Data-Driven Invariant Learning for Probabilistic Programs

Jialu Bao[1], Nitesh Trivedi[2], Drashti Pathak[3], Justin Hsu[1], Subhajit Roy[2]

[1]Cornell University, Ithaca, NY, USA.
[2]Indian Institute of Technology (IIT) Kanpur, India.
[3]Amazon, Bangalore, India.

Contributing authors: jb965@cornell.edu; nitesht@iitk.ac.in; dpathak1997@gmail.com; justin@cs.cornell.edu; subhajit@iitk.ac.in;

Abstract

Morgan and McIver's *weakest pre-expectation* framework is one of the most well-established methods for deductive verification of probabilistic programs. Roughly, the idea is to generalize binary state assertions to real-valued *expectations*, which can measure expected values of probabilistic program quantities. While loop-free programs can be analyzed by mechanically transforming expectations, verifying loops usually requires finding an *invariant expectation*, a difficult task.

We propose a new view of invariant expectation synthesis as a *regression* problem: given an input state, predict the *average* value of the post-expectation in the output distribution. Guided by this perspective, we develop the first *data-driven* invariant synthesis method for probabilistic programs. Unlike prior work on probabilistic invariant inference, our approach can learn piecewise continuous invariants without relying on template expectations. We also develop a data-driven approach to learn *sub-invariants* from data, which can be used to upper- or lower-bound expected values. We implement our approaches and demonstrate their effectiveness on a variety of benchmarks from the probabilistic programming literature.

Keywords: Probabilistic programs, Data-driven invariant learning, Weakest pre-expectations

1 Introduction

Probabilistic programs are imperative programs augmented with a *sampling* command that allows programs to draw from probability distributions. Probabilistic programs provide a natural way to express randomized computations. While the mathematical semantics of such programs is fairly well-understood [1], verification methods remain an active area of research. Existing automated techniques are either limited to specific properties (e.g., [2–5]), or target simpler computational models [6–8].

Reasoning about Expectations.

One of the earliest methods for reasoning about probabilistic programs is through *expectations*. Originally proposed by Kozen [9], expectations generalize standard, binary assertions to quantitative, real-valued functions on program states. Morgan and McIver further developed this idea into a powerful framework for reasoning about probabilistic imperative programs, called the *weakest pre-expectation calculus* [10, 11].

Concretely, Morgan and McIver defined an operator called the *weakest pre-expectation* (wpe), which takes an expectation E and a program P and produces an expectation E' such that $E'(\sigma)$ is the expected value of E in the output distribution $[\![P]\!]_\sigma$. In this way, the wpe operator can be viewed as a generalization of Dijkstra's weakest pre-conditions calculus [12] to probabilistic programs. For verification purposes, the wpe operator has two key strengths. First, it enables reasoning about probabilities and expected values. Second, when P is a loop-free program, it is possible to transform $\mathsf{wpe}(P, E)$ into a form that does not mention the program P via simple, mechanical manipulations, essentially analyzing the effect of the program on the expectation through syntactically transforming E.

However, there is a caveat: the wpe of a loop is defined as a least fixed point, and it is generally difficult to simplify this quantity into a more tractable form. Fortunately, the wpe operator satisfies a *loop rule* that simplifies reasoning about loops: if we can find an expectation I satisfying an *invariant* condition and some additional conditions, then we can easily bound the wpe of a loop. Checking the invariant condition involves analyzing just the body of the loop, rather than the entire loop. Thus, finding invariants is an important obstacle towards automated reasoning about probabilistic programs.

Discovering Invariants.

Two prior works have considered how to automatically infer invariant expectations for probabilistic loops. The first is Prinsys [13]. Using a template with one hole, Prinsys produces a first-order logical formula describing possible substitutions satisfying the invariant condition. While effective for their benchmark programs, the method's reliance on templates is limiting; furthermore, the user must manually solve a system of logical formulas to find the invariant.

The second work, by Chen et al. [14], focuses on inferring polynomial invariants. They apply Lagrange interpolation theorem to find a polynomial invariant. However, many invariants are not polynomials: for instance, an invariant may combine two polynomials piecewise by branching on a Boolean condition.

Our Approach: Invariant Learning.

We take a different approach inspired by data-driven invariant learning for “regular” programs [15, 16]. In these methods, an invariant is seen as a classifier between a set of “good” states that satisfy the specification, and a set of “bad” states that violate the specification. For training data, the program is profiled to collect execution states. Then, the program is executed with a set of inputs to generate the training data. The invariant is synthesized using machine learning algorithms to find a classifier between the “good” and the “bad” states”. Data-driven techniques reduce the reliance on templates, and can treat the program as a black box— the learner only needs to execute the program to gather input and output data. But to extend the data-driven method to the probabilistic setting, there exist significant challenges:

- **Quantitative invariants.** While the logic of expectations resembles the logic of standard assertions, an important difference is that expectations are *quantitative*: they map program states to real numbers, not a binary yes/no. While standard invariant learning is a *classification* task (i.e., predicting a binary label given a program state), our probabilistic invariant learning is closer to a *regression* task (i.e., predicting a number given a program state).
- **Stochastic data.** Standard invariant learning assumes the program behaves like a *function*: a given input state always leads to the same output state. In contrast, a probabilistic program takes an input state to a distribution over outputs. Since we are only able to observe a single draw from the output distribution each time we run the program, execution traces in our setting are inherently *noisy*. Accordingly, we cannot hope to learn an invariant that fits the observed data perfectly, even if the program has an invariant—our learner must be robust to noisy training data.
- **Complex learning objective.** To fit a probabilistic invariant to data, the logical constraints defining an invariant must be converted into a regression problem with a loss function suitable for standard machine learning algorithms and models. While typical regression problems relate the unknown quantity to be learned to known data, the conditions defining invariants are somehow self-referential: they describe how an unknown invariant must be related to itself. This feature makes casting invariant learning as machine learning a difficult task.
- **Quality of examples.** If a candidate invariants fails the verification check, the generated counterexamples are added back to the data-set for learning a better invariant. However, if the generated counterexamples are “close” to valid examples, the generated loss may not be enough to move the learning to a new invariant, thereby stalling progress. To enable progress, we pose the search for counterexamples as an optimization problem to search for the *worst-case* counterexamples that would generate appreciable loss.

Outline.

After covering preliminaries (Section 2) and stating our problem (Section 3), we present our contributions.

- A general method called Exist for learning invariants for almost surely terminating probabilistic programs (Section 4). Exist executes the program multiple times on a set of input states, and then uses machine learning algorithms to learn models encoding possible invariants. A CEGIS-like loop is used to iteratively expand the dataset after encountering incorrect candidate invariants.

- Concrete instantiations of Exist tailored for handling two problems: learning *exact invariants* (Section 5), and learning *sub-invariants* (Section 6). Our method for exact invariants learns a *model tree* [17], a generalization of binary decision trees to regression. The constraints for sub-invariants are more difficult to encode as a regression problem, and our method learns a *neural model tree* [18] with a custom loss function. While the models differ, both algorithms leverage off-the-shelf learning algorithms.
- An implementation of Exist and a thorough evaluation on a large set of benchmarks (Section 7). Our tool can learn invariants and sub-invariants for examples considered in prior work, and more difficult versions that are beyond the reach of prior work.

We discuss related work in Section 8.

2 Preliminaries

Probabilistic Programs.

We will consider programs written in **pWhile**, a basic probabilistic imperative language with the following grammar:

$$P \coloneqq \textbf{skip} \mid x \leftarrow e \mid x \xleftarrow{\$} d \mid P \,;\, P \mid \textbf{if } e \textbf{ then } P \textbf{ else } P \mid \textbf{while } e : P$$

Above, x ranges over a countable set of variables $\mathcal{X}$, e is an expression, and d is a distribution expression. Expressions are interpreted in program states $\sigma : \mathcal{X} \to \mathcal{V}$, which map variables to a set of values $\mathcal{V}$ (e.g., booleans, integers). The semantics of probabilistic programs is defined in terms of distributions. To avoid measure-theoretic technicalities, we assume that $\mathcal{V}$ is countable.

Definition 1. *A (discrete) distribution μ over a countable set S is a function of type $S \to \mathbb{R}_{\geq 0}$ satisfying $\sum_{s \in S} \mu(s) = 1$. We denote the set of distributions over S by* $\mathbf{Dist}(S)$.

Let Σ denote the set of all program states As is standard, programs P are interpreted as maps $[\![P]\!] : \Sigma \to \mathbf{Dist}(\Sigma)$. This definition requires two standard operations on distributions.

Definition 2. *Given a set S,* unit *maps any $s \in S$ to the Dirac distribution on s, i.e.,* $\mathsf{unit}(s)(s') \coloneqq 1$ *if $s = s'$ and* $\mathsf{unit}(s)(s') \coloneqq 0$ *if $s \neq s'$.*

Given $\mu \in \mathbf{Dist}(S)$ and a map $f : S \to \mathbf{Dist}(T)$, the map bind *combines them into a distribution over T,* $\mathsf{bind}(\mu, f) \in \mathbf{Dist}(T)$, *defined via*

$$\mathsf{bind}(\mu, f)(t) \coloneqq \sum_{s \in S} \mu(s) \cdot f(s)(t).$$

The full semantics is presented in Fig. 1; we comment on a few details here. First, given a state σ, we interpret expressions e and distribution expressions d as values

$$
\begin{aligned}
[\![\textbf{skip}]\!]_\sigma &\coloneqq \mathsf{unit}(\sigma) \\
[\![x \leftarrow e]\!]_\sigma &\coloneqq \mathsf{unit}(\sigma[x \mapsto [\![e]\!]_\sigma]) \\
[\![x \xleftarrow{\$} d]\!]_\sigma &\coloneqq \mathsf{bind}([\![d]\!]_\sigma, v \mapsto \mathsf{unit}(\sigma[x \mapsto v])) \\
[\![P_1 \, ; \, P_2]\!]_\sigma &\coloneqq \mathsf{bind}([\![P_1]\!]_\sigma, \sigma' \mapsto [\![P_2]\!]_{\sigma'}) \\
[\![\textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2]\!]_\sigma &\coloneqq \begin{cases} [\![P_1]\!]_\sigma & : [\![e]\!]_\sigma = tt \\ [\![P_2]\!]_\sigma & : [\![e]\!]_\sigma = f\!f \end{cases} \\
[\![\textbf{while } e : P]\!]_\sigma &\coloneqq \lim_{n \to \infty} [\![(\textbf{if } e \textbf{ then } P \textbf{ else skip})^n]\!]_\sigma
\end{aligned}
$$

Fig. 1: Program semantics

$[\![e]\!]_\sigma \in \mathcal{V}$ and distributions over values $[\![d]\!]_\sigma \in \mathbf{Dist}(\mathcal{V})$, respectively; we implicitly assume that all expressions are well-typed. Second, since the program semantics maps Σ to *distributions* over Σ, the semantics for loops is only well-defined when the loop is *almost surely terminating* (AST): from any initial state, the loop terminates with probability 1. Since our data-driven procedure will require running probabilistic programs on concrete inputs, *we assume throughout that all loops are almost surely terminating (AST).* This condition can often be verified using existing methods (e.g., [19–21]).

Weakest Pre-expectation Calculus.

Morgan and McIver's *weakest pre-expectation calculus* reasons about probabilistic programs by manipulating *expectations*.

Definition 3. *Denote the set of program states by Σ. Define the set of expectations, $\mathcal{E}$, to be $\{E \mid E : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}\}$. Define $E_1 \leq E_2$ iff $\forall \sigma \in \Sigma : E_1(\sigma) \leq E_2(\sigma)$. The set $\mathcal{E}$ is a complete lattice.*

While expectations are technically mathematical functions from Σ to the non-negative extended reals, for formal reasoning it is convenient to work with a more restricted syntax of expectations (see, e.g., [22]). We will often view numeric expressions as expectations. Boolean expressions b can also be converted to expectations; we let $[b]$ be the expectation that maps states where b holds to 1, and other states to 0. As an example of our notation, $[flip = 0] \cdot (x+1)$, $x+1$ are two expectations, and we have $[flip = 0] \cdot (x+1) \leq x+1$.

Now, we are ready to introduce Morgan and McIver's *weakest pre-expectation transformer* wpe. In a nutshell, this operator takes a program P and an expectation E to another expectation E', sometimes called the *pre-expectation*. Formally, wpe is defined in Fig. 2. The case for loops involves the least fixed-point (lfp) of $\Phi^{\mathsf{wpe}}_E \coloneqq \lambda X.([e] \cdot \mathsf{wpe}(P, X) + [\neg e] \cdot E)$, the *characteristic function* of the loop with respect to wpe [23]. The characteristic function is Scott-continuous on the complete lattice $\mathcal{E}$, so the least fixed-point exists by the Kleene fixed-point theorem.

The key property of the wpe transformer is that for any program P, $\mathsf{wpe}(P, E)(\sigma)$ is the expected value of E over the output distribution $[\![P]\!]_\sigma$.

$$\mathsf{wpe}(\mathbf{skip}, E) \coloneqq E \qquad\qquad \mathsf{wpe}(x \leftarrow e, E) \coloneqq E[e/x]$$

$$\mathsf{wpe}(x \xleftarrow{\$} d, E) \coloneqq \lambda\sigma. \sum_{v \in \mathcal{V}} [\![d]\!]_\sigma(v) \cdot E[v/x](\sigma) \qquad \mathsf{wpe}(P \,;\, Q, E) \coloneqq \mathsf{wpe}(P, \mathsf{wpe}(Q, E))$$

$$\mathsf{wpe}(\mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q, E) \coloneqq [e] \cdot \mathsf{wpe}(P, E) + [\neg e] \cdot \mathsf{wpe}(Q, E)$$

$$\mathsf{wpe}(\mathbf{while}\ e : P, E) \coloneqq \mathsf{lfp}(\lambda X.\ [e] \cdot \mathsf{wpe}(P, X) + [\neg e] \cdot E)$$

Fig. 2: Morgan and McIver's weakest pre-expectation operator

Theorem 1 (See, e.g., [23]). *For any program P and expectation $E \in \mathcal{E}$,* $\mathsf{wpe}(P, E) = \lambda\sigma. \sum_{\sigma' \in \Sigma} E(\sigma') \cdot [\![P]\!]_\sigma(\sigma')$

Intuitively, the weakest pre-expectation calculus provides a syntactic way to compute the expected value of an expression E after running a program P, except when the program is a loop. For a loop, the least fixed point definition of $\mathsf{wpe}(\mathbf{while}\ e : P, E)$ is hard to compute.

3 Problem Statement

Analogous to when analyzing the weakest pre-conditions of a loop, knowing a loop *invariant* or *sub-invariant* expectation helps one to bound the loop's weakest pre-expectations, but a (sub)invariant expectation can be difficult to find. Thus, we aim to develop an algorithm to automatically synthesize invariants and sub-invariants of probabilistic loops. More specifically, our algorithm tackles the following two problems:

1. **Finding exact invariants:** Given a loop **while** $G : P$ and an expectation postE as input, we want to find an expectation I such that

$$I = \Phi^{\mathsf{wpe}}_{\mathsf{postE}}(I) \coloneqq [G] \cdot \mathsf{wpe}(P, I) + [\neg G] \cdot \mathsf{postE}. \tag{1}$$

   Such an expectation I is an *exact invariant* of the loop with respect to postE.
2. **Finding sub-invariants:** Given a loop **while** $G : P$ and expectations $\mathsf{preE}, \mathsf{postE}$, we aim to learn an expectation I such that

$$I \leq \Phi^{\mathsf{wpe}}_{\mathsf{postE}}(I) \coloneqq [G] \cdot \mathsf{wpe}(P, I) + [\neg G] \cdot \mathsf{postE} \tag{2}$$

$$\mathsf{preE} \leq I. \tag{3}$$

   The first inequality Eq. (2) says that I is a sub-invariant: on states that satisfy G, the value of I lower bounds the expected value of itself after running one loop iteration from initial state, and on states that violate G, the value of I lower bounds the value of postE. The second inequality Eq. (3) says that I is lower-bounded by the given expectation preE.

Note that an exact invariant is a sub-invariant, so one indirect way to solve the second problem is to solve the first problem, and then check $\mathsf{preE} \leq I$. However, we aim to find a more direct approach to solve the second problem because often exact invariants can be complicated and hard to find, while sub-invariants can be simpler and easier to find.

Once we find (sub)-invariants for a loop, we can use the (sub)-invariants to derive provable bounds on the weakest pre-expectation of the loop if the (sub)-invariants satisfy some additional conditions. Prior work has identified various sufficient conditions; we use the conditions identified by Hark et al. [24] because they are relatively easy to check. We use the following corollary of Hark et al. [24, Theorem 38].

Proposition 2. *When E and I are both expectations, if in addition* one of *(a), (b) or (c) holds:*

(a) The number of iterations that **while** $G : P$ *runs is bounded, and* $(\Phi^{\mathsf{wpe}}_E)^n(I)$ *is finite for every* $n \in \mathbb{N}$.

(b) The following four conditions are all satisfied:
- *The expected looping time of* **while** $G : P$ *is finite for every initial state* $s \in \Sigma$,
- $\Phi^{\mathsf{wpe}}_E(I)$ *is finite.*
- *There exists an expectation I' such that* $I = [\neg e] \cdot E + [e] \cdot I$.
- *The conditional difference of the invariant I, i.e.,* $\Delta I \coloneqq \lambda s.([e] \cdot \mathsf{wpe}(P, |I - I(s)|))(s)$ *is bounded by a constant.*

(c) **while** $G : P$ *is almost surely terminating and both I and E are bounded.*

then we have:

$$I \leq \Phi^{\mathsf{wpe}}_E(I) \implies I \leq \mathsf{wpe}(\textbf{while } G : P, E) \tag{4}$$
$$\text{and } I = \Phi^{\mathsf{wpe}}_E(I) \implies I = \mathsf{wpe}(\textbf{while } G : P, E). \tag{5}$$

Proof. For Eq. (4), note that our conditions and the claim are exactly the same as those in Hark et al. [24, Theorem 38].

For Eq. (5), the definition of the weakest pre-condition operator $\mathsf{wpe}(\textbf{while } G : P, E) = \mathsf{lfp}\Phi^{\mathsf{wpe}}_E$ and the Park induction principle $I \geq \Phi^{\mathsf{wpe}}_E(I) \implies I \geq \mathsf{lfp}\Phi^{\mathsf{wpe}}_E$ [25] gives:

$$I \geq \Phi^{\mathsf{wpe}}_E(I) \implies I \geq \mathsf{wpe}(\textbf{while } G : P, E). \tag{6}$$

Combining this implication Eq. (6) with Eq. (4), we get Eq. (5). □

4 Algorithm

We solve both problems with one algorithm, Exist (short for EXpectation Invariant SynThesis). Our data-driven method resembles Counterexample Guided Inductive Synthesis (CEGIS) (see Fig. 3), but differs in two ways. First, candidates are synthesized by fitting a machine learning model to data consisted of program traces starting from random input states. Our target programs are also probabilistic, introducing a second source of randomness to program traces. Second, our approach seeks high-quality counterexamples—violating the target constraints as much as possible—in order to improve synthesis. For synthesizing invariants and sub-invariants, such

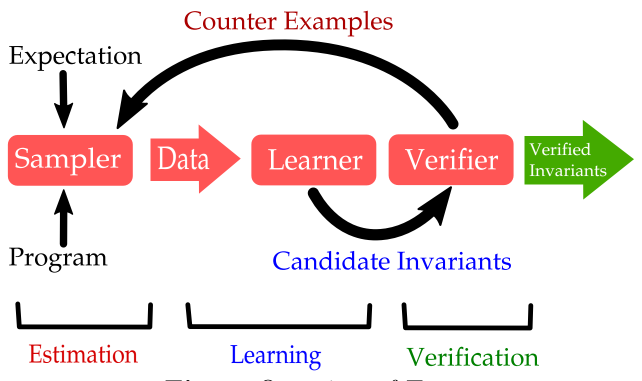


Fig. 3: Overview of Exist

```
Exist(geo, $pexp$, $N_{runs}$, $N_{states}$):
    $feat \leftarrow$ getFeatures(geo, $pexp$)
    $states \leftarrow$ sampleStates($feat$, $N_{states}$)
    $data \leftarrow$ sampleTraces(geo, $pexp$, $feat$, $N_{runs}$, $states$)
    while not timed out:
        $models \leftarrow$ learnInv($feat$, $data$)
        $candidates \leftarrow$ extractInv($models$)
        for $inv$ in $candidates$:
            $verified, cex \leftarrow$ verifyInv($inv$, geo)
            if $verified$:
                return $inv$
            else:
                $states \leftarrow states \cup cex$
                $states \leftarrow states \cup$ sampleStates($feat$, $N'_{states}$)
        $data \leftarrow data \cup$ sampleTraces(geo, $pexp$, $feat$, $nruns$, $states$)
```

Fig. 4: Algorithm Exist

counterexamples can be generated by using a computer algebra system to solve an optimization problem.

In this section, we introduce a meta-algorithm to tackle both problems discussed in Section 3. We will see how to instantiate the meta-algorithm's subroutines in Section 5 and Section 6.

We present the pseudocode in Fig. 4. Exist takes a probabilistic program geo, a post-expectation or a pair of pre/post-expectation $pexp$, and hyper-parameters N_{runs}

and N_{states}. Exist starts by generating a list of features *feat*, which are numerical expressions formed by program variables used in geo. Next, Exist samples N_{states} initialization *states* and runs geo from each of those states for N_{runs} trials, and records the value of *feat* on program traces as *data*. Then, Exist enters a CEGIS loop. In each iteration of the loop, first the learner learnInv trains models to minimize their violation of the required inequalities (e.g., Eq. (2) and Eq. (3) for learning sub-invariants) on *data*. Next, extractInv translates learned models into a set *candidates* of expectations. For each candidate *inv*, the verifier verifyInv looks for program states that *maximize inv*'s violation of required inequalities. If it cannot find any program state where *inv* violates the inequalities, the verifier returns *inv* as a valid invariant or sub-invariant. Otherwise, it produces a set *cex* of counter-example program states, which are added to the set of initial states. Finally, before entering the next iteration, the algorithm augments *states* with a new batch of N'_{states} initial states, generates trace data from running geo on each of these states for N_{runs} trials, and augments the dataset *data*. This data augmentation ensures that the synthesis algorithm collects more and more initial states, some randomly generated (sampleStates) and some from prior counterexamples (*cex*), guiding the learner towards better candidates. Like other CEGIS-based tools, our method is sound but not complete, i.e., if the algorithm returns an expectation then it is guaranteed to be an exact invariant or sub-invariant, but the algorithm might never return an answer; in practice, we set a timeout.

5 Learning Exact Invariants

In this section, we detail how we instantiate Exist's subroutines to learn an exact invariant I satisfying $I = \Phi^{\mathsf{wpe}}_{\mathsf{postE}}(I)$, given a loop geo and an expectation $pexp = \mathsf{postE}$.

At a high level, we first sample a set of program states *states* using sampleStates. From each program state $s \in states$, sampleTraces executes geo and estimates $\mathsf{wpe}(\mathsf{geo}, \mathsf{postE})(s)$. Next, learnInv trains regression models M to predict the estimated $\mathsf{wpe}(\mathsf{geo}, \mathsf{postE})(s)$ given the value of features evaluated on s. Then, extractInv translates the learned models M to an expectation I. In an ideal scenario, this I would be equal to $\mathsf{wpe}(\mathsf{geo}, \mathsf{postE})$, which is also always an exact invariant. But since I is learned from stochastic data, it may be noisy. So, we use verifyInv to check whether I satisfies the invariant condition $I = \Phi^{\mathsf{wpe}}_{\mathsf{postE}}(I)$.

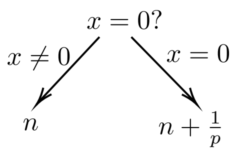
The reader may wonder why we took this complicated approach, first estimating the weakest pre-expectation of the loop, and then computing the invariant: If we are able to learn an expression for $\mathsf{wpe}(\mathsf{geo}, \mathsf{postE})$ directly, then why are we interested in the invariant I? The answer is that it is easier to verify if an I is an invariant than to check whether and I is the least fixed point. Once we verify that I is an invariant that additionally satisfies conditions in Proposition 2, then we also know that $I = \mathsf{wpe}(\mathsf{geo}, \mathsf{postE})$. Since our learning process is inherently noisy, this verification step is crucial and motivates why we want to find an invariant.

A running example.

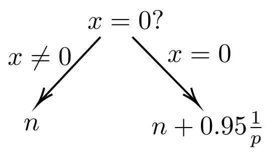
We will illustrate our approach using Fig. 5. The simple program geo repeatedly loops: whenever x becomes non-zero we exit the loop; otherwise we increase n by 1 and draw

```
while $x = 0$ :
    $n \leftarrow n + 1$;
    $x \xleftarrow{\$}$ Bernoulli($p$)
```

(a) Program: geo



(b) Model tree for $\mathsf{wpe}(\mathsf{geo}, n)$



(c) Another model tree

Fig. 5: Running example: Program and model tree

x from a biased coin-flip distribution (x gets 1 with probability p, and 0 otherwise). We aim to learn $\mathsf{wpe}(\mathsf{geo}, n)$, which is $[x \neq 0] \cdot n + [x = 0] \cdot (n + \frac{1}{p})$.

Our Regression Model.

Before getting into how Exist collects data and trains models, we introduce the class of regression models it uses – *model trees*, a generalization of decision trees to regression tasks [17]. Model trees are naturally suited to expressing piecewise functions of inputs, and are straightforward to train. While our method can in theory generalize to other regression models, our implementation focuses on model trees.

More formally, a model tree $T \in \mathcal{T}$ over features $\mathcal{F}$ is a full binary tree where each internal node is labeled with a predicate ϕ over variables from $\mathcal{F}$, and each leaf is labeled with a real-valued model $M \in \mathcal{M} : \mathbb{R}^{\mathcal{F}} \to \mathbb{R}$. Given a feature vector in $x \in \mathbb{R}^{\mathcal{F}}$, a model tree T over $\mathcal{F}$ produces a numerical output $T(x) \in \mathbb{R}$ as follows:

- If T is of the form $\mathsf{Leaf}(M)$, then $T(x) \coloneqq M(x)$.
- If T is of the form $\mathsf{Node}(\phi, T_L, T_R)$, then $T(x) \coloneqq T_R(x)$ if the predicate ϕ evaluates to true on x, and $T(x) \coloneqq T_L(x)$ otherwise.

Throughout this paper, we consider model trees of the following form as our regression model. First, node predicates ϕ are of the form $f \bowtie c$, where $f \in \mathcal{F}$ is a feature, $\bowtie \in \{<, \leq, =, >, \geq\}$ is a comparison, and c is a numeric constant. Second, leaf models on a model tree are either all *linear models* or all products of constant powers of features, which we call *multiplication models*. For example, assuming $n, \frac{1}{p}$ are both features, Fig. 5b and Fig. 5c are two model trees with linear leaf models, and Fig. 5b expresses the weakest pre-expectation $\mathsf{wpe}(\mathsf{geo}, n)$. Formally, the leaf model M on a feature vector f is either

$$M_l(f) = \sum_{i=1}^{|\mathcal{F}|} \alpha_i \cdot f_i \qquad \text{or} \qquad M_m(f) = \prod_{i=1}^{|\mathcal{F}|} f_i^{\alpha_i}$$

with constants $\{\alpha_i\}_i$. Note that multiplication models can also be viewed as linear models on logarithmic values of features because $\log M_m(f) = \sum_{i=1}^{|\mathcal{F}|} \alpha_i \cdot \log(f_i)$. While it is also straightforward to adapt our method to other leaf models, we focus on linear models and multiplication models because of their simplicity and expressiveness. Linear models and multiplication models also complement each other in their expressiveness: encoding expressions like $x + y$ uses simpler features with linear models (it suffices if $\mathcal{F} \ni x, y$, as opposed to needing $\mathcal{F} \ni x + y$ if using multiplicative models), while encoding $\frac{p}{1-p}$ uses simpler features with multiplicative models (it suffices if $\mathcal{F} \ni p, 1 - p$, as opposed to needing $\mathcal{F} \ni \frac{p}{1-p}$ if using linear models).

5.1 Generate Features (getFeatures)

Given a program, the algorithm first generates a set of features $\mathcal{F}$ that model trees can use to express unknown invariants of the given loop. For example, for geo, $I = [x \neq 0] \cdot n + [x = 0] \cdot (n + \frac{1}{p})$ is an invariant, and to have a model tree (with linear/multiplication leaf models) express I, we want $\mathcal{F}$ to include both n and $\frac{1}{p}$, or $n + \frac{1}{p}$ as one feature. $\mathcal{F}$ should include the program variables at a minimum, but it is often useful to have more complex features too. While generating more features increases the expressivity of the models, and richness of the invariants, there is a cost: the more features in $\mathcal{F}$, the more data is needed to train a model.

Starting from the program variables, getFeatures generates two lists of features, $\mathcal{F}_l$ for linear leaf models and $\mathcal{F}_m$ for multiplication leaf models. Intuitively, linear models are more expressive if the feature set $\mathcal{F}$ includes some products of terms, e.g., $n \cdot p^{-1}$, and multiplication models are more expressive if $\mathcal{F}$ includes some sums of terms, e.g., $n + 1$.

We assume program variables and optional user-supplied features *opt* are typed as probabilities (denoted using p_i), integers (denoted using n_i), booleans (denoted using b_i), or reals (denoted using x_i). In general, we do not restrict the integers n_i and reals x_i to be non-negative as our learning algorithm does not assume non-negativity; later, though, to ensure that what Exist generates are indeed expectations according to Definition 3, we assume variables in specific programs to be non-negative. Then, given program variables and user-supplied features $p_i, \ldots, n_i, \ldots, b_i, \ldots, x_i, \ldots$, a loop with guard G, and post expectation *pexp*, getFeatures generates

$$\begin{aligned}
\mathcal{F}_l &\ni G \mid pexp \mid p_i \mid n_i \mid b_i \mid x_i \mid p_i \cdot p_j \mid n_i \cdot n_j \mid x_i \cdot x_j \mid n_i \cdot x_j \mid b_i \cdot b_j \\
\mathcal{F}_m &\ni G \mid pexp \mid p_i \mid n_i \mid b_i \mid x_i \mid 1 + p_j \mid 1 - p_j \mid p_i + p_j \mid p_i + p_j - (p_i \cdot p_j) \\
&\quad \mid n_i + n_j \mid n_i - n_j \mid x_i + x_j \mid x_i - x_j \mid n_i + x_j \mid n_i - x_j \mid b_i + b_j \mid b_i - b_j .
\end{aligned}$$

5.2 Sample Initial States (sampleStates)

Recall that Exist aims to learn an expectation I that is equal to the weakest pre-expectation wpe(**while** $G : P$, postE). A natural idea for sampleTraces is to run the program from all possible initializations multiple times, and record the average value of postE from each initialization. This would give a map close to wpe(**while** $G : P$, postE) if we run enough trials so that the empirical mean is approximately the actual mean. However, this strategy is clearly impractical—many of the programs we consider have infinitely many possible initial states (e.g., programs with integer variables). Thus, sampleStates needs to choose a manageable number of initial states for sampleTraces to use.

In principle, a good choice of initializations should exercise as many parts of the program as possible. For instance, for geo in Fig. 5, if we only try initial states satisfying

$x \neq 0$, then it is impossible to learn the term $[x = 0] \cdot (n + \frac{1}{p})$ in $\mathsf{wpe}(\mathsf{geo}, n)$ from data. However, covering the control flow graph may not be enough. Ideally, to learn how the expected value of postE depends on the initial state, we also want data from multiple initial states along each path.

While it is unclear how to choose initializations to ensure optimal coverage, our implementation uses a simpler strategy: sampleStates generates N_{states} states in total, each by sampling the value of every program variable uniformly at random from a space. We assume program variables are typed as booleans, integers, probabilities, or floating point numbers and sample variables of some type from the corresponding space. For boolean variables, the sampling space is simply $\{0, 1\}$; for probability variables, the space includes reals in some interval bounded away from 0 and 1, because probabilities too close to 0 or 1 tend to increase the variance of programs (e.g., making some loops iterate for a very long time); for floating point number and integer variables, the spaces are respectively reals and integers in some bounded range. This strategy, while simple, is already very effective in nearly all of our benchmarks (see Section 7), though other strategies are certainly possible (e.g., performing a grid search of initial states from some space).

5.3 Sample Training Data (sampleTraces)

We gather training data by running the given program geo on the set of initializations generated by sampleStates. From each program state $s \in states$, the subroutine sampleTraces runs geo for N_{runs} times to get output states $\{s_1, \ldots, s_{N_{runs}}\}$ and produces the following training example (s, v), where

$$v = \frac{1}{N_{runs}} \sum_{i=1}^{N_{runs}} \mathsf{postE}(s_i).$$

Thus, the value v is the empirical mean of postE in the output state of running geo from initial state s_i; as N_{runs} grows large, this average value approaches the true expected value $\mathsf{wpe}(\mathsf{geo}, \mathsf{postE})(s)$.

5.4 Learning a Model Tree (learnInv)

Now that we have the training set $data = \{(s_1, v_1), \ldots, (s_K, v_K)\}$ (where $K = N_{states}$), we want to fit a model tree T to the data. We aim to apply off-the-shelf tools that can learn model trees with customizable leaf models and loss. For each data entry, v_i approximates $\mathsf{wpe}(\mathsf{geo}, \mathsf{postE})(s_i)$, so a natural idea is to train a model tree T that takes the value of features on s_i as input and predicts v_i. To achieve that, we want to define the loss to measure the error between predicted values $T(\mathcal{F}_l(s_i))$ (or $T(\mathcal{F}_m(s_i))$) and the target value v_i. Without loss of generality, we can assume our invariant I is of the form

$$I = \mathsf{postE} + [G] \cdot I' \tag{7}$$

because I being an invariant means

$$I = [\neg G] \cdot \mathsf{postE} + [G] \cdot \mathsf{wpe}(P, I) = \mathsf{postE} + [G] \cdot (\mathsf{wpe}(P, I) - \mathsf{postE}).$$

In many cases, the expectation $I' = \mathsf{wpe}(P, I) - \mathsf{postE}$ is simpler than I: for example, the weakest pre-expectation of geo can be expressed as $n + [x = 0] \cdot (\frac{1}{p})$; while I is represented by a tree that splits on the predicate $[x = 0]$ and needs both $n, \frac{1}{p}$ as features, the expectation $I' = \frac{1}{p}$ is represented by a single leaf model tree that only needs p as a feature. Also, since we are rewriting I in terms of I' and post only for the convenience of model-fitting, the I' here does not have to be an expectation, i.e., it could map states to negative values.

Aiming to learn weakest pre-expectations I in the form of Eq. (7), Exist trains model trees T to fit I'. More precisely, learnInv trains a model tree T_l with linear leaf models over features $\mathcal{F}_l$ by minimizing the loss

$$err_l(T_l, data) = \left(\sum_{i=1}^{K} (\mathsf{postE}(s_i) + G(s_i) \cdot T_l(\mathcal{F}_l(s_i)) - v_i)^2 \right)^{1/2}, \tag{8}$$

where $\mathsf{postE}(s_i)$ and $G(s_i)$ represents the value of expectation postE and G evaluated on the state s_i. This loss measures the sum error between the prediction $\mathsf{postE}(s_i) + G(s_i) \cdot T_l(\mathcal{F}_l(s_i))$ and target v_i. Note that when the guard G is false on an initial state s_i, the example contributes zero to the loss because $\mathsf{postE}(s_i) + G(s_i) \cdot T_l(\mathcal{F}_l(s_i)) = \mathsf{postE}(s_i) = v_i$; thus, we only need to generate and collect trace data for initial states where the guard G is true.

Analogously, learnInv trains a model tree T_m with multiplication leaf models over features $\mathcal{F}_m$ to minimize the loss $err_m(T_m, data)$, which is the same as $err_l(T_l, data)$ except $T_l(\mathcal{F}_l(s_i))$ is replaced by $T_m(\mathcal{F}_m(s_i))$ for each i.

5.5 Extracting Expectations from Models (extractInv)

Given the learned model trees T_l and T_m, we extract expectations that approximate $\mathsf{wpe}(\mathsf{geo}, \mathsf{postE})$ in three steps:

1. **Round T_l, T_m with different precisions.** Since we obtain the model trees T_l and T_m by learning and the training data is stochastic, the coefficients of features in T_l and T_m may be slightly off, so we apply several rounding schemes to generate a list of rounded model trees.

   For T_l, the learned model tree with linear leaf models, we round its coefficients to integers, one digit, and two digits respectively and get T_{l0}, T_{l1}, T_{l2}. For instance, rounding the model tree depicted in Fig. 5c to integers gives us the model tree in Fig. 5b. For T_m, the learned model tree with multiplication leaf models , we construct T_{m0}, T_{m1}, T_{m2} by rounding the leading constant coefficient to respective digits and all other exponentiating coefficients to integers: $c \cdot \prod_{i=1}^{|\mathcal{F}|} x_i^{a_i}$ gets rounded to $round(c, digit) \cdot \prod_{i=1}^{|\mathcal{F}|} x_i^{\mathrm{int}(a_i)}$.

2. **Translate into expectations.** Since we learn model trees, this step is straightforward: for example, $n+\frac{1}{p}$ can be seen as a model tree (with only a leaf) mapping the values of features $n, \frac{1}{p}$ to a number, or an expectation mapping program states where n, p are program variables to a number. We translate each model tree obtained from the previous step to an expectation.
3. **Form the candidate invariant.** Since we train the model trees to fit I' so that $\mathsf{postE} + [G] \cdot I'$ approximates $\mathsf{wpe}(\textbf{while } G : P, \mathsf{postE})$, we construct each candidate invariant $\mathit{inv} \in \mathit{invs}$ by replacing I' in the pattern $\mathsf{postE} + [G] \cdot I'$ by an expectation obtained in the second step.

5.6 Verify Extracted Expectations (verifyInv)

Recall that geo is a loop **while** $G : P$, and given a set of candidate invariants *invs*, we want to check if any $\mathit{inv} \in \mathit{invs}$ is a loop invariant, i.e., if *inv* satisfies

$$\mathit{inv} = [\neg G] \cdot \mathsf{postE} + [G] \cdot \mathsf{wpe}(P, \mathit{inv}). \tag{9}$$

Since the learned model might not predict the expected value for every data point exactly, we must verify whether *inv* satisfies this equality using verifyInv.

5.7 Search for *worst-case* counterexamples

If the invariant is not verified, verifyInv has to look for counterexamples that violate the conditions for valid invariants. These counterexamples are fed back to augment the dataset to (hopefully) learn a better invariant.

In this process, we hope that our data augmentation leads to *substantial* loss in the learning algorithm so as to steer the learning to a new invariant. We provide an example on the challenges of such an endeavor. Though our algorithm uses regression, let us use classification for ease of discussion and visualization. Figure 6a shows the case of learning a linear classifier: we are attempting to learn a classifier that separates the red and gray regions; the stars and circles are the data-points corresponding to the red and grey regions, respectively. Given the current dataset, we may learn an (incorrect) classifier, as shown by the dotted line.

Figure 6b shows the case where a new counterexample is found (counterexamples are shown with green boundary), that is used to augment the dataset. As the counterexample is quite close to the current classifier's decision boundary, it may not create enough loss to bulge the decision boundary. There are two possible solutions to this problem:

- **Engineer a classifier loss function that is more sensitive to violations.** We may engineer the loss function to penalize violations heavily. However, as our dataset is generated from estimates from a finite set of observations, there exist some amount of noise in the estimates. As our dataset is itself noisy, this option is not desirable as we would like our learnt classifier to be robust to noise. Further, such loss functions that are too sensitive to small violations can lead to instability in the learning process, making it difficult to converge.

- **Improve data augmentation.** Instead, we attempt to improve our data augmentation process in two ways: firstly, to attempt generate *better* datapoints for augmentation by searching for *worst-case* counterexamples, i.e. counterexamples that generate a highest possible loss on the objective function used to train the regression model (see Figure 6c). Secondly, we generate a set of counterexamples so that the *cumulative* loss is high enough to ensure progress towards the desirable invariant (see Figure 6d).

Hence, instead of searching for any counterexample that causes a violation, verifyInv searches for a set of counterexamples that maximizes the violation in order to drive the learning process forward in the next iteration. Formally, for every $inv \in invs$, verifyInv queries computer algebra systems to find a set of program states S such that S includes states maximizing the absolute difference of two sides in Eq. (9):

$$S \ni \mathbf{argmax}_s |inv(s) - ([\neg G] \cdot \mathsf{postE} + [G] \cdot wp(P, inv))\,(s)|.$$

If there are no program state where the absolute difference is non-zero, verifyInv returns *inv* as a true invariant. Otherwise, the maximizing states in S are added to the list of counterexamples *cex*; if no candidate in *invs* is verified, verifyInv returns False and the accumulated list of counterexamples *cex*. The next iteration of the CEGIS loop will sample program traces starting from these counterexample initial states, hopefully leading to a learned model with less error.

6 Learning Sub-invariants

Next, we instantiate Exist for our second problem: learning sub-invariants. Given a program $\mathsf{geo} = \mathbf{while}\ G : P$ and a pair of pre- and post- expectations $(\mathsf{preE}, \mathsf{postE})$, we want to find a expectation I such that $\mathsf{preE} \leq I$, and

$$I \leq \Phi^{\mathsf{wpe}}_{\mathsf{postE}}(I) := [\neg G] \cdot \mathsf{postE} + [G] \cdot \mathsf{wpe}(P, I)$$

Intuitively, $\Phi^{\mathsf{wpe}}_{\mathsf{postE}}(I)$ computes the expected value of the expectation I after one iteration of the loop. We want to train a model M such that M translates to an expectation I whose expected value increases each iteration, and $\mathsf{preE} \leq I$.

The high-level plan is the same as for learning exact invariants: we train a model to minimize a loss defined to capture the sub-invariant requirements. We generate features $\mathcal{F}$ and sample initializations *states* as before. Then, from each $s \in states$, we repeatedly run just the loop body P and record the set of output states in *data*; this departs from our method for exact invariants, which repeatedly runs the entire loop to completion. Given this trace data, for any program state $s \in states$ and expectation I, we can compute the empirical mean of I's value after running the loop body P on state s. Thus, we can approximate $\mathsf{wpe}(P, I)(s)$ for $s \in states$ and use this estimate to approximate $\Phi^{\mathsf{wpe}}_{\mathsf{postE}}(I)(s)$. We then define a loss to sum up the violation of $I \leq \Phi^{\mathsf{wpe}}_{\mathsf{postE}}(I)$ and $\mathsf{preE} \leq I$ on state $s \in states$, estimated based on the collected data.

The main challenge for our approach is that existing model tree learning algorithms do not support our loss function. Roughly speaking, model tree learners typically assume a node's two child subtrees can be learned separately; this is the case when

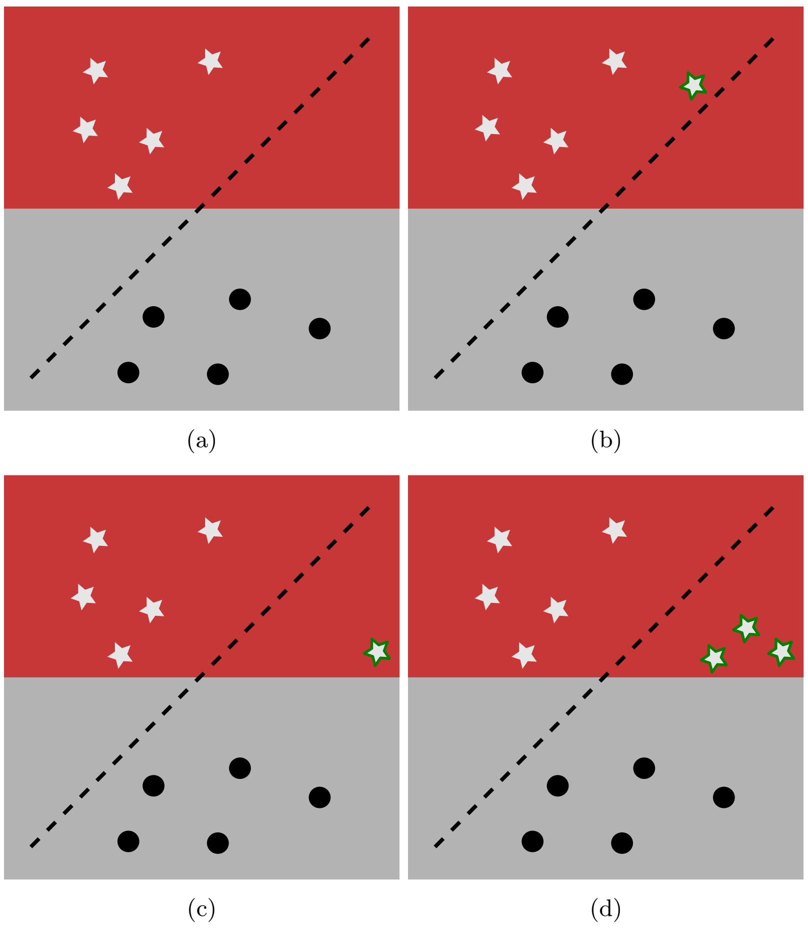


Fig. 6: Improved data augmentation

optimizing on the loss we used for exact invariants, but this is *not* the case for the loss for sub-invariants.

To solve this challenge, we first broaden the class of models to neural networks. To produce sub-invariants that can be verified, we still want to learn simple classes of models, such as piecewise functions of numerical expressions. Accordingly, we work with a class of neural architectures that can be translated into model trees, *neural model trees*, adapted from neural decision trees developed by Yang et al. [18]. We defer the technical details of neural model trees to Section 6.2.2; for now, we can treat them as differentiable approximations of standard model trees; since they are differentiable they can be learned with gradient descent, which can optimize on the sub-invariant loss function.

Outline.
We will discuss changes in $\mathsf{sampleTraces}$, $\mathsf{learnInv}$ and $\mathsf{verifyInv}$ for learning sub-invariants but omit descriptions of $\mathsf{getFeatures}$, $\mathsf{sampleStates}$, $\mathsf{extractInv}$ because Exist generates features, samples initial states and extracts expectations in the same way as in Section 5. To simplify the exposition, we will assume $\mathsf{getFeatures}$ generates the same set of features $\mathcal{F} = \mathcal{F}_l = \mathcal{F}_m$ for model trees with linear models and model trees with multiplication models.

6.1 Sample Training Data (sampleTraces)

Unlike when sampling data for learning exact invariants, here, $\mathsf{sampleTraces}$ runs only one iteration of the given program $\mathsf{geo} = \textbf{while } G : P$, that is, just P, instead of running the whole loop. Intuitively, this difference in data collection is because we aim to directly handle the sub-invariant condition, which encodes a single iteration of the loop. For exact invariants, our approach proceeded indirectly by learning the expected value of postE after running the loop to termination.

From any initialization $s_i \in states$ such that G holds on s_i, $\mathsf{sampleTraces}$ runs the loop body P for N_{runs} trials, each time restarting from s_i, and records the set of output states reached. If executing P from s_i leads to output states $\{s_{i1}, \dots, s_{iN_{runs}}\}$, then $\mathsf{sampleTraces}$ produces the training example:

$$(s_i, S_i) = (s_i, \{s_{i1}, \dots, s_{iN_{runs}}\}) ,$$

For initialization $s_i \in states$ such that G is false on s_i, $\mathsf{sampleTraces}$ simply produces $(s_i, S_i) = (s_i, \emptyset)$ since the loop body is not executed.

6.2 Learning a Neural Model Tree (learnInv)

Given the dataset $data = \{(s_1, S_1), \dots, (s_K, S_K)\}$ (with $K = N_{states}$), we want to learn an expectation I such that $\mathsf{preE} \leq I$ and $I \leq \Phi^{\mathsf{wpe}}_{\mathsf{postE}}(I)$. By case analysis on the guard G, the requirement $I \leq \Phi^{\mathsf{wpe}}_{\mathsf{postE}}(I)$ can be split into two constraints:

$$[G] \cdot I \leq [G] \cdot \mathsf{wpe}(P, I) \qquad \text{and} \qquad [\neg G] \cdot I \leq [\neg G] \cdot \mathsf{postE}.$$

If $I = \mathsf{postE} + [G] \cdot I'$, then the second requirement reduces to $[\neg G] \cdot postE \leq [\neg G] \cdot postE$ and is always satisfied. So to simplify the loss and training process, we again aim to learn an expectation I of the form of $\mathsf{postE} + [G] \cdot I'$. Thus, we want to train a model tree T such that T translates into an expectation I', and

$$\mathsf{preE} \leq \mathsf{postE} + [G] \cdot I' \tag{10}$$
$$[G] \cdot (\mathsf{postE} + [G] \cdot I') \leq [G] \cdot \mathsf{wpe}(P, \mathsf{postE} + [G] \cdot I') \tag{11}$$

Then, we define the loss of model tree T on $data$ to be

$$err(T, data) \coloneqq err_1(T, data) + err_2(T, data),$$

where $err_1(T, data)$ captures Eq. (10) and $err_2(T, data)$ captures Eq. (11).

Defining err_1 is relatively simple: we sum up the one-sided difference between $\mathsf{preE}(s)$ and $\mathsf{postE}(s) + G(s) \cdot T(\mathcal{F}(s))$ across $s \in states$, where T is the model tree getting trained and $\mathcal{F}(s)$ is the feature vector $\mathcal{F}$ evaluated on s. That is,

$$err_1(T, data) := \sum_{i=1}^{K} \max\left(0, \mathsf{preE}(s_i) - \mathsf{postE}(s_i) - G(s_i) \cdot T(\mathcal{F}(s_i))\right). \qquad (12)$$

Above, $\mathsf{preE}(s_i)$, $\mathsf{postE}(s_i)$, and $G(s_i)$ are the value of expectations preE, postE, and G evaluated on program state s_i.

The term err_2 is more involved. Similar to err_1, we aim to sum up the one-sided difference between two sides of Eq. (11) across state $s \in states$. On program state s that does not satisfy G, both sides are 0; for s that satisfies G, we want to evaluate $\mathsf{wpe}(P, \mathsf{postE} + [G] \cdot I')$ on s, but we do not have exact access to $\mathsf{wpe}(P, \mathsf{postE} + [G] \cdot I')$ and need to approximate its value on s based on sampled program traces. Recall that $\mathsf{wpe}(P, I)(s)$ is the expected value of I after running program P from s, and our dataset contains training examples (s_i, S_i) where S_i is a set of states reached after running P on an initial state s_i satisfying G. Thus, we can approximate $[G] \cdot \mathsf{wpe}(P, \mathsf{postE} + G \cdot I')(s_i)$ by

$$G(s_i) \cdot \frac{1}{|S_i|} \cdot \sum_{s \in S_i} \left(\mathsf{postE}(s) + G(s) \cdot I'(s)\right).$$

To avoid division by zero when s_i does not satisfy G and S_i is empty, we evaluate the expression in a short-circuit manner such that when $G(s_i) = 0$, the whole expression is immediately evaluated to zero.

Therefore, we define

$$\begin{aligned} err_2(T, data) = \sum_{i=1}^{K} \max \Bigg(& 0, G(s_i) \cdot \mathsf{postE}(s_i) + G(s_i) \cdot T(\mathcal{F}(s_i)) \\ & - G(s_i) \cdot \frac{1}{|S_i|} \cdot \sum_{s \in S_i} \left(\mathsf{postE}(s) + G(s) \cdot T(\mathcal{F}(s))\right) \Bigg). \end{aligned}$$

Standard model tree learning algorithms do not support this kind of loss function, and since our overall loss $err(T, data)$ is the sum of $err_1(T, data)$ and $err_2(T, data)$, we cannot use standard model tree learning algorithm to optimize $err(T, data)$ either. Fortunately, gradient descent does support this loss function. While gradient descent cannot directly learn model trees (See Section 6.2.1), we can use gradient descent to train a *neural* model tree T to minimize $err(T, data)$. The learned neural networks can be converted to model trees, and then converted to expectations as before (See Section 6.2.2).

6.2.1 Difficulty of Training with Standard Algorithms.

When calculating the error contributed by one training example in err_l, err_m or err_1, the training algorithm only needs to evaluate the model tree T on the feature vector of *one* program state: the data entry (s_i, v_i) in *data* contributes

$$(\mathsf{postE}(s_i) + G(s_i) \cdot T(\mathcal{F}_l(s_i)) - v_i)^2$$

to $err_l^2(T, data)$ and similarly $(\mathsf{postE}(s_i) + G(s_i) \cdot T(\mathcal{F}_m(s_i)) - v_i)^2$ to $err_m^2(T, data)$; the data entry (s_i, S_i) contributes

$$\max(0, \mathsf{preE}(s_i) - \mathsf{postE}(s_i) - G(s_i) \cdot T(\mathcal{F}(s_i)))$$

to $err_1^2(T, data)$. For $f = \mathcal{F}_l(s_i), \mathcal{F}_m(s_i)$ or $\mathcal{F}(s_i)$, when we calculate $T(f)$ for a single f as in above expressions, either T's root is a leaf and we simply apply its leaf model to the f, or we pass f to exactly *one* children subtree T' of T and recursively calculate $T'(f)$. We associate a training example (s_i, v_i) (or (s_i, S_i)) to a subtree T' if the feature vector f of s_i gets passed to T'. Thus, different subtrees get associated with disjoint sets of training examples, and the standard training algorithm for model trees is able to adopt the divide-and-conquer strategy and optimize children of T independently.

However, the err_2 error of a model tree T on a training example (s_i, S_i) is

$$\max\left(0, G(s_i) \cdot \mathsf{postE}(s_i) + G(s_i) \cdot T(\mathcal{F}(s_i)) - G(s_i) \cdot \frac{1}{|S_i|} \cdot \sum_{s \in S_i} \big(\mathsf{postE}(s) + G(s) \cdot T(\mathcal{F}(s))\big)\right),$$

which depends on the evaluated values of $T(\mathcal{F}(s_i))$ and $T(\mathcal{F}(s))$ for all $s \in S_i$. Evaluating $T(\mathcal{F}(s_i))$ and all $T(\mathcal{F}(s))$ can use multiple children subtrees of T. Thus, we cannot associate one training example (s_i, S_i) to exactly one children of T and optimize children of T independently. While children of T still gets associated with disjoint sets of feature vectors, we cannot train children of T to minimize err_2 just with sets of program states s_i because unlike when learning exact invariants, now we do not know what $T(\mathcal{F}(s_i))$ should be without calculating $\sum_{s \in S_i} \big(\mathsf{postE}(s) + G(s) \cdot T(\mathcal{F}(s))\big)$. Furthermore, because of the use of $\max(0, -)$ function in err_2, we cannot solve the problem by rearranging the terms across training examples.

6.2.2 Constructions of Neural Model Trees

To address the problem of optimizing on err_2, we consider another general training algorithm, *gradient descent*, i.e., iteratively taking the gradient of the loss with respect to the trainable parameters and adjust the parameters along the gradient to minimize the loss. Although gradient descent only provides theoretical guarantee of finding global minimum when the loss is convex, gradient descent and its stochastic variant have showed good performance across a wide range of problems, as demonstrated by the recent success of neural networks. To apply gradient descent, however, we need the training model to be differentiable with respect to trainable parameters, and model trees and decision trees are not differentiable with respect to each predicate in the internal node. To address this problem, we use a differentiable approximation of model trees based on neural networks, which we call *neural model trees*, and train them using standard gradient descent method. we start with a model called *neural decision tree* developed by Yang et al. [18]. As in standard decision tree learning, they consider internal nodes predicates and leaf labels as trainable parameters. Unfortunately, if we let the training parameter at an internal node range over predicates of the form $f \leq c$, it is unclear how to develop a differentiable loss function to optimize. Thus, deviating from standard decision tree, neural decision trees assume there is a predicate of the form $f_i \leq c_i$ for each feature f_i that we can split on and regard the cut point c_i as the training parameter, and in addition, they use a smooth approximation of the predicate $f_i \leq c_i$ so the loss becomes differentiable.

Concretely, a neural decision tree is a function

$$\mathsf{nndt} : \text{Trainable parameters } \Theta \times \text{Data} \to \text{Labels}$$

where the trainable parameters includes cut points c_i for each feature $f_i \in \mathcal{F}$ and numerical labels. The neural architecture developed by Yang et al. [18] implements the map nndt in two stages: $\mathsf{nndt} = \mathsf{nn}_{\text{label}} \circ \mathsf{nn}_{\text{classify}}$, where

$$\mathsf{nn}_{\text{label}} : \text{Trainable parameters } \Theta \times \text{Data} \to \text{OneHot}(|2^{\mathcal{F}}|)$$

takes input $(\{c_i\}, d)$ to a one-hot vector of length $2^{\mathcal{F}}$ where the hot bit represents the leaf that the data example d gets classified into according to the smoothed version of predicates $f_i \leq c_i$, and

$$\mathsf{nn}_{\text{classify}} : \text{OneHot}(|2^{\mathcal{F}}|) \to \text{Labels}$$

assigns a label to each leaf.

To approximate model trees with differentiable neural networks, we define

$$\mathsf{nnmt} : \text{Trainable parameters } \Theta \times \text{Data} \to \mathbb{R}$$

by having $\mathsf{nnmt}(\theta, d) = \mathsf{nn}_{\text{regress}}(\mathsf{nn}_{\text{classify}}(\theta, d))(d)$ where

$$\mathsf{nn}_{\text{regress}} : \text{OneHot}(|2^{\mathcal{F}}|) \to \text{Leaf Models}$$

associates each one-hot vector with a trainable leaf model.

Our tool assumes the leaf models are linear models or multiplication models. When the leaf models are linear, each leaf model can be represented by a vector of linear coefficients, and we can represent $\mathsf{nn}_{\text{regress}}$ as a $|2^{\mathcal{F}}| \times |\mathcal{F}|$ matrix. When we want to fit neural model trees with multiplication leaf models, we take the logarithm of data passed to the neural model, train a model with linear leaf models, and exponentiate the output. In both cases, nnmt is differentiable with respect to its trainable parameters, so we can apply standard stochastic gradient descent to train it.

6.3 Verify Extracted Expectations (verifyInv)

The verifier verifyInv is very similar to the one in Section 5 except here it solves a different optimization problem. For each candidate *inv* in the given list *invs*, it looks for a set S of program states such that S includes

$$\mathbf{argmax}_s \mathsf{preE}(s) - inv(s) \qquad \text{and} \qquad \mathbf{argmax}_s G(s) \cdot I(s) - [G] \cdot \mathsf{wpe}(P, I)(s).$$

As in our approach for exact invariant learning, the verifier aims to find counterexample states s that violate at least one of these constraints by as large of a margin as possible; these high-quality counterexamples guide data collection in the following iteration of the CEGIS loop. Concretely, the verifier accepts inv if it cannot find any program state s where $\mathsf{preE}(s) - inv(s)$ or $G(s) \cdot I(s) - [G] \cdot \mathsf{wpe}(P, I)(s)$ is positive. Otherwise, it adds all states $s \in S$ with strictly positive margin to the set of counterexamples cex.

7 Evaluation

We implemented our prototype in Python, using sklearn and tensorflow to fit model trees and neural model trees, and Wolfram Alpha to verify and perform counterexample generation. We have evaluated our tool on a set of 18 benchmarks drawn from different sources in prior work [13, 14, 26]. All our benchmarks are almost surely terminating: for all benchmarks except Gambler, their almost sure terminations are witnessed by simple ranking super-martingales that are linear on variables and tests on variables, e.g., $[x \geq 0]$; the gambler's ruin problem is also well-studied to be AST [19, 27]. While we check this condition by hand, existing work has explored synthesis of such ranking super-martingale (e.g., [19–21, 28, 29]).

Our experiments were designed to address the following research questions:

R1. Can Exist synthesize exact invariants for a variety of programs?

R2. Can Exist synthesize sub-invariants for a variety of programs?

We summarize our findings as follows:

- Exist successfully synthesized and verified exact invariants for 14/18 benchmarks within a timeout of 300 seconds. Our tool was able to generate these 14 invariants in reasonable time, taking between 1 to 237 seconds. The sampling phase dominates the time in most cases. We also compare Exist with a tool from prior literature, Mora [30]. We found that Mora can only handle a restrictive set of programs and cannot handle many of our benchmarks. We also discuss how our work compares with a few others in (Section 8).
- To evaluate sub-invariant learning, we created multiple problem instances for each benchmark by supplying different pre-expectations. On a total of 34 such problem instances, Exist was able to infer correct invariants in 27 cases, taking between 7 to 102 seconds.

We present in the extended version the tables of complete experimental results. Because the training data we collect are inherently stochastic, the results produced by our tool are not deterministic.[1] As expected, sometimes different trials on the same benchmarks generate different sub-invariants; while the exact invariant for each benchmark is unique, Exist may also generate semantically equivalent but syntactically different expectations in different trials (e.g. it happens for BiasDir).

[1]The code and data sampled in the trial that produced the tables in this paper can be found at https://github.com/JialuJialu/Exist.

Table 1: Exact Invariants generated by EXIST

Name	postE	Learned Invariant	ST	LT	VT	TT
BiasDir	x	$x + [x == y] \cdot (-0.2 \cdot x^2 - 0.2 \cdot y^2$ $-0.2 \cdot x \cdot y - 0.2 \cdot x - 0.2 \cdot y + 0.5)$	23.97	0.34	0.25	24.56
Bin0	x	$x + [n > 0] \cdot ([y \neq 0] \cdot p \cdot n \cdot y)$	72.80	5.09	1.15	79.04
Bin1	n	$x + [n < M] \cdot (M \cdot p - n \cdot p)$	25.67	12.03	0.22	37.91
Bin2	x	$x + [n > 0] \cdot (p \cdot n \cdot n$ $-p \cdot n \cdot y + n \cdot y)$	84.64	26.54	0.48	111.66
DepRV	$x \cdot y$	-	-	-	-	-
Detm	$count$	$count + [x \leq 10] \cdot (11 - x)$	0.09	0.72	0.06	0.87
Duel	t	-	-	-	-	-
Fair	$count$	$count + [c1 + c2 == 0] \cdot$ $(p1 + p2)/(p1 + p2 - p1 \cdot p2)$	5.78	1.62	0.30	7.69
Gambler	z	$z + [x > 0 \text{ and } y > x] \cdot$ $x \cdot (y - x)$	112.02	3.52	9.97	125.51
Geo0	z	$z + [flip == 0] \cdot (1 - p_1)/p_1$	12.01	0.85	2.65	15.51
Geo1	z	$z + [flip == 0] \cdot (1 - p_1)/p_1$	20.30	5.20	3.57	29.09
Geo2	z	$z + [flip == 0] \cdot (1 - p_1)/p_1$	10.78	2.17	0.12	13.07
GeoAr	x	-	-	-	-	-
LinExp	z	-	-	-	-	-
Mart	$rounds$	$rounds + [b > 0] \cdot (1/p)$	24.10	3.83	0.05	27.98
Prinsys	$[x == 1]$	$[x == 1] + [x == 0] \cdot (1 - p_2)$	1.60	0.17	1.25	3.02
RevBin	z	$z + [x > 0] \cdot (x/p)$	234.64	3.13	0.14	237.92
Sum0	x	$x + [n > 0] \cdot (0.5 \cdot p \cdot n^2 +$ $0.5 \cdot p \cdot n)$	102.12	34.61	26.74	163.48

Implementation Details.

For input parameters to EXIST, we use $N_{runs} = 500$ and $N_{states} = 500$. Besides input parameters listed in Fig. 4, we allow the user to supply a list of features as an optional input. In feature generation, getFeatures enumerates expressions made up by program variables and user-supplied features according to a grammar. Also, when incorporating counterexamples cex, we make 30 copies of each counterexample to give them more weights in the training. All experiments were conducted on a MacBook Pro 2020 with M1 chip running macOS Monterey Version 12.1.

7.1 R1: Evaluation of the Exact Invariant Method

Efficacy of Invariant Inference.

EXIST was able to infer exact invariants in 14/18 benchmarks. Out of 14 successfully inferred benchmarks, only 2 of them need user-supplied features ($n \cdot p$ for Bin2 and Sum0). Table 1 shows the postexpectation (postE), the inferred invariant (Invariant), sampling time (ST), learning time (LT), verification time (VT) and the total time (TT) for a few benchmarks. For generating exact invariants, the running time of EXIST is dominated by the sampling time. However, this phase can be parallelized easily.

We (manually) check that all the inferred invariants only evaluates to non-negative values if all the program variables take non-negative values, which is required for them to be expectations as defined in Definition 3 and to apply Proposition 2. We then (manually) check whether the inferred invariants are provably the weakest preexpectations according to Proposition 2. We find 13 out of the 14 exact invariants inferred by EXIST satisfy at least one condition, and thus, are provably the weakest preexpectation: the invariants inferred for Bin0, Bin1, Bin2 and Sum0 satisfy condition (a), and the subinvariants inferred for the rest of the benchmarks except Gambler all satisfy the condition (b). The invariant EXIST inferred for Gambler, $z + [x > 0 \text{ and } y > x] \cdot x \cdot (y - x)$, does not satisfy any of the conditions, but it is sound according to known results about random walks: the postexpectation z increases 1 at each iteration, and recurrence analysis shows that $x \cdot (y - x)$ is the expected number of iterations Gambler [27].

Failure Analysis.

EXIST failed to generate invariants for 4/18 benchmarks. For two of them, EXIST was able to generate expectations that are very close to an invariant (DepRV and LinExp); for the third failing benchmarks (Duel), the ground truth invariant is very complicated. For LinExp, while a correct invariant is $z + [n > 0] \cdot 2.625 \cdot n$, EXIST generates expectations like $z + [n > 0] \cdot (2.63 \cdot n - 0.02)$ as candidates. For DepRV, a correct invariant is $x \cdot y + [n > 0] \cdot (0.25 \cdot n^2 + 0.5 \cdot n \cdot x + 0.5 \cdot n \cdot y - 0.25 \cdot n)$, and in our experiment EXIST generates $0.25 \cdot n^2 + 0.5 \cdot n \cdot x + 0.5 \cdot n \cdot y - 0.27 \cdot n - 0.01 \cdot x + 0.12$. In both cases, the ground truth invariants use coefficients with several digits, and since learning from data is inherently stochastic, EXIST cannot generate them consistently. In our experiments, we observe that our CEGIS loop does guide the learner to move closer to the correct invariant in general, but sometimes progress obtained in multiple iterations can be offset by noise in one iteration. For GeoAr, we observe the verifier incorrectly accepted the complicated candidate invariants generated by the learner because Wolfram Alpha was not able to find valid counterexamples for our queries.

Comparison with Previous Work.

There are few existing tools that can automatically compute expected values after probabilistic loops. We experimented with one such tool, called MORA [30]. We managed to encode our benchmarks Geo0, Bin0, Bin2, Geo1, GeoAr, and Mart in their syntax. Among them, MORA fails to infer an invariant for Geo1, GeoAr, and Mart. We also tried to encode our benchmarks Fair, Gambler, Bin1, and RevBin but found MORA's syntax was too restrictive to encode them. Table 2 shows how we encoded two of our benchmarks into MORA.

Table 2: Encoding of GeoAr and Mart in Mora Syntax

Program	Encoding

bool z, int x,y, float p
while (z ≠ 0) do
y ← y + 1;
c $\xleftarrow{\$}$ Bernoulli(p)
if c then z ← 0
else x ← x + y
	int z = 1, x = 0, y = 0
	while true:
	y ← y + z;
	z ← 0 @ p;z
	x ← x + y * z

int c, b, rounds, float p
while (b > 0) do
d $\xleftarrow{\$}$ Bernoulli(p)
if d then
c ← c + b;
b ← 0;
else
c ← c - b;
b ← 2 * b;
rounds ← rounds + 1;
	int c = 0
	b = 1
	d = 0
	rounds = 1
	while true:
	d ← 1 @ p;d
	c ← c + b*(d-1)+b*d
	b ← 2*b*(1-d)
	rounds ← rounds + 1 - d

7.2 R2: Evaluation of the Sub-invariant Method

Efficacy of invariant inference.

Exist is able to synthesize sub-invariants for 27/34 benchmarks. Two out of 27 successfully inferred benchmarks use user-supplied features – Gambler with pre-expectation $x \cdot (y - x)$ uses $(y - x)$, and Sum0 with pre-expectation $x + [x > 0] \cdot (p \cdot n/2)$ uses $p \cdot n$. Contrary to the case for exact invariants, the learning time dominates. This is not surprising: the sampling time is shorter because we only run one iteration of the loop, but the learning time is longer as we are optimizing a more complicated loss function.

We check whether the subinvariants synthesized by Exist satisfy one of the conditions (a), (b) or (c) in Proposition 2, and thus, provably lower bounds the weakest preexpectation. There are two subinvariants that do not satisfy any of the conditions: Gambler with inferred subinvariant $z + [x > 0 \text{ and } y > x] \cdot x \cdot (y - x)$ and GeoAr with the inferred invariants x; both of them do lower bound the expectations calculated manually through Theorem 1. For the rest of 25 subinvariants inferred by Exist, the subinvariants inferred by Exist for Bin0, Bin1, Bin2, LinExp, DepRV and Sum0 satisfy condition (a), and the subinvariants inferred for the rest of the benchmarks satisfy the condition (b). As before, Table 3 reports the details for all these benchmarks.

One interesting thing that we found when gathering benchmarks is that for many loops, pre-expectations used by prior work or natural choices of pre-expectations are themselves sub-invariants. Thus, for some instances, the sub-invariants generated by Exist is the same as the pre-expectation preE given to it as input. However, Exist is not checking whether the given preE is a sub-invariant: the learner in Exist does not know about preE besides the value of preE evaluated on program states. Also, we also designed benchmarks where pre-expectations are *not* sub-invariants (BiasDir with preE $= [x \neq y] \cdot x$, DepRV with preE $= x \cdot y + [n > 0] \cdot 1/4 \cdot n^2$, Gambler with preE $= x \cdot (y - x)$, Geo0 with preE $= [flip == 0] \cdot (1 - p1)$), and Exist is able to generate sub-invariants for 3/4 such benchmarks.

Table 3: Sub-Invariants generated by Exist

Name	postE	preE	Learned Invariant	ST	LT	VT	TT
BiasDir	x	$[x \neq y] \cdot x$	$x + [x == y] \cdot (0.1 \cdot p - 0.5 \cdot x - 0.5 \cdot y + 0.1)$	12.37	21.36	0.75	34.48
		$[x == y] \cdot 1/2$	$x + [x == y] \cdot (-0.5 \cdot x - 0.5 \cdot y + 0.5)$	12.33	27.12	0.10	39.55
Bin0	x	$x + [n > 0] \cdot (p \cdot n \cdot y)$	-	-	-	-	-
		x	x	16.69	21.47	0.44	38.59
Bin1	n	$x + [n < M] \cdot (p \cdot M - p \cdot n)$	-	-	-	-	-
		n	x	8.55	17.84	0.10	26.52
Bin2	x	$x + [n > 0] \cdot (1 - p) \cdot n \cdot y$	-	-	-	-	-
		x	x	16.71	21.64	0.78	39.13
DepRV	$x \cdot y$	$x \cdot y + [n > 0] \cdot (1/4 \cdot n \cdot n)$	-	-	-	-	-
		$x \cdot y$	$x \cdot y$	16.08	17.04	0.17	33.28
Detm	$count$	$count + [x <= 10] \cdot 1$	$count + [x <= 10] \cdot 1$	3.95	17.67	0.03	21.66
		$count$	$count$	5.99	10.31	0.04	16.35
Duel	t	$c \cdot (-p_1 + p_2 - p_1 \cdot p_2) + 1$	-	-	-	-	-
Fair	$count$	$count + [c_1 + c_2 == 0] \cdot (p_1 + p_2)$	$[c_1 + c_2 == 0] \cdot (p_1 + p_2) + count$	8.74	25.84	0.27	34.85
		$count$	$count$	6.71	11.73	0.40	18.85
Mart	$rounds$	$rounds + [b > 0] \cdot 1$	$rounds + [b > 0] \cdot 1$	17.68	31.49	0.11	49.27
		$rounds$	$rounds$	16.61	21.32	0.16	38.09

Failure Analysis.

On program instances where Exist fails to generate a sub-invariant, we observe two common causes. First, gradient descent seems to get stuck in local minima because the learner returns suboptimal models with relatively low loss. The loss we are training on is very complicated and likely to be highly non-convex, so this is not surprising. Second, we observed inconsistent behavior due to noise in data collection and learning.

Table 4: Table 3 Continued

Name	postE	preE	Learned Invariant	ST	LT	VT	TT
Gambler	z	z	z	6.99	12.56	0.43	19.98
		$x \cdot (y - x)$	$z+ [x > 0 \;\&\; y > x] \cdot x \cdot (y - x)$	7.31	28.87	8.29	44.46
Geo0	z	$z+ [flip == 0] \cdot (1 - p_1)$	$z+ [flip == 0] \cdot (1 - p_1)$	8.69	28.04	0.10	36.84
		z	z	8.08	12.01	3.62	23.71
		$[flip == 0] \cdot (1 - p_1))$	$z+ [flip == 0] \cdot (1 - p_1)$	8.70	26.13	0.19	35.02
Geo1	z	z	z	8.80	13.66	0.03	22.48
Geo2	z	z	z	8.19	14.49	0.05	22.73
GeoAr	x	$x + [z! = 0] \cdot y \cdot (1 - p)/p$	-	-	-	-	-
		x	x	8.51	40.98	0.39	49.89
LinExp	z	$z + [n > 0] \cdot 2$	$[n > 0] \cdot (n + 1)$	53.72	30.01	0.35	84.98
		$z + [n > 0] \cdot 2 \cdot n$	$z + [n > 0] \cdot 2 \cdot n$	29.18	28.61	0.68	58.48
Prinsys	$[x == 1] \cdot 1$	$[x == 1] \cdot 1$	$[x == 1]$	1.10	5.85	0.33	7.29
RevBin	z	$z + [x > 0] \cdot x$	$z + [x > 0] \cdot x/p$	18.17	71.15	2.17	91.55
		z	z	15.62	18.74	0.06	34.42
Sum0	x	$x + [n > 0] \cdot (p \cdot n \cdot n/2)$	-	-	-	-	-
		$x + [n > 0] \cdot (p \cdot n/2)$	$x + [n > 0] \cdot (p \cdot n)$	19.60	76.71	5.94	102.29

For instance, for GeoAr with preE $= x + [z \neq 0] \cdot y \cdot (1 - p)/p$, EXIST could sometimes find a sub-invariant with supplied feature $(1 - p)$, but we could not achieve this result consistently.

Comparison with Learning Exact Invariants.

The performance of EXIST on learning sub-invariants is less sensitive to the complexity of the ground truth invariants. For example, EXIST is not able to generate an exact invariant for LinExp as its exact invariant is complicated, but EXIST is able to generate sub-invariants for LinExp. However, we also observe that when learning sub-invariants, Exist returns complicated expectations with high loss more often.

8 Related Work

Invariant Generation for Probabilistic Programs.

There has been a steady line of work on probabilistic invariant generation over the last few years. The Prinsys system [13] employs a template-based approach to guide the search for probabilistic invariants. Prinsys is able encode invariants with guard expressions, but the system doesn't produce invariants directly—instead, Prinsys produces logical formulas encoding the invariant conditions, which must be solved manually.

Chen et al. [14] proposed a counterexample-guided approach to find polynomial invariants, by applying Lagrange interpolation. However, invariants involving guard expressions—common in our examples—cannot be found, since they are not polynomials. Additionally, Chen et al. [14] uses a weaker notion of invariant, which only needs to be correct on certain initial states; our tool generates invariants that are correct on all initial states. Feng et al. [31] improves on Chen et al. [14] by using *Stengle's Positivstellensatz* to encode invariants constraints as a semidefinite programming problem. Their method can find polynomial sub-invariants that are correct on all initial states. However, their approach cannot synthesize piecewise linear invariants, and their implementation has additional limitations and could not be run on our benchmarks.

Subsequent to the original publication of our results, Batz et al. [32] proposed a different method to synthesize invariant expectations. Their approach is based on a rich class of templates with numerical-valued holes, and uses an efficient CEGIS loop to improve the invariant expectations. Unlike our approach, Batz et al. [32] use access to the program source code. In this way, they are able to use a verifier to check correctness of invariants and find counterexamples within their CEGIS loop. In contrast, our approach does not rely on direct verification during synthesis since our method does not have access to the internals of the program.

There is also a line of work on abstract interpretation for analyzing probabilistic programs; Chakarov and Sankaranarayanan [33] search for linear expectation invariants using a "pre-expectation closed cone domain", while recent work by Wang et al. [34] employs a sophisticated algebraic program analysis approach.

Another line of work applies *martingales* to derive insights of probabilistic programs. Chakarov and Sankaranarayanan [35] showed several applications of martingales in program analysis, and Barthe et al. [36] gave a procedure to generate candidate martingales for a probabilistic program; however, this tool gives no control over which expected value is analyzed—the user can only guess initial expressions and the tool generates valid bounds, which may not be interesting. Our tool allows the user to pick which expected value they want to bound.

Another line of work for automated reasoning uses *moment-based analysis.* Bartocci et al. [30, 37] develop the Mora tool, which can find the moments of variables as functions of the iteration for loops that run forever by using ideas from computational algebraic geometry and dynamical systems. This method is highly efficient and is guaranteed to compute moments exactly. However, there are two limitations. First, the moments can give useful insights about the distribution of variables' values after each iteration, but they are fundamentally different from our notion of invariants which allow us to compute the expected value of any given expression *after termination* of a loop. Second, there are important restrictions on the probabilistic programs. For instance, conditional statements are not allowed and the use of symbolic inputs is limited. As a result, most of our benchmarks cannot be handled by Mora.

In a similar vein, Kura et al. [38], Wang et al. [39] bound higher *central moments* for running time and other monotonically increasing quantities. Like our work, these works consider probabilistic loops that terminate. However, unlike our work, they are limited to programs with constant size increments.

Data-driven Invariant Synthesis.

We are not aware of other data-driven methods for learning probabilistic invariants, but a recent work Abate et al. [40] proves probabilistic termination by learning ranking supermartingales from trace data. Our method for learning sub-invariants (Section 6) can be seen as a natural generalization of their approach. However, there are also important differences. First, we are able to learn general sub-invariants, not just ranking supermatingales for proving termination. Second, our approach aims to learn model trees, which lead to simpler and more interpretable sub-invariants. In contrast, Abate, et al. [40] learn ranking functions encoded as two-layer neural networks.

Data-driven inference of invariants for deterministic programs has drawn a lot of attention, starting from Daikon [16]. ICE learning with decision trees [41] modifies the decision tree learning algorithm to capture *implication counterexamples* to handle inductiveness. Hanoi [42] uses counterexample-based inductive synthesis (CEGIS) [43] to build a data-driven invariant inference engine that alternates between weakening and strengthening candidates for synthesis. Recent work uses neural networks to learn invariants [44]. These systems perform classification, while our work uses regression. Data from fuzzing has been used for *almost correct* inductive invariants for programs with closed-box operations [45].

Probabilistic Reasoning with Pre-expectations.

Following Morgan and McIver, there are now pre-expectation calculi for domain-specific properties, like expected runtime [23] and probabilistic sensitivity [46]. All of these systems define the pre-expectation for loops as a least fixed-point, and practical reasoning about loops requires finding an invariant of some kind.

9 Conclusion

Inspired by data-driven invariant generation techniques for standard programs, we present the first data-driven invariant generation algorithm for probabilistic program. Our method is the first to be able to learn exact piecewise linear probabilistic invariants fully automatically, without relying on templates or manually solving logical formulas.

Going forward, one potential direction is to improve sampling performance. While our tool finds invariants in a reasonable amount of time, it needs many input-output traces in order to reliably find invariants. Methods from statistics, like boosting, might be useful to increase stability of our learning approach. More broadly, a natural question is whether other quantitative invariants could be learned through regression, rather than classification. In general, our work further strengthens the research direction invested in exploring the synergy between machine learning and formal methods is solving hard tasks like program verification.

Acknowledgements

This work is in part supported by National Science Foundation grant #1943130 and #2152831. We also thank Google Research for supporting our research. We thank Ugo Dal Lago, Işil Dillig, IIT Kanpur PRAISE group, Cornell PL Discussion Group, and all reviewers for helpful feedback. We also thank Anmol Gupta in IITK for building a prototype verifier using Mathematica.

Data availability statement

The source code of the implementation and data is publicly available at:
https://github.com/JialuJialu/Exist

References

[1] Kozen, D.: Semantics of probabilistic programs **22**(3) (1981) https://doi.org/10.1016/0022-0000(81)90036-2

[2] Smith, C., Hsu, J., Albarghouthi, A.: Trace abstraction modulo probability. In: POPL (2019). https://doi.org/10.1145/3290352

[3] Albarghouthi, A., Hsu, J.: Synthesizing coupling proofs of differential privacy. In: POPL (2018). https://doi.org/10.1145/3158146

[4] Carbin, M., Misailovic, S., Rinard, M.C.: Verifying quantitative reliability for programs that execute on unreliable hardware. In: OOPSLA (2013). https://doi.org/10.1145/2509136.2509546

[5] Roy, S., Hsu, J., Albarghouthi, A.: Learning differentially private mechanisms. In: SP (2021). https://doi.org/10.1109/SP40001.2021.00060

[6] Baier, C., Clarke, E.M., Hartonas-Garmhausen, V., Kwiatkowska, M.Z., Ryan, M.: Symbolic model checking for probabilistic processes. In: ICALP (1997)

[7] Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV (2011). https://doi.org/10.1007/978-3-642-22110-1_47

[8] Dehnert, C., Junges, S., Katoen, J., Volk, M.: A storm is coming: A modern probabilistic model checker. In: CAV (2017). https://doi.org/10.1007/978-3-319-63390-9_31

[9] Kozen, D.: A probabilistic PDL **30**(2) (1985) https://doi.org/10.1016/0022-0000(85)90012-1

[10] Morgan, C., McIver, A., Seidel, K.: Probabilistic predicate transformers. TOPLAS (1996) https://doi.org/10.1145/229542.229547

[11] McIver, A., Morgan, C.: Abstraction, Refinement, and Proof for Probabilistic Systems, (2005). https://doi.org/10.1007/b138392

[12] Dijkstra, E.W.: Guarded commands, non-determinancy and a calculus for the derivation of programs. In: Language Hierarchies and Interfaces (1975). https://doi.org/10.1007/3-540-07994-7_51

[13] Gretz, F., Katoen, J., McIver, A.: Prinsys - on a quest for probabilistic loop invariants. In: QEST (2013). https://doi.org/10.1007/978-3-642-40196-1_17

[14] Chen, Y., Hong, C., Wang, B., Zhang, L.: Counterexample-guided polynomial loop invariant generation by Lagrange interpolation. In: CAV (2015). https://doi.org/10.1007/978-3-319-21690-4_44

[15] Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for esc/java. In: FME (2001). https://doi.org/10.1007/3-540-45251-6_29

[16] Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. (2007) https://doi.org/10.1016/j.scico.2007.01.015

[17] Quinlan, J.R.: Learning with continuous classes. In: AJCAI, vol. 92 (1992)

[18] Yang, Y., Morillo, I.G., Hospedales, T.M.: Deep neural decision trees. CoRR (2018) 1806.06988

[19] Chatterjee, K., Fu, H., Goharshady, A.K.: Termination analysis of probabilistic programs through Positivstellensatz's. In: CAV (2016)

[20] Chatterjee, K., Fu, H., Novotný, P., Hasheminezhad, R.: Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In: POPL (2016). https://doi.org/10.1145/2837614.2837639

[21] McIver, A., Morgan, C., Kaminski, B.L., Katoen, J.: A new proof rule for almost-sure termination. In: POPL (2018). https://doi.org/10.1145/3158121

[22] Batz, K., Kaminski, B.L., Katoen, J., Matheja, C.: Relatively complete verification of probabilistic programs: an expressive language for expectation-based reasoning. In: POPL (2021). https://doi.org/10.1145/3434320

[23] Kaminski, B.L., Katoen, J., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected run-times of probabilistic programs. In: ESOP (2016). https://doi.org/10.1007/978-3-662-49498-1_15

[24] Hark, M., Kaminski, B.L., Giesl, J., Katoen, J.: Aiming low is harder: induction for lower bounds in probabilistic program verification. (2020). https://doi.org/10.1145/3371105

[25] Park, D.: Fixpoint induction and proofs of program properties. Machine intelligence **5** (1969)

[26] Kaminski, B.L., Katoen, J.-P.: A weakest pre-expectation semantics for mixed-sign expectations. In: LICS (2017). https://doi.org/10.5555/3329995.3330088

[27] Leighton, T., Rubinfeld, R.: Random Walks – Lecture notes in Mathematics for Computer Science. MIT CS 6.042/18.062 (2006). https://web.mit.edu/neboat/Public/6.042/randomwalks.pdf

[28] Huang, M., Fu, H., Chatterjee, K., Goharshady, A.K.: Modular verification for almost-sure termination of probabilistic programs. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–29 (2019)

[29] Majumdar, R., Sathiyanarayana, V.: Sound and complete proof rules for probabilistic termination. arXiv preprint arXiv:2404.19724 (2024)

[30] Bartocci, E., Kovács, L., Stankovič, M.: Mora-automatic generation of moment-based invariants. In: TACAS (2020). https://doi.org/10.1007/978-3-030-45190-5_28

[31] Feng, Y., Zhang, L., Jansen, D.N., Zhan, N., Xia, B.: Finding polynomial loop invariants for probabilistic programs. In: ATVA (2017)

[32] Batz, K., Chen, M., Junges, S., Kaminski, B.L., Katoen, J.-P., Matheja, C.: Probabilistic program verification via inductive synthesis of inductive invariants. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 410–429 (2023). Springer

[33] Chakarov, A., Sankaranarayanan, S.: Expectation invariants for probabilistic program loops as fixed points. In: SAS (2014). https://doi.org/10.1007/978-3-319-10936-7_6

[34] Wang, D., Hoffmann, J., Reps, T.W.: PMAF: an algebraic framework for static analysis of probabilistic programs. In: PLDI (2018). https://doi.org/10.1145/3192366.3192408

[35] Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: CAV (2013). https://doi.org/10.1007/978-3-642-39799-8_34

[36] Barthe, G., Espitau, T., Ferrer Fioriti, L.M., Hsu, J.: Synthesizing probabilistic invariants via Doob's decomposition. In: CAV (2016). https://doi.org/10.1007/978-3-319-41528-4_3

[37] Bartocci, E., Kovács, L., Stankovič, M.: Automatic generation of moment-based invariants for prob-solvable loops. In: ATVA (2019). https://doi.org/10.1007/978-3-030-31784-3_15

[38] Kura, S., Urabe, N., Hasuo, I.: Tail probabilities for randomized program runtimes via martingales for higher moments. In: TACAS (2019). https://doi.org/10.1007/978-3-030-17465-1_8

[39] Wang, D., Hoffmann, J., Reps, T.: Central moment analysis for cost accumulators in probabilistic programs. In: PLDI (2021). https://doi.org/10.1145/3453483.3454062

[40] Abate, A., Giacobbe, M., Roy, D.: Learning probabilistic termination proofs. In: CAV (2021). https://doi.org/10.1007/978-3-030-81688-9_1

[41] Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: POPL (2016). https://doi.org/

10.1145/2914770.2837664

[42] Miltner, A., Padhi, S., Millstein, T., Walker, D.: Data-driven inference of representation invariants. In: PLDI 20 (2020). https://doi.org/10.1145/3385412.3385967

[43] Solar-Lezama, A.: Program sketching. Int. J. Softw. Tools Technol. Transf. (2013) https://doi.org/10.1007/s10009-012-0249-7

[44] Si, X., Dai, H., Raghothaman, M., Naik, M., Song, L.: Learning loop invariants for program verification. In: NeurIPS (2018). https://doi.org/10.5555/3327757.3327873

[45] Lahiri, S., Roy, S.: Almost correct invariants: Synthesizing inductive invariants by fuzzing proofs. In: ISSTA (2022)

[46] Aguirre, A., Barthe, G., Hsu, J., Kaminski, B.L., Katoen, J.-P., Matheja, C.: A pre-expectation calculus for probabilistic sensitivity. In: POPL (2021). https://doi.org/10.1145/3434333