Learning Model Checking and the Kernel Trick for Signal Temporal Logic on Stochastic Processes[⋆]

Luca Bortolussi[1,2](✉), Giuseppe Maria Gallo[1], Jan Křetínský(✉)[3], and Laura Nenzi[1,4](✉)

[1] University of Trieste, Italy
[2] Modelling and Simulation Group, Saarland University, Germany
[3] Technical University of Munich, Germany
[4] University of Technology, Vienna, Austria

Abstract. We introduce a similarity function on formulae of signal temporal logic (STL). It comes in the form of a *kernel function*, well known in machine learning as a conceptually and computationally efficient tool. The corresponding *kernel trick* allows us to circumvent the complicated process of feature extraction, i.e. the (typically manual) effort to identify the decisive properties of formulae so that learning can be applied. We demonstrate this consequence and its advantages on the task of *predicting (quantitative) satisfaction* of STL formulae on stochastic processes: Using our kernel and the kernel trick, we learn (i) computationally efficiently (ii) a practically precise predictor of satisfaction, (iii) avoiding the difficult task of finding a way to explicitly turn formulae into vectors of numbers in a sensible way. We back the high precision we have achieved in the experiments by a theoretically sound PAC guarantee, ensuring our procedure efficiently delivers a close-to-optimal predictor.

1 Introduction

Is it possible to predict the probability that a system satisfies a property without knowing or executing *the system, solely based on previous experience with the system behaviour w.r.t. some* other *properties? More precisely, let $\mathbb{P}_M[\varphi]$ denote the probability that a (linear-time) property φ holds on a run of a stochastic process M. Is it possible to predict $\mathbb{P}_M[\varphi]$ knowing only $\mathbb{P}_M[\psi_i]$ for properties $\psi_1, \ldots, \psi_k$, which were randomly chosen (a-priori, not knowing φ) and thus do not necessarily have any logical relationship, e.g. implication, to φ?*

While this question cannot be in general answered with complete reliability, we show that in the setting of signal temporal logic, under very mild assumptions, it can be answered with high accuracy and low computational costs.

[⋆] This research was funded in part by the German Research Foundation (DFG) project No. 383882557 "Statistical Unbounded Verification" (KR 4890/2-1), by the Austrian FWF projects ZK-35 and by the Italian PRIN project "SEDUCE" n. 2017TWRCNB. Corresponding authors: lbortolussi@units.it, jan.kretinsky@tum.de, lnenzi@units.it

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 281–300, 2022.
https://doi.org/10.1007/978-3-030-99524-9_15

Probabilistic verification and its limits. Stochastic processes form a natural way of capturing systems whose future behaviour is determined at each moment by a unique (but possibly unknown) probability measure over the successor states. The vast range of applications includes not only engineered systems such as software with probabilistic instructions or cyber-physical systems with failures but also naturally occurring systems such as biological systems. In all these cases, predictions of the system behaviour may be required even in cases the system is not (fully) known or is too large. For example, consider a safety-critical cyber-physical system with a third-party component, or a complex signalling pathway to be understood and medically exploited.

Probabilistic model checking, e.g. [4], provides a wide repertoire of analysis techniques, in particular to determine the probability $\mathbb{P}_M[\varphi]$ that the system M satisfies the logical formula φ. However, there are two caveats. Firstly, despite recent advances, [12] the scalability is still quite limited, compared to e.g. hardware or software verification. Moreover, this is still the case even if we only require *approximate* answers, i.e., for a given precision ε, to determine v such that $\mathbb{P}_M[\varphi] \in [v-\varepsilon, v+\varepsilon]$. Secondly, knowledge of the model M is required to perform the analysis.

Statistical model checking [33] fights these two issues at an often acceptable cost of relaxing the guarantee to *probably approximately* correct (PAC), requiring that the approximate answer of the analysis may be incorrect with probability at most δ. This allows for a statistical evaluation: Instead of analyzing the model, we evaluate the satisfaction of the given formula on a number of observed runs of the system and derive a statistical prediction, which is valid only with some confidence. Nevertheless, although M may be unknown, it is still necessary to execute the system in order to obtain its runs.

"Learning" model checking is a new paradigm we propose, in order to fill in a hole in the model-checking landscape where very little access to the system is possible. We are given a set of input-output pairs for model checking, i.e., a collection $\{(\psi_i, p_i)\}_i$ of formulae and their satisfaction values on a given model M, where p_i can be the probability $\mathbb{P}_M[\psi_i]$ of satisfying ψ_i, or its robustness (in case of real-valued logics), or any other quantity. From the data, we learn a predictor for the model checking problem: a classifier for Boolean satisfaction, or a regressor for quantitative domains of p_i. Note that apart from the results on the a-priori given formulae, no knowledge of the system is required; also, no runs are generated and none have to be known. As an example consequence, a user can investigate properties of a system even before buying it, solely based on producer's guarantees on the standardized formulae ψ_i.

Advantages of our approach can be highlighted as follows, not intending to replace standard model checking in standard situations but focusing on the case of extremely limited (i) information and (ii) online resources. *Probabilistic* model checking re-analyzes the system for every new property on the input; *statistical* model checking can generate runs and then, for every new property, analyzes these runs; *learning* model checking performs one analysis with complexity dependent only on the size of the data set (a-priori formulae) and then, for every new formula on input, only evaluates a simple function (whose size is again independent of the system and the property, and depends only on the data set size). Consequently, it has the least access to information and the least computational demands. While lack of any guarantees is typical for machine-learning techniques and, in this context with the lowest resources required, expectable, yet we provide PAC guarantees.

Technique and our approach. To this end, we show how to efficiently learn on the space of temporal formulae via the so-called *kernel trick*, e.g. [32]. This in turn requires to introduce a mapping of formulae to vectors (in a Hilbert space) that preserves the information on the formulae. *How to transform a formula into a vector of numbers (of always the same length)?* While this is not clear at all for finite vectors, we take the dual perspective on formulae, namely as functionals mapping trajectories to values. This point of view provides us with a large bag of functional analysis tools [11] and allows us to define the needed semantic similarity of two formulae (the inner product on the Hilbert space).

Application examples. Having discussed the possibility of learning model checking, the main potential of our kernel (and generally introducing kernels for any further temporal logics) is that it opens the door to *efficient learning on formulae* via kernel-based machine-learning techniques [27,31]. Let us sketch a few further applications that immediately suggest themselves:

Game-based synthesis Synthesis with temporal-logic specifications can often be solved via games on graphs [25,19]. However, exploration of the game graph and finding a winning strategy is done by graph algorithms ignoring the logical information. For instance, choosing between a and $\neg a$ is tried out blindly even for specifications that require us to visit as. Approaches such as [21] demonstrate how to tackle this but hit the barrier of inefficient learning of formulae. Our kernel will allow for learning reasonable choices from previously solved games.

Translating, sanitizing and simplifying specifications A formal specification given by engineers might be somewhat different from their actual intention. Using the kernel, we can, for instance, find the closest simple formula to their inadequate translation from English to logic, which is then likely to match better. (Moreover, the translation would be easier to automate by natural language processing since learning from previous cases is easy once the kernel gives us an efficient representation for formulae learning.)

Requirement mining A topic which received a lot of attention recently is that of identifying specifications from observed data, i.e. to tightly characterize a set of observed behaviours or anomalies [7]. Typical methods are using either formulae templates [6] or methods based e.g. on decision trees [9] or genetic algorithms [28]. Our kernel opens a different strategy to tackle this problem: lifting the search problem from the discrete combinatorial space of syntactic structures of formulae to a continuous space in which distances preserve semantic similarity (using e.g. kernel PCA [27] to build finite-dimensional embeddings of formulae into $\mathbb{R}^m$).

Our main contributions are the following:

- From the technical perspective, we define a kernel function for temporal formulae (of signal temporal logic, see below) and design an efficient way to learn it. This includes several non-standard design choices, improving the quality of the predictor (see Conclusions).
- Thereby we open the door to various learning-based approaches for analysis and synthesis and further applications, in particular also to what we call the *learning* model checking.
- We demonstrate the efficiency practically on predicting the expected satisfaction of formulae on stochastic systems. We complement the experimental results with a theoretical analysis and provide a PAC bound.

1.1 Related Work

Signal temporal logic (STL) [24] is gaining momentum as a requirement specification language for complex systems and, in particular, cyber-physical systems [7]. STL has been applied in several flavours, from runtime-monitoring [7], falsification problems [17] to control synthesis [18], and recently also within learning algorithms, trying to find a maximally discriminating formula between sets of trajectories [9,6]. In these applications, a central role is played by the real-valued quantitative semantics [15], measuring robustness of satisfaction. Most of the applications of STL have been directed to deterministic (hybrid) systems, with less emphasis on non-deterministic or stochastic ones [5].

Metrics and distances form another area in which formal methods are providing interesting tools, in particular logic-based distances between models, like bisimulation metrics for Markov models [2,3,1], which are typically based on a branching logic. In fact, extending these ideas to linear time logic is hard [14], and typically requires statistical approximations. Finally, another relevant problem is how to measure the distance between two logic formulae, thus giving a metric structure to the formula space, a task relevant for learning which received little attention for STL, with the notable exception of [23].

Kernels make it possible to work in a *feature space* of a higher dimension without increasing the computational cost. Feature space, as used in machine learning [31,13], refers to an n-dimensional real space that is the co-domain of a mapping from the original space of data. The idea is to map the original space in a new one that is easier to work with. The so-called *kernel trick*, e.g. [32] allows us to efficiently perform approximation and learning tasks over the feature space without explicitly constructing it. We provide the necessary background information in Section 2.2.

Overview of the paper: Section 2 recalls STL and the classic kernel trick. Section 3 provides an overview of our technique and results. Section 4 then discusses all the technical development in detail. In Section 5, we experimentally evaluate the accuracy of our learning method. In Section 6, we conclude with future work.

2 Background

Let $\mathbb{R}, \mathbb{R}_{\geq 0}, \mathbb{Q}, \mathbb{N}$ denote the sets of non-negative real, rational, and (positive) natural numbers, respectively. For vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ (with $n \in \mathbb{N}$), we write $\boldsymbol{x} = (x_1, \ldots, x_n)$ to access the components of the vectors, in contrast to sequences of vectors $\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots \in \mathbb{R}^n$. Further, we write $\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \sum_{i=1}^{n} x_i y_i$ for the scalar product of vectors.

2.1 Signal Temporal Logic

Signal Temporal Logic (STL) [24] is a linear-time temporal logic suitable to monitor properties of trajectories. A *trajectory* is a function $\xi : I \to D$ with a *time domain* $I \subseteq \mathbb{R}_{\geq 0}$, and a *state space* $D \subseteq \mathbb{R}^n$ for some $n \in \mathbb{N}$. We define the *trajectory space* $\mathcal{T}$ as the set of all possible continuous functions[5] over D. An *atomic predicate* of STL is a continuous computable predicate[6] on $\boldsymbol{x} \in \mathbb{R}^n$ of the form of $f(x_1, ..., x_n) \geq 0$, typically linear, i.e. $\sum_{i=1}^{n} q_i x_i \geq 0$ for $q_1, \ldots, q_n \in \mathbb{Q}$.

Syntax. The set $\mathcal{P}$ of STL formulae is given by the following syntax:

$$\varphi := tt \mid \pi \mid \neg\, \varphi \mid \varphi_1 \, \wedge \, \varphi_2 \mid \varphi_1 \mathbf{U}_{[a,b]} \varphi_2$$

where tt is the Boolean *true* constant, π ranges over atomic predicates, *negation* $\neg$ and *conjunction* $\wedge$ are the standard Boolean connectives and $\mathbf{U}_{[a,b]}$ is the *until* operator, with $a, b \in \mathbb{Q}$ and $a < b$. As customary, we can derive the *disjunction* operator $\vee$ by De Morgan's law and the *eventually* (a.k.a. future) operator $\mathbf{F}_{[t_1,t_2]}$ and the *always* (a.k.a. globally) operator $\mathbf{G}_{[t_1,t_2]}$ operators from the until operator.

Semantics. STL can be given not only the classic Boolean notion of *satisfaction*, denoted by $s(\varphi, \xi, t) = 1$ if ξ at time t satisfies φ, and 0 otherwise, but also a quantitative one, denoted by $\rho(\varphi, \xi, t)$. This measures the quantitative level of satisfaction of a formula for a given trajectory, evaluating how "robust" is the satisfaction of φ with respect to perturbations in the signal [15]. The quantitative semantics is defined recursively as follows:

$$\begin{aligned}
\rho(\pi, \xi, t) &= f_\pi(\xi(t)) \qquad \text{for } \pi(x_1, ..., x_n) = \big(f_\pi(x_1, ..., x_n) \geq 0\big) \\
\rho(\neg\, \varphi, \xi, t) &= -\,\rho(\varphi, \xi, t) \\
\rho(\varphi_1 \, \wedge \, \varphi_2, \xi, t) &= \min\big(\rho(\varphi_1, \xi, t), \rho(\varphi_2, \xi, t)\big) \\
\rho(\varphi_1 \mathbf{U}_{[a,b]} \varphi_2, \xi, t) &= \max_{t' \in [a+t, b+t]} \big(\min\big(\rho(\varphi_2, \xi, t'), \min_{t'' \in [t, t']} \rho(\varphi_1, \xi, t'')\big)\big)
\end{aligned}$$

Soundness and Completeness Robustness is compatible with satisfaction in that it complies with the following soundness property: if $\rho(\varphi, \xi, t) > 0$ then $s(\varphi, \xi, t) = 1$; and if $\rho(\varphi, \xi, t) < 0$ then $s(\varphi, \xi, t) = 0$. If the robustness is 0, both

[5] The whole framework can be easily relaxed to piecewise continuous càdlàg trajectories endowed with the Skorokhod topology and metric [8].

[6] Results are easily generalizable to predicates defined by piecewise continuous càdlàg functions.

satisfaction and the opposite may happen, but either way only non-robustly: there are arbitrarily small perturbations of the signal so that the satisfaction changes[7]. In fact, it complies also with a completeness property that ρ measures how robust the satisfaction of a trajectory is with respect to perturbations, see [15] for more detail.

Stochastic process in this context is a probability space $\mathcal{M} = (\mathcal{T}, \mathcal{A}, \mu)$, where $\mathcal{T}$ is a trajectory space and μ is a probability measure on a σ-algebra $\mathcal{A}$ over $\mathcal{T}$. Note that the definition is essentially equivalent to the standard definition of a stochastic process as a collection $\{D_t\}_{t \in I}$ of random variables, where $D_t(\xi) \in D$ is the signal $\xi(t)$ at time t on ξ [8]. The only difference is that we require, for simplicity[8], the signal be continuous.

Expected robustness and satisfaction probability. Given a stochastic process $\mathcal{M} = (\mathcal{T}, \mathcal{A}, \mu)$, we define the *expected robustness* $R_{\mathcal{M}} : \mathcal{P} \times I \to \mathbb{R}$ as

$$R_{\mathcal{M}}(\varphi, t) := \mathbb{E}_{\mathcal{M}}[\rho(\varphi, \xi, t)] = \int_{\xi \in \mathcal{T}} \rho(\varphi, \xi, t) d\mu(\xi)\,.$$

The qualitative counterpart of the expected robustness is the *satisfaction probability* $S(\varphi)$, i.e. the probability that a trajectory generated by the stochastic process $\mathcal{M}$ satisfies the formula φ: $S_{\mathcal{M}}(\varphi, t) := \mathbb{E}_{\mathcal{M}}[s(\varphi, \xi, t)] = \int_{\xi \in \mathcal{T}} s(\varphi, \xi, t) d\mu(\xi)$.[9] Finally, when $t = 0$ we often drop the parameter t from all these functions.

2.2 Kernel Crash Course

We recall the needed background for readers less familiar with machine learning.

Learning linear models. Linear predictors take the form of a vector of weights, intuitively giving positive and negative importance to features. A predictor given by a vector $\boldsymbol{w} = (w_1, \ldots, w_d)$ evaluates a data point $\boldsymbol{x} = (x_1, \ldots, x_d)$ to $w_1 x_1 + \cdots + w_d x_d = \langle \boldsymbol{w}, \boldsymbol{x} \rangle$. To use it as a classifier, we can, for instance, take the sign of the result and output yes iff it is positive; to use it as a regressor, we can simply output the value. During learning, we are trying to separate, respectively approximate, the training data $\boldsymbol{x_1}, \ldots \boldsymbol{x_N}$ with a linear predictor, which corresponds to solving an optimization problem of the form (f is a suitable loss)

$$\underset{\boldsymbol{w} \in \mathbb{R}^d}{\arg\min}\, f(\langle \boldsymbol{w}, \boldsymbol{x_1} \rangle, \ldots, \langle \boldsymbol{w}, \boldsymbol{x_N} \rangle, \langle \boldsymbol{w}, \boldsymbol{w} \rangle)$$

where the possible, additional last term comes from regularization (preference of simpler weights, with lots of zeros in $\boldsymbol{w}$).

[7] The satisfaction of subformulae changes and, provided the predicates are "independent" of each other, the satisfaction of the whole formula, too.

[8] Again, this assumption can be relaxed since continuous functions are dense in the Skorokhod space of càdlàg functions.

[9] As argued above, this is essentially equivalent to integrating the indicator function of robustness being positive since a formula has robustness exactly zero only with probability zero as we sample all values from continuous distributions.

Need for a feature map Φ : *Input* $\to \mathbb{R}^n$. In order to learn, the input object first needs to be transformed to a vector of numbers. For instance, consider learning the logical exclusive-or function (summation in $\mathbb{Z}_2$) $y = x_1 \oplus x_2$. Seeing true as 1 and false as 0 already transforms the input into elements of $\mathbb{R}^2$. However, observe that there is no linear function separating sets of points $\{(0,0),(1,1)\}$ (where xor returns true) and $\{(0,1),(1,0)\}$ (where xor returns false). In order to facilitate learning by linear classifiers, richer feature space may be needed than what comes directly with the data. In our example, we can design a feature map to a higher-dimensional space using $\Phi : (x_1, x_2) \mapsto (x_1, x_2, x_1 \cdot x_2)$. Then e.g. $x_3 \leq \frac{x_1+x_2-1}{2}$ holds in the new space iff $x_1 \oplus x_2$ and we can learn this linear classifier.

Another example can be seen in Fig. 1. The inner circle around zero cannot be linearly separated from the outer ring. However, considering $x_3 := x_1^2 + x_2^2$ as an additional feature turns them into easily separable lower and higher parts of a paraboloid.

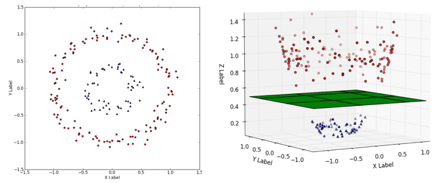

Fig. 1. An example illustrating the need for feature maps in linear classification [20].

In both examples, a feature map Φ mapping the input to a space with higher dimension ($\mathbb{R}^3$), was used. Nevertheless, two issues arise:

1. What should be the features? Where do we get good candidates?
2. How to make learning efficient if there are too many features?

On the one hand, identifying the right features is hard, so we want to consider as many as possible. On the other hand, their number increases the dimension and thus decreases the efficiency both computationally and w.r.t. the number of samples required.

Kernel trick. Fortunately, there is a way to consider a huge amount of features, but with efficiency independent of their number (and dependent only on the amount of training data)! This is called the kernel trick. It relies on two properties of linear classifiers:

– The optimization problem above, after the feature map is applied, takes the form

$$\arg\min_{\boldsymbol{w} \in \mathbb{R}^n} f\big(\langle \boldsymbol{w}, \Phi(\boldsymbol{x_1})\rangle, \ldots, \langle \boldsymbol{w}, \Phi(\boldsymbol{x_N})\rangle, \langle \boldsymbol{w}, \boldsymbol{w}\rangle\big)$$

– Representer theorem: The optimum of the above can be written in the form

$$\boldsymbol{w}^* = \sum_{i=1}^{N} \alpha_i \Phi(\boldsymbol{x_i})$$

Intuitively, anything orthogonal to training data cannot improve precision of the classification on the training data, and only increases $||\boldsymbol{w}||$, which we try to minimize (regularization).

Consequently, plugging the latter form into the former optimization problem yields an optimization problem of the form (g is a suitable loss derived from f):

$$\arg\min_{\boldsymbol{\alpha}\in\mathbb{R}^N} g\big(\boldsymbol{\alpha}, \langle \Phi(\boldsymbol{x_i}), \Phi(\boldsymbol{x_j})\rangle_{1\leq i,j\leq N}\big)$$

In other words, optimizing weights $\boldsymbol{\alpha}$ of expressions where data only appear in the form $\langle \Phi(\boldsymbol{x}_i), \Phi(\boldsymbol{x}_j)\rangle$. Therefore, we can take all features in $\Phi(\boldsymbol{x}_i)$ into account *if*, at the same time, we can efficiently evaluate the kernel function

$$k : (\boldsymbol{x}, \boldsymbol{y}) \mapsto \langle \Phi(\boldsymbol{x}), \Phi(\boldsymbol{y})\rangle$$

i.e. *without* explicitly constructing $\Phi(\boldsymbol{x})$ and $\Phi(\boldsymbol{y})$. Then we can efficiently learn the predictor on the rich set of features. Finally, when the predictor is applied to a new point $\boldsymbol{x}$, we only need to evaluate the expression

$$\langle \boldsymbol{w}, \Phi(\boldsymbol{x})\rangle = \sum_{i=1}^{N} \alpha_i \langle \Phi(\boldsymbol{x_i}), \Phi(\boldsymbol{x})\rangle = \sum_{i=1}^{N} \alpha_i k(\boldsymbol{x_i}, \boldsymbol{x})$$

3 Overview of Our Approach and Results

In this section, we describe what our tasks are if we want to apply the kernel trick in the setting of temporal formulae, what our solution ideas are, and where in the paper they are fully worked out.

1. *Design the kernel function:* define a similarity measure for STL formulae and prove it takes the form $\langle \Phi(\cdot), \Phi(\cdot)\rangle$
   (a) *Design an embedding of formulae into a Hilbert space* (vector space with possibly infinite dimension) ([10], Thm.3 in App.B proves this is well defined): Although learning can be applied also to data with complex structures such as graphs, the underlying techniques typically work on vectors. How do we turn a formula into a vector?
   Instead of looking at the syntax of the formula, we can look at its semantics. Similarly to Boolean satisfaction, where a formula can be identified with its language, i.e., the set $\mathcal{T} \to 2 \cong 2^{\mathcal{T}}$ of trajectories that satisfy it, we can regard an STL formula φ as a map $\rho(\varphi, \cdot) : \mathcal{T} \to \mathbb{R} \cong \mathbb{R}^{\mathcal{T}}$ of trajectories to their robustness. Observe that this is a real function, i.e., an *infinite-dimensional* vector of reals. Although explicit computations with such objects are problematic, kernels circumvent the issue. In summary, we have the implicit features given by the map:
   $$\varphi \overset{\Phi}{\mapsto} \rho(\varphi, \cdot)$$
   (b) *Design similarity on the feature representation (in Sec. 4.1):* Vectors' similarity is typically captured by their scalar product $\langle \boldsymbol{x}, \boldsymbol{y}\rangle = \sum_i x_i y_i$ since it gets larger whenever the two vectors "agree" on a component. In complete analogy, we can define for infinite-dimensional vectors (i.e.

functions) f, g their "scalar product" $\langle f, g\rangle = \int f(x)g(x)\,dx$. Hence we want the kernel to be defined as

$$k(\varphi, \psi) = \langle \rho(\varphi, \cdot), \rho(\psi, \cdot)\rangle = \int_{\xi \in \mathcal{T}} \rho(\varphi, \xi)\rho(\psi, \xi)\,d\xi$$

(c) *Design a measure on trajectories (Sec. 4.2):* Compared to finite-dimensional vectors, where in the scalar product each component is taken with equal weight, integrating over uncountably many trajectories requires us to put a finite measure on them, according to which we integrate. Since, as a side effect, it necessarily expresses their importance, we define a probability measure μ_0 preferring "simple" trajectories, where the signals do not change too dramatically (the so-called total variation is low). This finally yields the definition of the kernel as[10]

$$\boxed{k(\varphi, \psi) = \int_{\xi \in \mathcal{T}} \rho(\varphi, \xi)\rho(\psi, \xi)\,d\mu_0(\xi) \qquad (1)}$$

2. *Learn the kernel (Sec. 5.1):*
   (a) *Get training data $\boldsymbol{x_i}$:* The formulae for training should be chosen according to the same distribution as they are coming in the final task of prediction. Since that distribution is unknown, we assume at least a general preference of simple formulae and thus design a probability distribution $\mathcal{F}_0$, preferring formulae with simple syntax trees (see Section 5.1). We also show that several hundred formulae are sufficient for practically precise predictions.
   (b) *Compute the "correlation" of the data $\langle \phi(\boldsymbol{x_i}), \phi(\boldsymbol{x_j})\rangle$ by kernel $k(\boldsymbol{x_i}, \boldsymbol{x_j})$:* Now we evaluate (1) for all the data pairs. Since this involves an integral over all trajectories, we simply approximate it by Monte Carlo: We choose a number of trajectories according to μ_0 and sum the values for those. In our case, 10 000 provide a very precise approximation.
   (c) *Optimize the weights $\boldsymbol{\alpha}$ (using values from (b)):* Thus we get the most precise linear classifier given the data, but penalizing too "complicated" ones since they tend to overfit and not generalize well (so-called regularization). Recall that the dimension of $\boldsymbol{\alpha}$ is the size of the training data set, not the infinity of the Hilbert space.
3. *Evaluate the predictive power of the kernel* and thus implicitly the kernel function design:
   – We evaluate the accuracy of predictions of robustness for single trajectories (Sec. 5.2), the expected robustness on a stochastic system and the corresponding Boolean notion of satisfaction probability (Sec. 5.3). Moreover, we show that there is no need to derive kernel for each stochastic process separately depending on their probability spaces, but the one

[10] On the conceptual level; technically, additional normalization and Gaussian transformation are performed to ensure usual desirable properties, see Cor. 1 in Sec. 4.1.

derived from the generic μ_0 is sufficient and, surprisingly, even more accurate (Sec. 5.4).
– Besides the experimental evaluation, we provide a PAC bound on our methods in terms of Rademacher complexity [26] (Sec. 4.4).

4 A Kernel for Signal Temporal Logic

In this section, we sketch the technical details of the construction of the STL kernel, of the correctness proof, and of PAC learning bounds. More details on the definition, including proofs, are provided in [10], Appendix B.

4.1 Definition of STL Kernel

Let us fix a formula $\varphi \in \mathcal{P}$ in the STL formulae space and consider the robustness $\rho(\varphi, \cdot, \cdot) : \mathcal{T} \times I \to \mathbb{R}$, seen as a real-valued function on the domain $\mathcal{T} \times I$, where $I \subset \mathbb{R}$ is a bounded interval, and $\mathcal{T}$ is the trajectory space of continuous functions. The STL kernel is defined as follows.

Definition 1. *Fixing a probability measure μ_0 on $\mathcal{T}$, we define the STL-kernel*

$$k'(\varphi, \psi) = \int_{\xi \in \mathcal{T}} \int_{t \in I} \rho(\varphi, \xi, t) \rho(\psi, \xi, t) dt d\mu_0$$

The integral is well defined as it corresponds to a scalar product in a suitable Hilbert space of functions. Formally proving this, and leveraging foundational results on kernel functions [26], in [10], Appendix B, we prove the following:

Theorem 1. *The function k' is a proper kernel function.*

In the previous definition, we can fix time to $t = 0$ and remove the integration w.r.t. time. This simplified version of the kernel is called *untimed*, to distinguish it from the *timed* one introduced above.

In the rest of the paper, we mostly work with two derived kernels, k_0 and k:

$$k_0(\varphi, \psi) = \frac{k'(\varphi, \psi)}{\sqrt{k'(\varphi, \varphi) k'(\psi, \psi)}} \qquad k(x, y) = \exp\left(-\frac{1 - 2k_0(x, y)}{\sigma^2}\right). \tag{2}$$

The normalized kernel k_0 rescales k' to guarantee that $k(\varphi, \varphi) \geq k(\varphi, \psi)$, $\forall \varphi, \psi \in \mathcal{P}$. The Gaussian kernel k, additionally, allows us to introduce a soft threshold σ^2 to fine tune the identification of significant similar formulae in order to improve learning. The following proposition is straightforward in virtue of the closure properties of kernel functions [26]:

Corollary 1. *The functions k_0 and k are proper kernel functions.*

4.2 The Base Measure μ_0

In order to make our kernel meaningful and not too expensive to compute, we endow the trajectory space $\mathcal{T}$ with a probability distribution such that more complex trajectories are less probable. We use the total variation [29] of a trajectory[11] and the number of changes in its monotonicity as indicators of its "complexity".

Because later we use the probability measure μ_0 for Monte Carlo approximation of the kernel k, it is advantageous to define μ_0 algorithmically, by providing a *sampling algorithm*. The algorithm samples from continuous piece-wise linear functions, a dense subset of $\mathcal{T}$, and is described in detail in [10], Appendix A. Essentially, we simulate the value of a trajectory at discrete steps $\varDelta$, for a total of N steps (equal to 100 in the experiments) by first sampling its total variation distance from a squared Gaussian distribution, and then splitting such total variation into the single steps, changing sign of the derivative at each step with small probability q. We then interpolate linearly between consecutive points of the discretization and make the trajectory continuous piece-wise linear.

In Section 5.4, we show that using this simple measure still allows us to make predictions with remarkable accuracy even for other stochastic processes on $\mathcal{T}$.

4.3 Normalized Robustness

Consider the predicates $x_1 - 10 \geq 0$ and $x_1 - 10^7 \geq 0$. Given that we train and evaluate on μ_0, whose trajectories typically take values in the interval $[-3, 3]$ (see also [10], Appendix A), both predicates are essentially equivalent for satisfiability. However, their robustness on the same trajectory differs by orders of magnitude. This very same effect, on a smaller scale, happens also when comparing $x_1 \geq 10$ with $x_1 \geq 20$. In order to ameliorate this issue and make the learning less sensitive to outliers, we also consider a *normalized robustness*, where we rescale the value of the secondary (output) signal to $(-1, 1)$ using a sigmoid function. More precisely, given an atomic predicate $\pi(x_1, ..., x_n) = (f_\pi(x_1, ..., x_n) \geq 0)$, we define $\hat{\rho}(\pi, \xi, t) = \tanh(f_\pi(x_1, ..., x_n))$. The other operators of the logic follow the same rules of the standard robustness described in Section 2.1. Consequently, both $x_1 - 10 \geq 0$ and $x_1 - 10^7 \geq 0$ are mapped to very similar robustness for typical trajectories w.r.t. μ_0, thus reducing the impact of outliers.

4.4 PAC Bounds for the STL Kernel

Probably Approximately Correct (PAC) bounds [26] for learning provide a bound on the generalization error on unseen data (known as risk) in terms of the training loss plus additional terms which shrink to zero as the number of samples grows. These additional terms typically depend also on some measure of the complexity of the class of models we consider for learning (the so-called hypothesis space),

[11] The total variation of function f defined on $[a, b]$ is $V_a^b(f) = \sup_{P \in \mathbf{P}} \sum_{i=0}^{n_P - 1} |f(x_{i+1}) - f(x_i)|$, where $\mathbf{P} = \{P = \{x_0, \ldots, x_{n_P}\} \mid P \text{ is a partition of } [a, b]\}$.

which ought to be finite. The bound holds with probability $1 - \delta$, where $\delta > 0$ can be set arbitrarily small at the price of the bound getting looser.

In the following, we will state a PAC bound for learning with STL kernels for classification. A bound for regression, and more details on the classification bound, can be found in [10], Appendix C. We first recall the definition of the risk L and the empirical risk $\hat{L}$ for classification. The former is an average of the zero-one loss over the data generating distribution p_{data}, while the latter averages over a finite sample D of size m of p_{data}. Formally,

$$L(h) = \mathbb{E}_{\varphi \sim p_{data}} \left[\mathbb{I}\big(h(\varphi) \neq y(\varphi)\big) \right] \quad \text{and} \quad \hat{L}_D(h) = \frac{1}{m} \sum_{i=1}^{m} \mathbb{I}\big(h(\varphi_i) \neq y(\varphi_i)\big),$$

where $y(\varphi)$ is the actual class (truth value) associated with φ, in contrast to the predicted class $h(\varphi)$, and $\mathbb{I}$ is the indicator function.

The major issue with PAC bounds for kernels is that we need to constrain in some way the model complexity. This is achieved by requesting the functions that can be learned have a bounded norm. We recall that the norm $\|h\|_{\mathbb{H}}$ of a function h obtainable by kernel methods, i.e. $h(\varphi) = \sum_{i=1}^{N} \alpha_i k(\varphi_i, \varphi)$, is $\|h\|_{\mathbb{H}} = \boldsymbol{\alpha}^T K \boldsymbol{\alpha}$, where K is the Gram matrix (kernel evaluated between all pairs of input points, $K_{ij} = k(\varphi_i, \varphi_j)$). The following theorem, stating the bounds, can be proved by combining bounds on the Rademacher complexity for kernels with Rademacher complexity based PAC bounds, as we show in [10], Appendix C.

Theorem 2 (PAC bounds for Kernel Learning in Formula Space). *Let k be a kernel (e.g. normalized, exponential) for STL formulae $\mathcal{P}$, and fix $\Lambda > 0$. Let $y : \mathcal{P} \to \{-1, 1\}$ be a target function to learn as a classification task. Then for any $\delta > 0$ and hypothesis function h with $\|h\|_{\mathbb{H}} \leq \Lambda$, with probability at least $1 - \delta$ it holds that*

$$L(h) \leq \hat{L}_D(h) + \frac{\Lambda}{\sqrt{m}} + 3\sqrt{\frac{\log \frac{2}{\delta}}{2m}}. \tag{3}$$

The previous theorem gives us a way to control the learning error, provided we restrict the full hypothesis space. Choosing a value of Λ equal to 40 (the typical value we found in experiments) and confidence 95%, the bound predicts around 650 000 samples to obtain an accuracy bounded by the accuracy on the training set plus 0.05. This theoretical a-priori bound is much larger than the training set sizes in the order of hundreds, for which we observe good performance in practice.

5 Experiments

We test the performance of the STL kernel in predicting (a) robustness and satisfaction on single trajectories, and (b) expected robustness and satisfaction probability estimated statistically from K trajectories. Besides, we test the kernel on trajectories sampled according to the a-priori base measure μ_0 and according to the respective stochastic models to check the generalization power of the generic μ_0-based kernel. Here we report the main results; for additional details as well as plots and tables for further ways of measuring the error, we refer the interested reader to [10], Appendix D.

Computation of the STL robustness and of the kernel was implemented in Python exploiting PyTorch [30] for parallel computation on GPUs. All the experiments were run on a AMD Ryzen 5000 with 16 GB of RAM and on a consumer NVidia GTX 1660Ti with 6 GB of DDR6 RAM. We run each experiment 1000 times for single trajectories and 500 for expected robustness and satisfaction probability where we use 5000 trajectories for each run. Where not indicated differently, each result is the mean over all experiments. Computational time is fast: the whole process of sampling from μ_0, computing the kernel, doing regression for training, test set of size 1000 and validation set of size 200, takes about 10 seconds on GPU. We use the following acronyms: RE = relative error, AE= absolute error, MRE = mean relative error, MAE = mean absolute error, MSE = mean square error.

5.1 Setting

To compute the kernel itself, we sampled 10 000 trajectories from μ_0, using the sampling method described in Section 4.2. As regression algorithm (for optimizing $\boldsymbol{\alpha}$ of Sections 2.2 and 3) we use the *Kernel Ridge Regression* (KRR) [27]. KRR was as good as, or superior, to other regression techniques (a comparison can be found in [10], Appendix D.1).

Training and test set are composed of M formulae sampled randomly according to the measure $\mathcal{F}_0$ given by a syntax-tree random recursive growing scheme (reported in detail in [10], Appendix D.1), where the root is always an operator node and each node is an atomic predicate with probability p_{leaf} (fixed in this experiments to 0.5), or, otherwise, another operator node (sampling the type using a uniform distribution). In these experiments, we fixed $M = 1000$.

Hyperparameters We vary several hyperparameters, testing their impact on errors and accuracy. Here we briefly summarize the results.
- The impact of *formula complexity*: We vary the parameter p_{leaf} in the formula generating algorithm in the range $[0.2, 0.3, 0.4, 0.5]$ (average formula size around $[100, 25, 10, 6]$ nodes in the syntax tree), but only a slight increase in the median relative error is observed for more complex formulae: $[0.045, 0.037, 0.031, 0.028]$.
- The addition of *time bounds* in the formulae has essentially no impact on the performance in terms of errors.
- There is a very small improvement ($< 10\%$) using *integrating signals w.r.t. time* (timed kernel) vs using only robustness at time zero (untimed kernel), but at the cost of a 5-fold increase in computational training time.

- *Size of training set*: The error in estimating robustness decreases as we increase the amount of training formulae, see Fig. 2. However, already for a few hundred formulae, the predictions are quite accurate.

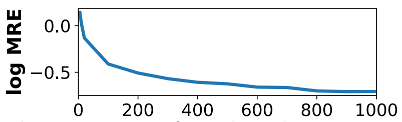

Fig. 2. MRE of predicted average robustness vs the size of the training set.

- *Exponential kernel* k gives a 3-fold improvement in accuracy w.r.t. normalized kernel k_0.
- *Dimensionality of signals*: Error tends to increase linearly with dimensionality. For 1000 formulae in the training set, from dimension 1 to 5, MRE is [0.187, 0.248, 0.359, 0.396, 0.488] and MAE is [0.0537, 0.0735, 0.0886, 0.098, 0.112].

5.2 Robustness and Satisfaction on Single Trajectories

In this experiment, we predict the Boolean satisfiability of a formula using as a discriminator the sign of the robustness. We generate the training and test set of formulae using $\mathcal{F}_0$, and the function sampling trajectories from μ_0 with dimension $n = 1, 2, 3$, using an independent sample than the one for evaluating the kernel. We evaluate the standard robustness ρ and the normalized one $\hat{\rho}$ of each trajectory for each formula in the training and test sets. We then predict ρ and $\hat{\rho}$ for the test set and check if the sign of the predicted robustness agrees with that of the true one, which is a proxy for satisfiability, as discussed previously. Accuracy and distribution of the $\log_{10}$ MRE over all experiments are reported in Fig. 3. Results are good for both but the normalized robustness performs always better. Accuracy is always greater than 0.96 and gets slightly worse when increasing the dimension. We report the mean of quantiles of ρ and $\hat{\rho}$ for RE and AE for n=3 (the toughest case) in Table 1 (top two rows). Errors for the normalized one are also always lower and slightly worsen when increasing the dimension.

In Fig. 4 (left), we plot the true standard robustness for random test formulae in contrast to their predicted values and the corresponding log RE. Here we

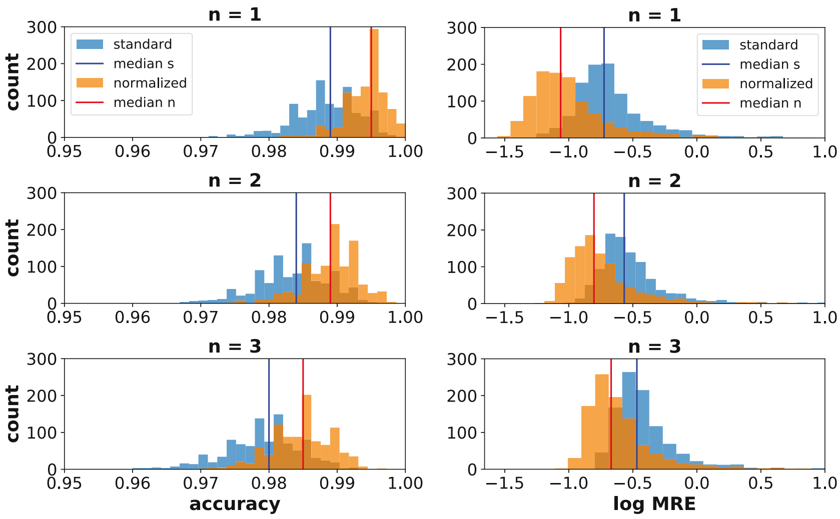

Fig. 3. Accuracy of satisfiability prediction (left) and log_{10} of the MRE (right) over all 1000 experiments for standard and normalized robustness for samples from μ_0 with dimensionality of signals $n = 1, 2, 3$. (Note the logarithmic scale, with log value of -1 corresponding to 0.1 of the standard non-logarithmic scale.)

Table 1. Mean of quantiles for RE and AE over all experiments for prediction of the standard and normalized robustness (ρ, $\hat{\rho}$), expected robustness (R, $\hat{R}$), the satisfaction probability (S) with trajectories sampled from μ_0 and signals with dimensionality n=3, and of the normalized expected robustness on trajectories sampled from *Immigration* (1 dim), *Isomerization* (2 dim), and *Transcription* (3 dim)

	relative error (RE)						absolute error (AE)			
	5perc	1quart	median	3quart	95perc	99perc	1quart	median	3quart	99perc
ρ	0.0035	0.018	0.045	0.141	0.870	4.28	0.016	0.039	0.105	0.689
$\hat{\rho}$	0.0008	0.001	0.006	0.019	0.564	2.86	0.004	0.012	0.039	0.286
R	0.0045	0.021	0.044	0.103	0.548	2.41	0.013	0.029	0.070	0.527
$\hat{R}$	0.0006	0.003	0.007	0.020	0.133	0.55	0.001	0.003	0.007	0.065
S	0.0005	0.003	0.008	0.030	0.586	81.8	0.001	0.003	0.007	0.072
$\hat{R}$ imm	0.0053	0.0067	0.016	0.049	0.360	1.83	0.0037	0.008	0.019	0.151
$\hat{R}$ iso	0.0030	0.0092	0.026	0.091	0.569	2.74	0.0081	0.021	0.057	0.460
$\hat{R}$ trancr	0.0072	0.0229	0.071	0.240	1.490	7.55	0.018	0.049	0.12	0.680

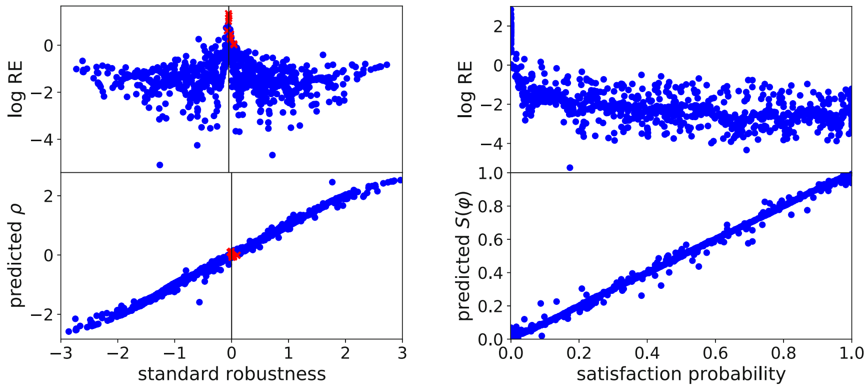

Fig. 4. (left) True standard robustness vs predicted values and RE on single trajectories sampled from μ_0. The misclassified formulae are the red crosses. (right) Satisfaction probability vs predicted values and RE (again for a single experiment).

can clearly observe that the misclassified formulae (red crosses) tend to have a robustness close to zero, where even tiny absolute errors unavoidably produce large relative errors and frequent misclassification.

We test our method also on three specifications of the ARCH-COMP 2020 [16], to show that it works well even on real formulae. We obtain still good results, with an accuracy equal to 1, median AE = 0.0229, and median RE = 0.0316 in the worst case (the AT1 of the Automatic Transmission (AT) Benchmark, see [10], Appendix D.2).

5.3 Expected Robustness and Satisfaction Probability

In these experiments, we approximate the expected standard $R(\varphi)$ and normalized $\hat{R}(\varphi)$ and the satisfaction probability $S(\phi)$ using a fixed set of 5000 trajectories sampled according to μ_0, independent of the one used to compute the kernel, evaluating it for each formula in the training and test sets, and predicting $R(\varphi)$, $\hat{R}(\varphi)$ and $S(\phi)$ for the test set.

For the robustness, the mean of quantiles of RE and AE shows good results as can be seen in Table 1, rows 3–4. Values of MSE, MAE and MRE are smaller than those achieved on single trajectories with medians for n=3 equal to 0.0015, 0.064, and 0.2 for $R(\varphi)$ and 0.00021, 0.0067, and 0.048 for the $\hat{R}(\varphi)$. Normalized robustness continues to outperform the standard one.

For the satisfaction probability, values of MSE and MAE errors are very low, with a median for n=3 equal to 0.000247 for MSE and 0.0759 for MAE. MRE instead is higher and equal to 3.21. The reason can be seen in Fig. 4 (right), where we plot the satisfaction probability vs the relative error for a random experiment. We can see that all large relative errors are concentrated on formulae with satisfaction probability close to zero, for which even a small absolute deviation can cause large errors. Indeed the 95th percentile of RE is still pretty low, namely 0.586 (cf. Table 1, row 5), while we observe the 99th percentile of RE blowing up to 81.8 (at points of near zero true probability). This heavy tailed behaviour suggests to rely on median for a proper descriptor of typical errors, which is 0.008 (hence the typical relative error is less than 1%).

5.4 Kernel Regression on Other Stochastic Processes

The last aspect that we investigate is whether the definition of our kernel w.r.t. the fixed measure μ_0 can be used for making predictions also for other stochastic processes, i.e. without redefining and recomputing the kernel every time that we change the distribution of interest on the trajectory space.

Standardization. To use the same kernel of μ_0 we need to standardize the trajectories so that they have the same scale as our base measure. Standardization, by subtracting to each variable its sample mean and dividing by its sample standard deviation, will result in a similar range of values as that of trajectories sampled from μ_0, thus removing distortions due to the presence of different scales and allowing us to reason on the trajectories using thresholds like those generated by the STL sampling algorithm.

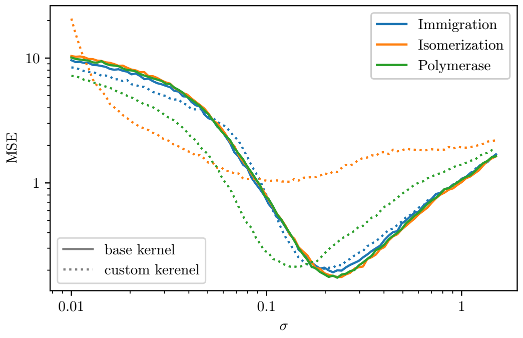

Fig. 5. Expected robustness prediction using the kernel evaluated according to the *base kernel*, and a *custom kernel*. We depict MSE as a function of the bandwidth σ of the Gaussian kernel (with both axes in logarithmic scale).

Performance of base and custom kernel. We consider three different stochastic models: *Immigration* (1 dim), *Isomerization* (2 dim) and *Polymerise* (2 dim), simulated using the Python library StochPy [22] (see also [10], Appendix D.5).

We compare the performance using the kernel evaluated according to the base measure μ_0 (base kernel), and a custom kernel computed replacing μ_0 with the measure on trajectories given by the stochastic model itself. Results show that the base kernel is still the best performing one, see Fig. 5. This can be explained by the fact that the measure μ_0 is broad in terms of coverage of the trajectory space, so even if two formulae are very similar, there will be, with a high probability, a set of trajectories for which the robustnesses of the two formulae are very different. This allows us to better distinguish among STL formulae, compared to models that tend to focus the probability mass on narrower regions of $\mathcal{T}$ as, for example, the Isomerization model, which is the model with the most homogeneous trajectory space and has indeed the worst performance.

Expected Robustness Setting is the same as for the corresponding experiment on μ_0. Instead of the Polymerase model, we consider here a *Transcription* model [22] (see also [10], Appendix D.5), to have also a 3-dimensional model. Results of quantile for RE and AE for the normalized robustness are reported in Table 1, bottom three rows. The results on the different models are remarkably promising, with the Transcription model (median RE 7%) performing a bit worse than Immigration and Isomerization (1.6% and 2.6% median RE). Similar experiments have been done also on single trajectories, where we obtain similar results as for the Expected Robustness [10], Appendix D.5.

6 Conclusions

To enable any learning over formulae, their features must be defined. We circumvented the typically manual and dubious process by adopting a more canonic, infinite-dimensional feature space, relying on the quantitative semantics of STL. To effectively work with such a space, we defined a kernel for STL. To further overcome artefacts of the quantitative semantics, we proposed several normalizations of the kernel. Interestingly, we can use *exactly* the same kernel with a fixed base measure over trajectories across different stochastic models, not requiring any access to the model. We evaluated the approach on realistic biological models from the stochpy library as well as on realistic formulae from Arch-Comp and concluded a good accuracy already with a few hundred training formulae.

Yet smaller training sets are possible through a wiser choice of the training formulae: one can incrementally pick formulae significantly different (now that we have a similarity measure on formulae) from those already added. Such active learning results in a better coverage of the formula space, allowing for a more parsimonious training set. Besides estimating robustness of concrete formulae, one can lift the technique to computing STL-based distances between stochastic models, given by differences of robustness over *all* formulae, similarly to [14]. To this end, it suffices to resort to a dual kernel construction, and build non-linear embeddings of formulae into finite-dimensional real spaces using the kernel-PCA techniques [27]. Our STL kernel, however, can be used for many other tasks, some of which we sketched in Introduction. Finally, to further improve its properties, another direction for future work is to refine the quantitative semantics so that equivalent formulae have the same robustness, e.g. using ideas like in [23].

References

1. Amortila, P., Bellemare, M.G., Panangaden, P., Precup, D.: Temporally extended metrics for markov decision processes. In: SafeAI@AAAI. CEUR Workshop Proceedings, vol. 2301. CEUR-WS.org (2019)
2. Bacci, G., Bacci, G., Larsen, K.G., Mardare, R.: A complete quantitative deduction system for the bisimilarity distance on markov chains. Log. Methods Comput. Sci. **14**(4) (2018)
3. Bacci, G., Bacci, G., Larsen, K.G., Mardare, R., Tang, Q., van Breugel, F.: Computing probabilistic bisimilarity distances for probabilistic automata. In: CONCUR. LIPIcs, vol. 140, pp. 9:1–9:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
4. Baier, C., Katoen, J.P.: Principles of model checking. MIT press (2008)
5. Bartocci, E., Bortolussi, L., Nenzi, L., Sanguinetti, G.: System design of stochastic models using robustness of temporal properties. Theor. Comput. Sci. **587**, 3–25 (2015). https://doi.org/10.1016/j.tcs.2015.02.046, https://doi.org/10.1016/j.tcs.2015.02.046
6. Bartocci, E., Bortolussi, L., Sanguinetti, G.: Data-driven statistical learning of temporal logic properties. In: Proc. of FORMATS. pp. 23–37 (2014)
7. Bartocci, E., Deshmukh, J., Donzé, A., Fainekos, G., Maler, O., Ničković, D., Sankaranarayanan, S.: Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In: Lectures on Runtime Verification, pp. 135–175. Springer (2018)
8. Billingsley, P.: Probability and measure. John Wiley & Sons (2008)
9. Bombara, G., Vasile, C.I., Penedo, F., Yasuoka, H., Belta, C.: A Decision Tree Approach to Data Classification using Signal Temporal Logic. In: Hybrid Systems: Computation and Control. pp. 1–10. ACM Press (2016). https://doi.org/10.1145/2883817.2883843
10. Bortolussi, L., Gallo, G.M., Křetínský, J., Nenzi, L.: Learning model checking and the kernel trick for signal temporal logic on stochastic processes. Tech. Rep. 2201.09928, arXiv (2022), https://arxiv.org/abs/2201.09928
11. Brezis, H.: Functional analysis, Sobolev spaces and partial differential equations. Springer Science & Business Media (2010)
12. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018)
13. Comaniciu, D., Meer, P.: Mean shift: A robust approach toward feature space analysis. IEEE Transactions on Pattern Analysis & Machine Intelligence **24**(5), 603–619 (2002)
14. Daca, P., Henzinger, T.A., Kretínský, J., Petrov, T.: Linear distances between markov chains. In: Desharnais, J., Jagadeesan, R. (eds.) CONCUR. LIPIcs, vol. 59, pp. 20:1–20:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.20, https://doi.org/10.4230/LIPIcs.CONCUR.2016.20
15. Donzé, A., Ferrere, T., Maler, O.: Efficient robust monitoring for stl. In: International Conference on Computer Aided Verification. pp. 264–279. Springer (2013)
16. Ernst, G., Arcaini, P., Bennani, I., Donze, A., Fainekos, G., Frehse, G., Mathesen, L., Menghi, C., Pedrielli, G., Pouzet, M., Yaghoubi, S., Yamagata, Y., Zhang, Z.: Arch-comp 2020 category report: Falsification. In: Frehse, G., Althoff, M. (eds.) ARCH20. 7th International Workshop on Applied Verification of Continuous and

Hybrid Systems (ARCH20). EPiC Series in Computing, vol. 74, pp. 140–152. EasyChair (2020). https://doi.org/10.29007/trr1, https://easychair.org/publications/paper/ps5t

17. Fainekos, G., Hoxha, B., Sankaranarayanan, S.: Robustness of specifications and its applications to falsification, parameter mining, and runtime monitoring with s-taliro. In: Finkbeiner, B., Mariani, L. (eds.) Runtime Verification (RV). Lecture Notes in Computer Science, vol. 11757, pp. 27–47. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_3, https://doi.org/10.1007/978-3-030-32079-9_3
18. Haghighi, I., Mehdipour, N., Bartocci, E., Belta, C.: Control from signal temporal logic specifications with smooth cumulative quantitative semantics. In: 58th IEEE Conference on Decision and Control, CDC 2019, Nice, France, December 11-13, 2019. pp. 4361–4366. IEEE (2019). https://doi.org/10.1109/CDC40024.2019.9029429, https://doi.org/10.1109/CDC40024.2019.9029429
19. Jacobs, S., Bloem, R., Colange, M., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Luttenberger, M., Meyer, P.J., Michaud, T., Sakr, M., Sickert, S., Tentrup, L., Walker, A.: The 5th reactive synthesis competition (SYNTCOMP 2018): Benchmarks, participants & results. CoRR **abs/1904.07736** (2019)
20. Kim, E.: Everything you wanted to know about the kernel trick (but were too afraid to ask). https://www.eric-kim.net/eric-kim-net/posts/1/kernel_trick.html, accessed on Jan 20, 2021
21. Kretínský, J., Manta, A., Meggendorfer, T.: Semantic labelling and learning for parity game solving in LTL synthesis. In: ATVA. Lecture Notes in Computer Science, vol. 11781, pp. 404–422. Springer (2019)
22. Maarleveld, T.R., Olivier, B.G., Bruggeman, F.J.: Stochpy: a comprehensive, user-friendly tool for simulating stochastic biological processes. PloS one **8**(11), e79345 (2013)
23. Madsen, C., Vaidyanathan, P., Sadraddini, S., Vasile, C.I., DeLateur, N.A., Weiss, R., Densmore, D., Belta, C.: Metrics for signal temporal logic formulae. In: 2018 IEEE Conference on Decision and Control (CDC). pp. 1542–1547. IEEE (2018)
24. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Proc. FORMATS (2004)
25. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: CAV (1). Lecture Notes in Computer Science, vol. 10981, pp. 578–586. Springer (2018)
26. Mohri, M., Rostamizadeh, A., Talwalkar, A.: Foundations of machine learning. The MIT Press, Cambridge, Massachusetts, second edition edn. (2018)
27. Murphy, K.P.: Machine learning: a probabilistic perspective. MIT press (2012)
28. Nenzi, L., Silvetti, S., Bartocci, E., Bortolussi, L.: A robust genetic algorithm for learning temporal specifications from data. In: McIver, A., Horváth, A. (eds.) QEST. Lecture Notes in Computer Science, vol. 11024, pp. 323–338. Springer (2018). https://doi.org/10.1007/978-3-319-99154-2_20, https://doi.org/10.1007/978-3-319-99154-2_20
29. Pallara, D Ambrosio, L., Fusco, N.: Functions of bounded variation and free discontinuity problems. Oxford University Press, Oxford (2000)
30. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in pytorch. In: NIPS 2017 Workshop on Autodiff (2017), https://openreview.net/forum?id=BJJsrmfCZ
31. Rasmussen, C.E., Williams, C.K.I.: Gaussian Processes for Machine Learning. MIT Press (2006)

32. Shawe-Taylor, J., Cristianini, N.: Kernel methods for pattern analysis. Cambridge Univ Pr (2004)
33. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: CAV. Lecture Notes in Computer Science, vol. 2404, pp. 223–235. Springer (2002)