $wl(S, \varphi, \Omega) \leq \kappa$ and is the minimal in size among all STL formulas that have predicates in Π and $wl(S, \varphi, \Omega) \leq \kappa$.*

5 Learning decision trees over temporal logic formulas

In this section, we present our second algorithmic framework for learning temporal logic formulas. While learning using such a framework does not guarantee minimal formulas, on the bright side, we obtain decision trees over temporal logic formulas that are considered to be human-interpretable structures. The framework can be adapted to devise learning algorithms for LTL_f and STL formulas in an identical manner. Thus, in this section, we only describe the algorithm for learning LTL_f formulas.

Decision trees over LTL_f formulas A decision tree over LTL_f formulas is a tree-like structure where all nodes of the tree are labeled by LTL_f formulas. While the leaf nodes of a decision tree are labeled by either *true* or *false*, the inner nodes are labeled by (non-trivial) LTL_f formulas which represent decisions to predict the class of a trace. Each inner

node leads to two subtrees connected by edges, where the left edge is represented with a solid edge and the right edge with a dashed one. Figure 2 depicts a decision tree over LTL_f formulas.

A decision tree t over LTL_f formula corresponds to an LTL_f formula $\varphi_t := \bigvee_{\rho \in \Pi} \bigwedge_{\varphi \in \rho} \varphi'$, where Π is the set of paths that originate in the root node and end in a leaf node labeled by *true* and $\varphi' = \varphi$ if it appears before a solid edge in $\rho \in \Pi$, otherwise $\varphi' = \neg\varphi$. For the decision tree in Fig. 2, the equivalent LTL_f formula is $(\varphi_1 \wedge \varphi_2) \vee \neg\varphi_1$.

For evaluating a decision tree t on a trace u, we use the valuation $V(\varphi_t, u)$ of the equivalent LTL_f formula φ on u. We can, in fact, extend the valuation function and loss function for LTL_f formulas to decision trees as $V(t, u) = V(\varphi_t, u)$ and $l(t, \varphi) = l(S, \varphi)$.

5.1 The learning algorithm

Our decision tree learning algorithm shares similarity with the class of decision tree learning algorithms known as Top-Down Induction of Decision Trees (TDIDT) [31]. Popular decision tree learning algorithms such as ID3, C4.5, CART are all part of the TDIDT algorithm family. In such algorithms, decision trees are constructed in a top-down fashion by finding suitable features (i.e., predicates over the attributes) of the data to partition it and then applying the same method inductively to the individual partitions.

Algorithm 2 outlines our approach to infer a decision tree over LTL_f formulas. In our algorithm, we first check stopping criterion (Line 1) that is responsible for the termination of the algorithm. If the stopping criterion is met, we return a leaf node. We discuss the exact stopping criterion used in our algorithm in Sect. 5.3.

If the stopping criterion fails, we search for an appropriate LTL_f formula φ using Algorithm 1 for the current node of the decision tree. Our search for φ is based on a score function, and we infer the minimal one that achieves a score greater than a user-defined minimum score μ on the sample. The choice of the score function and parameter μ is a crucial aspect of the algorithm, and we discuss more about this in Sect. 5.2.

Having inferred formula φ, next we split the sample into two sub-samples S_1 and S_2 with respect to φ as follows: $S_1 = \{(u, b) \mid V(\varphi, u) = 1\}$, and $S_2 = \{(u, b) \mid V(\varphi, u) = 0\}$. The final step is to recursively apply the decision tree learning on each of the resulting sub-samples (Line 6) to obtain trees t_1 and t_2. The decision tree returned is a tree with root node φ and subtrees t_1 and t_2.

Algorithm 2: Decision tree learning algorithm

Input: Sample S, Minimum score value μ, Threshold κ
Parameter: Stopping criterion *stop*, Score function s

1 **if** $stop(S, \kappa)$ **then**
2 **return** $leaf(S)$
3 **else**
4 Infer minimal formula φ with $s(S, \varphi) \geq \mu$ using Algorithm 1
5 Split S into S_1, S_2 using φ
6 Infer trees t_1, t_2 by recursively applying algorithm to S_1 and S_2
7 **return** *decision tree with root node* φ *and subtrees* t_1, t_2

5.2 LTL_f Formulas for decision nodes

Ideally, we aim to infer LTL_f formulas at each decision node that, in addition to being small, also ensure that the resulting sub-samples after a split are as "homogenous" as possible. In simpler words, we want the sub-samples obtained after a split to predominantly consist of traces of one particular class. More homogenous splits result in early termination of the algorithm resulting in small decision trees. To achieve this, one can simply infer a minimal LTL_f formula that perfectly classifies the sample. While in principle, this solves our problem, in practice inferring an LTL_f formula that perfectly classifies a sample is a computationally expensive process [27]. Moreover, it results in a trivial decision tree consisting of a single decision node. Thus, to avoid that, we wish to infer concise LTL_f formulas that classify most traces correctly on the given sample.

To mechanize the search for concise LTL_f formulas for the splits, we measure the quality of an LTL_f formula using a *score* function. In our algorithm, we use this function to infer a minimal LTL_f formula having a score greater than a user-defined threshold μ. The parameter μ regulates the trade-off between the height of the tree and the size of the LTL_f formulas in the decision nodes of a tree. While all TDIDT algorithms involve certain metrics (e.g., Gini impurity, entropy) to measure the efficacy of a feature to perform a split, these metrics are based on nonlinear operations on the fraction of examples of each class in a sample. Searching LTL_f formulas, however, based on such metrics cannot be handled using a MaxSAT framework.

One possible choice of score $s_l(S, \varphi) = 1 - l(S, \varphi)$, which relies on the loss function. A formula φ with $s_l(S, \varphi) \geq \mu$ is a formula with $l(S, \varphi) \leq 1 - \mu$. Thus, for inferring LTL_f formulas with score greater than μ, we invoke Algorithm 1 to produce a minimal LTL_f formula φ with $l(S, \varphi) \leq 1 - \mu$. Note that, for this score, one must choose the μ to be smaller than $1 - \kappa$, else one will end up with a trivial decision tree with a single decision node.

While s_l as the metric seems to be an obvious choice, it often results in a problem which we refer to as *empty splits*. Precisely, the problem of empty splits occurs when one of the sub-samples, i.e., either S_1 or S_2, becomes empty. Empty splits lead to an unbounded recursion branch of the learning algorithm, since using the best LTL_f formula (w.r.t. s_l) does

not produce any meaningful splits. This problem is more prominent in examples where the sample is skewed toward one class of examples. For instance, consider a sample $S = \{(u, 1)\} \cup \{(v_1, 0), (v_2, 0), \ldots (v_{99}, 0)\}$; for this sample, if one searches for an LTL_f formula with $\mu = 0.9$, *false* is a minimal formula; this formula, however, results in empty splits, since $S_1 = \emptyset$.

To address this problem, we use a score that relies on *wl* with a weight function Ω_r defined as follows:

$$\Omega_r(u) = \begin{cases} 0.5/|\{(u, b)|b = 1\}| \text{ for } (u, 1) \in S, \\ 0.5/|\{(u, b)|b = 0\}| \text{ for } (u, 0) \in S \end{cases}$$

Intuitively, the above Ω_r function normalizes the weight provided to traces, based on the number of examples in its class.

Our final choice of score, based on the above Ω_r function, is $s_r(S, \varphi) = max\{wl(S, \varphi, \Omega_r), 1 - wl(S, \varphi, \Omega_r)\}$. Using such a score, we also avoid having *asymmetric splits*. We say a split is asymmetric when the fraction of positive examples in S_1 is greater than or equal 0.5. Choosing the score to be $1-wl(S, \varphi, \Omega_r)$ always leads to asymmetric splits, since φ in order to minimize $wl(S, \varphi, \Omega_r)$ several positive traces need to end up in S_1. Now, for finding an LTL_f formula based on s_r, we need to invoke Algorithm 1 twice with $\kappa = 1 - \mu$; once with the original sample and once with the same sample but with class labels inverted and then, choosing the one that provides a formula with a better split.

While any score function that avoids the problem of empty and asymmetric splits is sufficient for our learning algorithm, we have used s_r as a score function in our experiments. We show that if we infer an LTL_f formula φ such that $s_r(S, \varphi) > 0.5$, we never encounter empty splits using the following lemma.

Lemma 2 *Given a sample S and an LTL_f formula φ, if $s_r(S, \varphi) > 0.5$, there exists traces u_1, u_2 in S such that $V(u_1, \varphi) = 1$ and $V(u_2, \varphi) = 0$.*

5.3 Stopping criterion

The stopping criterion is essential for the termination of the algorithm. Toward the definition of the stopping criterion, we define the following two quantities:

$$p_1(S) = |\{(u, b) \mid b = 1\}|/|S|$$
$$p_2(S) = |\{(u, b) \mid b = 0\}|/|S|$$

We now define the stopping criterion as follows: $stop(S) = true$ if $p_1(S) \leq \kappa$ or $p_2(S) \leq \kappa$, and *false* otherwise. Intuitively, the stopping criterion ensures that the algorithm terminates when the fraction of positive examples or fraction of negative examples in a resulting sample is less or equal to κ. When the stopping criterion holds, the algorithm halts and returns a leaf node labeled by $leaf(S)$ where *leaf* is defined as $leaf(S) = false$ if $p_1(S) \leq \kappa$ and *true* if $p_2(S) \leq \kappa$.

The following theorem ensures the correctness and termination of Algorithm 2.

Theorem 3 *Given sample S and threshold $\kappa \in [0, 1]$, Algorithm 2 terminates and returns a decision tree over LTL_f formula t such that $l(S, t) \leq \kappa$.*

6 Experimental evaluation

6.1 LTL_f inference

In this section, we evaluate the performance of our proposed algorithms and compare them to the SAT-based learning algorithms by Neider and Gavran [27]. Specifically, we compare the following four algorithms: *SAT-flie*: the SAT-based learning algorithms introduced by Neider and Gavran (Algorithm 1 from [27]), *MaxSAT-flie*: our MaxSAT-based algorithm (Algorithm 1), *SAT-DT*: the decision tree-based learning algorithm introduced by Neider and Gavran (Algorithm 2 from [27])[3] and *MaxSAT-DT*: our decision tree learning algorithm (Algorithm 2).

We implement all learning algorithms in a Python tool[4] using Microsoft Z3 [25]. All experiments were conducted on a Debian machine with Intel Xeon E7-8857 CPU at 3GHz using up to 6GB of RAM.

We generate samples based on common LTL_f patterns that can be found in practice [11]. Table 1 lists the set of the LTL_f formulas used for the generation.

In a first sample set (without noise), we generate 148 samples with the generation method proposed by Neider and Gavran [27]. The size of the generated samples ranges between 12 and 1000, consisting of traces of length up to 15. Furthermore, we derive a second sample set from the first one, by introducing 5% noise: for each sample of the first set, we invert the labels of up to 5% of the traces, randomly.

We evaluate the performance of all the algorithms on the two sample sets previously defined. We set a timeout of 900*s* on each run. Table 2 presents the parameters of the algorithms, as well as their respective performances.

We first compare *MaxSAT-flie* (proposed in this paper) and *SAT-flie* (proposed in [27]). With $\kappa = 0.001$, *MaxSAT-flie* performs worse than *SAT-flie*.

This is due to the fact that a MaxSAT problem is computationally more difficult to solve than a SAT problem [15]. For inferring an LTL_f formula exactly classifying a sample, using the SAT problem suffices and thus, *SAT-flie* performs better than *MaxSAT-flie*.

[3] We adapted *SAT-DT* to learn decision trees with a similar stopping criteria as ours.

[4] https://github.com/cryhot/samples2LTL.

Table 1 LTL_f patterns used for generation of samples

Absence	Existence	Universality	Disjunction of common patterns
$\mathbf{G}(\neg p_0)$	$\mathbf{F}(p_0)$	$\mathbf{G}(p_0)$	$\mathbf{G}(\neg p_0) \vee \mathbf{F}(p_0 \wedge \mathbf{F}(p_1)) \vee \mathbf{G}(\neg p_3) \vee \mathbf{F}(p_2 \wedge \mathbf{F}(p_3))$
$\mathbf{F}(p_1) \rightarrow (\neg p_0 \, \mathbf{U} \, p_1)$	$\mathbf{G}(\neg p_0) \vee \mathbf{F}(p_0 \wedge \mathbf{F}(p_1))$	$\mathbf{F}(p_1) \rightarrow (p_0 \, \mathbf{U} \, p_1)$	$\mathbf{F}(p_2) \vee \mathbf{F}(p_0) \vee \mathbf{F}(p_1)$
$\mathbf{G}(p_1 \rightarrow \mathbf{G}(\neg p_0))$	$\mathbf{G}(p_0 \wedge (\neg p_1 \rightarrow (\neg p_1 \, \mathbf{U} (p_2 \wedge \neg p_1))))$	$\mathbf{G}(p_1 \rightarrow \mathbf{G}(p_0))$	$\mathbf{G}(p_0 \wedge (\neg p_1 \rightarrow (\neg p_1 \, \mathbf{U} (p_2 \wedge \neg p_1)))) \vee \mathbf{G}(p_3 \wedge (\neg p_4 \rightarrow (\neg p_4 \, \mathbf{U} (p_5 \wedge \neg p_4))))$

Table 2 Summary of all the tested algorithms—comparison of numbers of timeouts, running times in seconds, inferred formula sizes

Algorithm	Samples without noise			Samples with 5% noise		
	Timeouts	Avg. time	Avg. size	Timeouts	Avg. time	Avg. size
SAT-flie	36/148	293.31	3.76	124/148	780.51	5.96
MaxSAT-flie ($\kappa = 0.001$)	47/148	357.26	3.47	130/148	801.03	4.89
MaxSAT-flie ($\kappa = 0.05$)	27/148	218.46	2.86	87/148	548.65	2.95
MaxSAT-flie($\kappa = 0.1$)	26/148	211.81	2.59	40/148	275.97	2.54
SAT-DT ($\kappa = 0.05$)	51/148	342.35	5.92	127/148	786.16	9.62
MaxSAT-DT ($\kappa = 0.05, \mu = 0.8$)	23/148	174.58	6.77	85/148	543.50	7.05
MaxSAT-DT ($\kappa = 0.05, \mu = 0.6$)	7/148	74.97	30.91	38/148	281.60	56.55

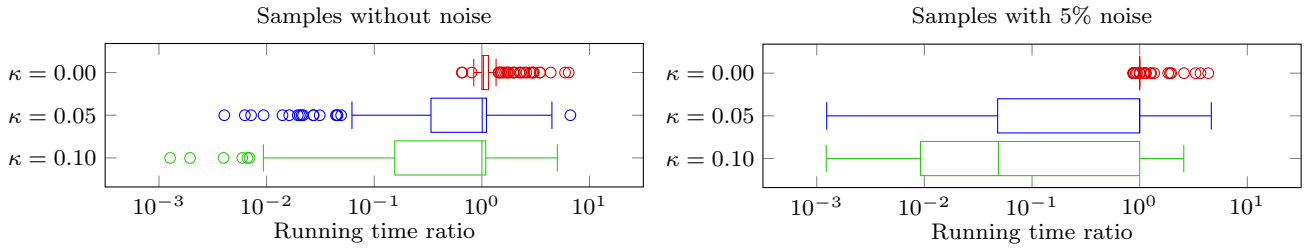


Fig. 3 Comparison of the ratio of the running time of *MaxSAT-flie*(κ) over the running time of *SAT-flie* for all samples in each sample set

For greater values of κ, *MaxSAT-flie* performs better than *SAT-flie*, especially on the samples with noise. To affirm this claim, we calculate the ratio of the running times of *MaxSAT-flie* and *SAT-flie* for each sample of each set (Fig. 3). For example, given a sample S, this ratio would be the running time of *MaxSAT-flie* on S divided by the running time of *SAT-flie* on S. We refer the readers to Appendix 5 for additional figures comparing *MaxSAT-flie* and *SAT-flie*.

We evaluate the size of the inferred LTL_f formula by *MaxSAT-flie* and *SAT-flie* on each sample of each set in Fig. 4. The size of the formula inferred by *MaxSAT-flie* will by design be less than or equal to the size of the formula inferred by *SAT-flie*. As the running time of both algorithms grows exponentially with the number of iterations, it is lower for *MaxSAT-flie* when the inferred formula size is strictly smaller than the size of the formula inferred by *SAT-flie*. However, when both inferred formulas have the same size, there is no running time gain, hence the median running time often being equal to 1 in Fig. 3.

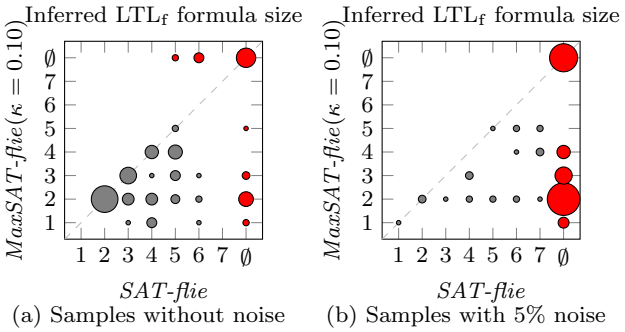


Fig. 4 Inferred LTL_f formula size comparison of *SAT-flie* and *MaxSAT-flie* with threshold $\kappa = 0.10$ on all samples. The surface of a bubble is proportional to the number of samples it represents. The timed out instances are represented by $\emptyset$

We now compare the two algorithms proposed in this paper: did *MaxSAT-DT* perform any better than *MaxSAT-flie*? To be able to compare learned decision trees to learned LTL_f formulas, we measure the size of a tree t in terms of the size of

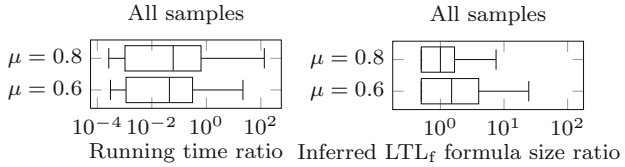


Fig. 5 On each sample (all sample sets combined), comparison of the ratio of the performances of *MaxSAT-DT*(μ) over the performances of *MaxSAT-flie*, with $\kappa = 0.05$ for both algorithms, and where both algorithms did not time out

the formula φ_t this tree encodes. Figure 5 presents a comparison of the running time ratio as well as the inferred formula size ratio of these two algorithms, on each sample of each set that did not time out with both algorithms. We observe that the running time is generally lower for *MaxSAT-DT* than for *MaxSAT-flie*. However, *MaxSAT-DT* tends to infer larger formulas than formulas inferred by *MaxSAT-flie*. This trade-off between running time and inferred formula size is more pronounced for lower values of μ.

Regarding *SAT-DT* (proposed in [27]), we observe a large number of timeouts, especially when evaluated on the samples with 5% noise.

6.2 STL inference

In this section, we propose a second case study and evaluate the performance of our proposed algorithms when adapted to STL formula inference: we present the advantages of *MaxSAT-DT* (decision tree learning algorithm—algorithm 2) compared to *MaxSAT-flie* (MaxSAT-based algorithm—Algorithm 1). We implement both learning algorithms in a C++ tool using Microsoft Z3 [25].

Our samples consist of traces generated by policies learned from reinforcement learning (RL) using *model-based reinforcement learning* (MBRL) algorithm [26]. These traces describe a Pusher-robot that interacts with a ball and a wall. The states of the system are composed of seven features in total, with their corresponding predicates: two Boolean features with corresponding predicates in the form $u_j = \theta$ for $j \in \{1, 2\}$ (for example, $u_1 = 1$ when the ball is in contact with the robot) and five continuous features with corresponding predicates in the form $u_j > \theta$ for $j \in \{3, \ldots, 7\}$ (for example, u_4 represents the total upper arm movement of the Pusher-robot). We note that this system is hybrid, but we simply consider Boolean features as continuous features.

We consider a total of four samples, each of them corresponding to an identified strategy of the Pusher-robot we would like to explain with an STL formula. Each sample contains 300 traces: 150 positive traces from the current strategy, and 150 negative traces from the other three strategies. We set a timeout of 900s on each run.

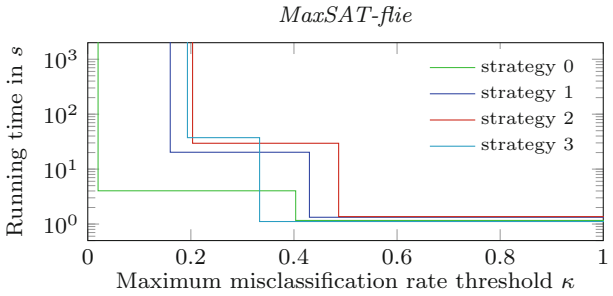


Fig. 6 Impact of the threshold κ on the running time of *MaxSAT-flie*, represented as a step function, for each strategy. Each step corresponds to a certain number of iterations in Algorithm 1, i.e., to an inferred STL formula of a certain size, with a misclassification rate lower than or equal to κ

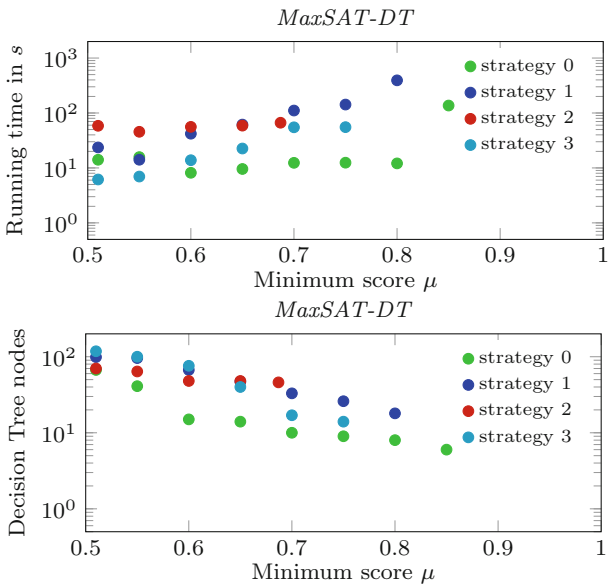


Fig. 7 Impact of the minimum score hyper-parameter μ on the running time and the number of Decision-Tree nodes of *MaxSAT-DT*($\kappa = 0$), for each strategy. Each strategy timed out for μ greater than or equal to 0.9, 0.85, 0.7 and 0.8, respectively

Figure 6 shows the running time of *MaxSAT-flie* for different numbers of iteration in Algorithm 1, presented by misclassification rate. For example, on the strategy 3 sample, we could infer the formula $\mathbf{F}_{[1,3)}\, s_0 = 0$ of size 2 with a misclassification rate of 19.33% (any $\kappa \in [0.1933, 0.3333)$ would have the same effect), with a runtime of 37 seconds. On the same sample, we could infer the formula $(s_5 > 0.003)\, \mathbf{U}_{[1,3)}(s_0 = 0)$ of size 3 with a misclassification rate of 15.67%, with a runtime of 38 minutes (which is way over the chosen timeout but is a good example of non-trivial inferred STL formula).

We run *MaxSAT-DT* on each of the four samples (Fig. 7). *MaxSAT-DT* could produce STL formulas perfectly classifying each sample, i.e., with $\kappa = 0$, where *MaxSAT-flie* timed out for the same κ. Increasing the hyper-parameter μ pro-

duces better quality splits of the sample: this way the number of nodes in the decision tree is reduced, but the running time is increased in return. We observe that the runtime of *MaxSAT-DT* increases in a step function shape when μ increases, in the same manner than the runtime of *MaxSAT-flie* increases when κ decreases, but with more steps: for example, the strategy 2 sample times-out abruptly with $\mu > 0.67$ because one of the decision tree nodes requires now an STL formula of larger size in order to satisfy the criteria.

7 Related work

Inference of LTL formulas The most prominent work in the area of LTL inference is the works by Neider and Gavran [27] (which is the basis of this work) and Camacho et al [10]. Both of these works exploit a SAT-based inference method. While Neider and Gavran use a syntax DAG representation of LTL for the SAT formulation, Camacho et al. use Alternating Finite Automaton (AFA). However, both works suffer from failure when the input sample consists of noise.

The work by Kim et al [19] is prominent work that can infer LTL formulas from noisy samples. They exploit the Bayesian inference problem for inferring satisfactory LTL formulas from noisy data. They, however, rely on templates for the inferred LTL formulas that is often undesirable.

Inference of STL formulas The work of Bombara et al [5] is one of the first works in the inference of Signal Temporal Logic (STL) formulas. Their algorithm also relies on decision trees for inferring STL classifiers. While their algorithm can, in fact, infer STL formulas with arbitrary misclassification error on the data, the STL formulas used for the nodes of the decision trees come from a predefined set.

Another notable work is by Mohammadinejad et al. [24] who present an algorithm for searching STL formulas using enumerative search. They exploit STL grammar to iteratively generate all STL formulas of a particular size. Further, they employ strategies to eliminate equivalent formulas by checking the semantics of STL on the sample. Our work, in contrast, relies on constraint solvers to search for formulas and, thus, will benefit from any advancement in solver technologies.

There are several other works in the general area of STL mining [16,17]. The problem setting of such works is different from ours. In particular, these works aim at extracting STL patterns from data which necessarily need not separate two classes of trace.

Inference of other logics In general, the problem of inferring temporal logic has been in the spotlight for a number of years. Clear evidence of the fact is the variety of temporal logics for which the inference problem has been looked at—Past Time Linear Temporal Logic (PLTL) [2], Property Specification Language (PSL) [33], Interval Temporal Logic [6], and several others [41–43].

8 Conclusion

We developed two novel algorithms for inferring LTL_f/STL formulas from a set of labeled traces/signals allowing misclassifications. Moreover, we demonstrated that our algorithms are efficient in inferring formulas, especially from noisy data, and can be used to interpret AI-generated data. As a part of future work, we like to apply our MaxSAT-based approach for inferring models in other formalisms (e.g., [33]) and perform an extensive evaluation of the algorithms.

Acknowledgements This work has been supported by the Defense Advanced Research Projects Agency (DARPA) (Contract number HR001120C0032), Army Research Laboratory (ARL) (Contract number W911NF2020132 and ACC-APG-RTP W911NF), National Science Foundation (NSF) (Contract Number 1646522), and Deutsche Forschungsgemeinschaft (DFG) (Grant Number 434592664).

Appendix 1 Construction of temporal formulas described in Remarks 1 and 2

LTL_f *formula from Remark 1* To construct a trivial LTL_f formula φ with $l(S,\varphi) = 0$, one needs to perform the following steps: construct formulas $\varphi_{u,v}$ for all $(u,1) \in S$ and $(v,0) \in S$, such that $V(\varphi_{u,v}, u) = 1$ and $V(\varphi_{u,v}, u) = 0$, using a sequence of $\mathbf{X}$-operators and an appropriate propositional formula to describe the first symbol where u and v differ; now $\varphi = \bigvee_{(u,1)\in S} \bigwedge_{(v,0)\in S} \varphi_{u,v}$ is the desired formula.

STL *Formula from Remark 2* With the predicates $\Pi = \{u_j \geq \theta | 1 \leq j \leq m\}$, we construct $\varphi_{u^+,u^-} := \mathbf{F}_{[i,i+1)}\, u_j \geq \frac{u_j^+[i]+u_j^-[i]}{2}$ for all $(u^+,1) \in S$ and $(u^-,0) \in S$, assuming u^+ and u^- differs at time i and coordinate j, ensuring that $V(u^+, \varphi_{u^+,u^-}) \neq V(u, \varphi_{u^+,u^-})$. Without loss of generality, we can ensure that $V(u^+, \varphi_{u^+,u^-}) = 1$ by negating the preceding formula when necessary. Now $\varphi := \bigvee_{(u^+,1)\in S} \bigwedge_{(u^-,0)\in S} \varphi_{u^+,u^-}$ is the desired formula.

Appendix 2 List of all the constraints used

Appendix 2.1 Constraints for learning minimal LTL formula

Structural constraints

$$\Big[\bigwedge_{1\leq i\leq n} \bigvee_{\lambda\in\Lambda} x_{i,\lambda}\Big] \wedge \Big[\bigwedge_{1\leq i\leq n} \bigwedge_{\lambda\neq\lambda'\in\Lambda} \neg x_{i,\lambda} \vee \neg x_{i,\lambda'}\Big] \tag{9}$$

$$[\bigwedge_{2\leq i\leq n} \bigvee_{1\leq j\leq i} l_{i,j}] \wedge [\bigwedge_{2\leq i\leq n} \bigwedge_{1\leq j\leq j'\leq n} \neg l_{i,j} \vee \neg l_{i,j'}] \tag{10}$$

$$[\bigwedge_{2\leq i\leq n} \bigvee_{1\leq j\leq i} r_{i,j}] \wedge [\bigwedge_{2\leq i\leq n} \bigwedge_{1\leq j\leq j'\leq n} \neg r_{i,j} \vee \neg r_{i,j'}] \tag{11}$$

$$\bigwedge_{\substack{2\le i\le n, 1\le j,j'<i \\ \lambda\in\{\mathbf{X},\mathbf{U},\neg,\vee\}}} [x_{i,\lambda} \wedge l_{i,j} \wedge r_{i,j'}] \rightarrow \Big[\bigvee_{\lambda'\in\Lambda} x_{j,\lambda'} \wedge \bigvee_{\lambda'\in\Lambda} x_{j',\lambda'}\Big] \tag{12}$$

$$\bigvee_{p\in\mathcal{P}} x_{1,p} \tag{13}$$

Formula 9 ensures that each node of the syntax DAG has a unique label. Similarly, Formulas 10 and 11 ensure that each node of a syntax DAG has a unique left and right child, respectively. Finally, Formula 13 ensures that Node 1 is labeled by a propositional variable.

Constraints for semantics

$$\bigwedge_{1\le i\le n}\bigwedge_{p\in\mathcal{P}} x_{i,p} \rightarrow \Big[\bigwedge_{0\le\tau<|u|} \begin{cases} y^u_{i,\tau} \text{ if } p\in u[\tau] \\ \neg y^u_{i,\tau} \text{ if } p\notin u[\tau] \end{cases}\Big] \tag{14}$$

$$\bigwedge_{\substack{1\le i\le n \\ 1\le j<i}} x_{i,\neg} \wedge l_{i,j} \rightarrow \Big[\bigwedge_{0\le\tau<|u|} \Big[y^u_{i,\tau} \leftrightarrow \neg y^u_{j,\tau}\Big]\Big] \tag{15}$$

$$\bigwedge_{\substack{1\le i\le n \\ 1\le j,j'<i}} x_{i,\vee} \wedge l_{i,j} \wedge r_{i,j'} \rightarrow \Big[\bigwedge_{0\le\tau<|u|} \Big[y^u_{i,\tau} \leftrightarrow y^u_{j,\tau} \vee y^u_{j',\tau}\Big]\Big] \tag{16}$$

$$\bigwedge_{\substack{1\le i\le n \\ 1\le j<i}} x_{i,\mathbf{X}} \wedge l_{i,j} \rightarrow \Big[\bigwedge_{0\le\tau<|u|-1} \Big[y^u_{i,\tau} \leftrightarrow y^u_{j,\tau+1}\Big]\Big] \tag{17}$$

$$\bigwedge_{\substack{1\le i\le n \\ 1\le j,j'<i}} x_{i,\mathbf{U}} \wedge l_{i,j} \wedge r_{i,j'} \rightarrow \Big[\bigwedge_{0\le\tau<|u|} \Big[y^u_{i,\tau} \leftrightarrow \bigvee_{\tau\le\tau'<|u|} \Big[y^u_{j',\tau'} \wedge \bigwedge_{\tau\le t<\tau'} y^u_{j,t}\Big]\Big]\Big] \tag{18}$$

The constraints are similar to the ones proposed by Neider and Gavran, except that they have been adapted to comply with the semantics of $\mathrm{LTL_f}$. Formula 14 implements the semantics of propositions and states that if Node i is labeled with $p \in \mathcal{P}$, then $y^u_{i,\tau}$ is set to 1 if and only if $p \in u[i]$. Formulas 15 and 16 implement the semantics of negation and disjunction, respectively: if Node i is labeled with $\neg$ and Node j is its left child, then $y^u_{i,\tau}$ equals the negation of $y^u_{j,\tau}$; on the other hand, if Node i is labeled with $\vee$, Node j is its left child, and Node j' is its right child, then $y^u_{i,\tau}$ equals the disjunction of $y^u_{j,\tau}$ and $y^u_{j',\tau}$. Formula 17 implements the semantics of the $\mathbf{X}$-operator and states that if Node i is labeled with $\mathbf{X}$ and its left child is Node j, then $y^u_{i,\tau}$ equals $y^u_{j,\tau+1}$. Finally, Formula 18 implements the semantics of the $\mathbf{U}$-operator; it states that if Node i is labeled with $\mathbf{U}$, its left child is Node j, and its right child is Node j', then $y^u_{i,\tau}$ is set to 1 if and only if there exists a position τ' for which $y^u_{j',\tau'}$ is set to 1 and for all positions t lying between τ and τ', $y^u_{j,t}$ is set to 1.

Appendix 3 Proofs of the theoretical results

Appendix 3.1 Proofs from section 3

Proof of Lemma 1 The hard constraints of Φ^S_n are Φ^{str}_n and Φ^n_u. Now, Φ^{str}_n is satisfiable since there always exists a valid $\mathrm{LTL_f}$ formula of size n. As a result, using the syntax DAG of a $\mathrm{LTL_f}$ formula of size n, we can find an assignment to the variables of Φ^{str}_n that makes it satisfiable. The constraint Φ^n_u, on the other hand, simply tracks the valuation of the prospective formula on traces u. One can easily find an assignment of the variables of Φ^n_u using the semantics of $\mathrm{LTL_f}$.

For proving the second part, let us assume that v is an assignment that satisfies the hard constraints. We now claim that the sum of the weights of the satisfied soft constraints is equal to $1 - wl(S, \varphi_v, \Omega)$. If we can prove this, then if v is an assignment that maximizes the weight of the satisfied soft constraints directly implies that φ_v minimizes the wl function. Now toward proving the claim, we have the following:

$$\begin{aligned} wl(S,\varphi_v,\Omega) &= \sum_{V(\varphi_v,u)\neq b} \Omega(u) \\ &= \sum \Omega(u) - \sum_{V(\varphi_v,u)=b} \Omega(u) \\ &= 1 - \sum_{V(\varphi_v,u)=b} \Omega(u) \\ &= 1 - \sum_{v(y^u_{n,0})=b} \Omega(u) \end{aligned}$$

All the summations appearing in the above equation are over $(u, b) \in S$. Moreover, the quantity $\sum_{v(y^u_{n,0})=b} \Omega(u)$, appearing in the final line, refers to sum of the weights of the satisfied soft constraints, since the constraints in which $v(y^u_{n,0}) = b$ are the ones that are satisfied. □

Proof of Theorem 1 The termination of Algorithm 1 is guaranteed by the fact that there always exists an $\mathrm{LTL_f}$ formula φ for which $wl(\varphi, S, \Omega) = 0$ as indicated by Remark 1. Second, the fact that φ has $wl(\varphi, S, \Omega) \le \kappa$ is a consequence of Lemma 1. Finally, the minimality of the formula is a consequence of the fact that Algorithm 1 searches for an $\mathrm{LTL_f}$ formula in increasing order of size. □

Appendix 3.2 Proofs from Sect. 4

Proof of Lemma 1 in the case of STL The only new constraint in Φ^{str}_n compared to STL case is $0 \le a_i < b_i$ for $i \in \{1, \ldots, n\}$. The constraint Φ^n_u still simply tracks the valuation of the prospective formula on traces u. Thus, all these hard constraints are satisfiables, as explained in Proof 1.

For proving the second part, the weights of the satisfied soft constraints are still equal to $1-wl(S, \varphi_v, \Omega)$. Once again, the proof is similar to Proof 1. □

Proof of Theorem 2 The termination of the MaxSMT-based STL learning algorithm under the conditions of Remark 2 is guaranteed by the fact that there always exists an STL formula φ for which $wl(\varphi, S, \Omega) = 0$, as discussed in the beginning of Sect. 4. Second, the fact that φ has $wl(\varphi, S, \Omega) \leq \kappa$ is a consequence of Proof 2. Finally, the minimality of the formula is guaranteed as explained in proof 1. □

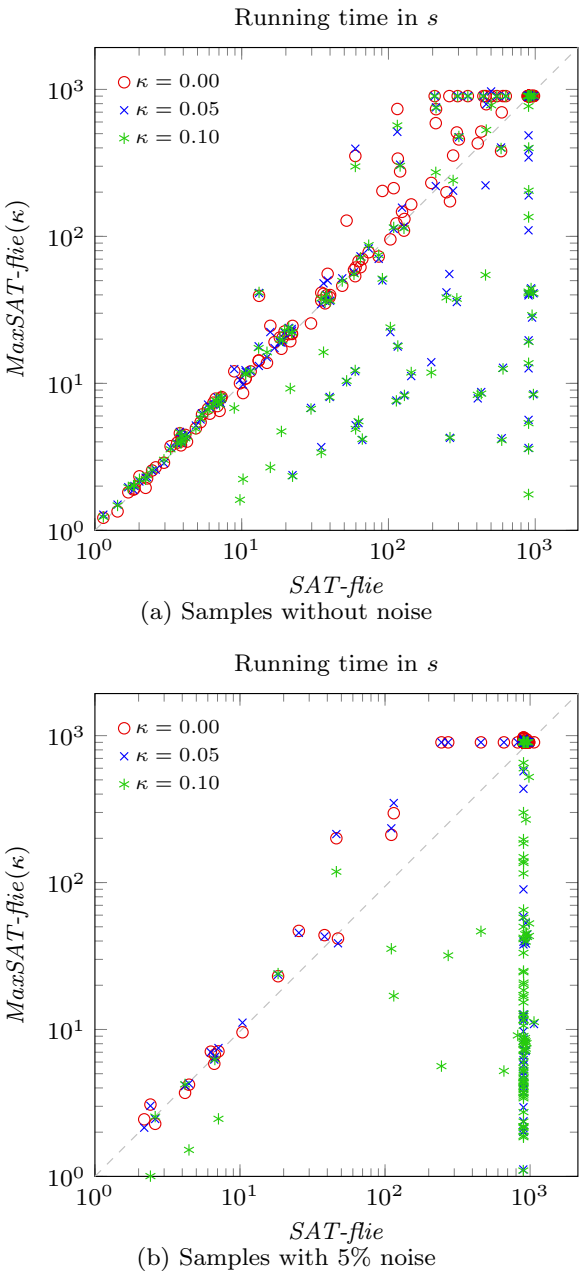
Appendix 3.3 Proofs from Sect. 5

Proof of Lemma 2 Toward contradiction, without loss of generality, let us assume that for all u in S and formula φ with $s_r(S, \varphi) > 0.5$, we have $V(u, \varphi) = 1$. In such a case, $|V(u, \varphi) - b| = 0$ for $(u, 1) \in S$ and $|V(u, \varphi) - b| = 1$ for $(u, 0) \in S$. We can, thus, calculate that $\sum_{(u,1)\in S} |V(u, \varphi) - b| = 0$, $\sum_{(u,0)\in S} |V(u, \varphi) - b| = |\{(u, 0) \in S | b = 0\}|$, and consequently $s_r(S, \varphi) = 0.5$, violating our assumption. □

Proof of Theorem 3 First, observe that at each decision node, we can always infer an LTL_f formula φ for which $s_r(S, \varphi) \geq \mu$, for any value of μ. This is because there always exists an LTL_f formula φ that produces perfect classification, and for this, $s_r(S, \varphi) = 1$. Second, observe that whenever a split is made during the learning algorithm, sub-samples S_1 and S_2 are both non-empty due to Lemma 2. This implies that the algorithm terminates since a sample can be only split finitely many times. Now, for ensuring the decision tree t achieves a $l(S, t) \leq \kappa$, we use induction over the structure of the decision tree. If t is leaf node *true* or *false*, then $l(S, t) \leq \kappa$ using the stopping criteria. Now, say that t is a decision tree with root φ and subtrees t_1 and t_2, meaning $\varphi_t = (\varphi \wedge \varphi_{t_1}) \vee (\neg\varphi \wedge \varphi_{t_2})$. Also, say that the sub-samples produced by φ are S_1 and S_2. By induction hypothesis, we can say that $l(S_1, t_1) \leq \kappa$ and $l(S_2, t_2) \leq \kappa$. Now, it is easy to observe that $l(S_1, (\varphi \wedge \varphi_{t_1})) \leq \kappa$ and $l(S_2, (\neg\varphi \wedge \varphi_{t_2})) \leq \kappa$, since φ satisfies all traces in S_1 and $\neg\varphi$ does not satisfy any trace in S_2. We, thus, have $l(S, t) = l(S_1 \uplus S_2, (\varphi \wedge \varphi_{t_1}) \vee (\neg\varphi \wedge \varphi_{t_2})) \leq \kappa$. □

Appendix 4 Additional theoretical observations

We explain here why Problems 1 and 2 are adapted to Temporal Logic inference from noisy data. Note that when a sample S is constructed from a LTL_f (or equivalently, STL) formula



(a) Samples without noise

(b) Samples with 5% noise

Fig. 8 Running time comparison of *SAT-flie* and *MaxSAT-flie*

ψ of reference (small in size, in principle), i.e., such that $l(S, \psi) = 0$, and given a minimal LTL_f formula φ such that $l(S, \varphi) = 0$, we always have $|\varphi| \leq |\psi|$. However, after introducing noise in the sample such that $l(S', \psi) \lessapprox 0$, and given a minimal formula φ' such that $l(S', \varphi') = 0$, we have no such guarantee on the size of φ'. Intuitively, the size of φ' is growing the more random the classification labels of S' are. However, if we have a bound on the noise, i.e., if we have $l(S', \psi) \leq \kappa$, given a minimal LTL_f formula φ'_κ such that $l(S', \varphi'_\kappa) \leq \kappa$, we can now ensure that $|\varphi'_\kappa| \leq |\psi|$. Hence, Problem 1 is adapted in the context of LTL_f inference from noisy data.

Appendix 5 Experimental results

Figure 8 presents a comparison of the running time of *MaxSAT-flie* (proposed in this paper) and *SAT-flie* (proposed in [27]), on each sample of the LTL_f sample sets.

References

1. Aréchiga N (2019) Specifying safety of autonomous vehicles in signal temporal logic. In: IV, pp 58–63. IEEE
2. Arif MF, Larraz D, Echeverria M, Reynolds A, Chowdhury O, Tinelli C (2020) SYSLITE: syntax-guided synthesis of PLTL formulas from finite traces. In: FMCAD, IEEE, pp 93–103
3. Asarin E, Donzé A, Maler O, Nickovic D (2012) Parametric identification of temporal properties. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 7186 LNCS(September), 147–160. https://doi.org/10.1007/978-3-642-29860-8_12
4. Bacchus F, Kabanza F (2000) Using temporal logics to express search control knowledge for planning. Artif Intell 116(1–2):123–191
5. Bombara G, Vasile CI, Penedo F, Yasuoka H, Belta C (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, ACM, pp 1–10
6. Brunello A, Sciavicco G, Stan IE (2019) Interval temporal logic decision tree learning. In: JELIA, Lecture notes in computer science, vol. 11468, Springer, pp 778–793
7. Budde CE, Argenio PRD, Sedwards S (2018) Qualitative and quantitative trace analysis with extended signal temporal logic. Int J Softw Tools Technol Transf 1:340–358. https://doi.org/10.1007/978-3-319-89963-3
8. Camacho A, Baier JA, Muise CJ, McIlraith SA (2018) Finite LTL synthesis as planning. In: ICAPS, AAAI Press, pp 29–38
9. Camacho A, Icarte RT, Klassen TQ, Valenzano RA, McIlraith SA (2019) LTL and beyond: formal languages for reward function specification in reinforcement learning. In: IJCAI, pp 6065–6073. ijcai.org
10. Camacho A, McIlraith SA (2019) Learning interpretable models expressed in linear temporal logic. In: ICAPS, AAAI Press, pp 621–630
11. Dwyer MB, Avrunin GS, Corbett JC (1998) Property specification patterns for finite-state verification. In: Proceedings of the second workshop on formal methods in software practice, FMSP, Association for Computing Machinery, p 7–15
12. Fainekos GE, Kress-Gazit H, Pappas GJ (2005) Temporal logic motion planning for mobile robots. In: ICRA, IEEE, pp 2020–2025
13. Gabel M, Su Z (2010) Online inference and enforcement of temporal properties. In: ICSE (1), ACM, pp 15–24
14. Giacomo GD, Vardi MY (2013) Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI, IJCAI/AAAI, pp 854–860
15. Halaby ME (2016) On the computational complexity of maxsat. Electron Colloq Comput Complex 23:34
16. Hoxha B, Dokhanchi A, Fainekos G (2018) Mining parametric temporal logic properties in model-based design for cyber-physical systems. Int J Softw Tools Technol Transf 20(1):79–93
17. Jin X, Donzé A, Deshmukh JV, Seshia SA (2013) Mining requirements from closed-loop control models. In: HSCC, ACM, pp 43–52
18. Gaglione JR, Neider D, Roy R, Topcu U, Xu Z (2021) Learning linear temporal properties from noisy data: a MaxSAT-Based approach. In: Automated technology for verification and analysis, Springer International Publishing, pp 74–90. https://doi.org/10.1007/978-3-030-88885-5_6
19. Kim J, Muise C, Shah A, Agarwal S, Shah J (2019) Bayesian inference of linear temporal logic specifications for contrastive explanations. In: IJCAI, pp 5591–5598. ijcai.org
20. Kong Z, Jones A, Belta C (2017) Temporal logics for learning and detection of anomalous behavior. IEEE Trans Autom Control 62(3):1210–1222
21. Lemieux C, Park D, Beschastnikh I (2015) General LTL specification mining (T). In: ASE, IEEE Computer Society. pp 81–92
22. Maler O, Nickovic D (2004) Monitoring temporal properties of continuous signals. In: Proceedings of FORMATS-FTRTFT. Vol. 3253 of LNCS, Springer, pp 152–166
23. Maler O, Nickovic D (2004) Monitoring temporal properties of continuous signals. Lect Notes Comput Sci (Incl Subser Lect Notes Artif Intell Lect Notes Bioinf) 3253:152–166. https://doi.org/10.1007/978-3-540-30206-3_12
24. Mohammadinejad S, Deshmukh JV, Puranic AG, Vazquez-Chanlatte M, Donzé A (2020) Interpretable classification of time-series data using efficient enumerative techniques. In: HSCC, ACM, pp 9:1–9:10
25. de Moura LM, Bjørner N (2008) Z3: an efficient SMT solver. In: TACAS, Lecture notes in computer science, vol. 4963, Springer, pp 337–340
26. Nagabandi A, Konoglie K, Levine S, Kumar V (2019) Deep dynamics models for learning dexterous manipulation, pp 1–12
27. Neider D, Gavran I (2018) Learning linear temporal properties. In: Bjørner N, Gurfinkel A (eds) 2018 Formal methods in computer aided design, FMCAD 2018, IEEE, pp 1–10
28. Pnueli A (1977) The temporal logic of programs. In: Proceedings of 18th annual symposium on foundations of computer science, pp 46–57
29. Pradel M, Gross TR (2012) Leveraging test generation and specification mining for automated bug detection without false positives. In: ICSE, IEEE Computer Society, pp 288–298
30. Pradel M, Jaspan C, Aldrich J, Gross TR (2012) Statically checking API protocol conformance with mined multi-object specifications. In: ICSE, IEEE Computer Society, pp 925–935
31. Quinlan JR (1986) Induction of decision trees. Mach Learn 1(1):81–106
32. Raman V, Donzé A, Sadigh D, Murray RM, Seshia SA (2015) Reactive synthesis from signal temporal logic specifications. In: HSCC, ACM, pp 239–248
33. Roy R, Fisman D, Neider D (2020) Learning interpretable models in the property specification language. In: IJCAI, pp 2213–2219. ijcai.org
34. Sebastiani R, Trentin P (2017) On optimization modulo theories, MaxSMT and sorting networks. CoRR arxiv:1702.02385
35. Shah A, Kamath P, Shah JA, Li S (2018) Bayesian inference of temporal task specifications from demonstrations. In: NeurIPS, pp 3808–3817
36. Tseitin GS (1983) On the Complexity of Derivation in Propositional Calculus, Springer, Berlin Heidelberg, pp 466–483
37. Walkinshaw N, Derrick J, Guo Q (2009) Iterative refinement of reverse-engineered models by model-based testing. In: FM, Lecture notes in computer science, vol. 5850, Springer, pp 305–320
38. Weimer W, Necula GC (2005) Mining temporal specifications for error detection. In: TACAS, Lecture notes in computer science, vol. 3440, Springer, pp 461–476
39. Xu Z, Belta C, Julius A (2015) Temporal logic inference with prior information: An application to robot arm movements. In: IFAC conference on analysis and design of hybrid systems (ADHS), pp 141 – 146
40. Xu Z, Birtwistle M, Belta C, Julius A (2016) A temporal logic inference approach for model discrimination. IEEE Life Sci. Lett. 2(3):19–22

41. Xu Z, Julius AA (2019) Robust temporal logic inference for provably correct fault detection and privacy preservation of switched systems. IEEE Syst. J. 13(3):3010–3021
42. Xu Z, Nettekoven AJ, Agung Julius A, Topcu U (2019) Graph temporal logic inference for classification and identification. In: 2019 IEEE 58th conference on decision and control (CDC), pp 4761–4768
43. Xu Z, Ornik M, Julius AA, Topcu U (2019) Information-guided temporal logic inference with prior knowledge. In: 2019 American control conference (ACC), pp 1891–1897
44. Yang J, Evans D, Bhardwaj D, Bhat T, Das M (2006) Perracotta: mining temporal API rules from imperfect traces. In: ICSE, ACM, pp 282–291