Data-Driven Invariant Learning for Probabilistic Programs

No Author Given

No Institute Given

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, and also works with black-box access to the program. 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—standard imperative programs augmented with a sampling command—are a common way to express randomized computations. While the mathematical semantics of such programs is fairly well-understood [25], verification methods remain an active area of research. Existing automated techniques are either limited to specific properties (e.g., [3, 9, 35]), or target simpler computational models [4, 15, 28].

Reasoning about Expectations. One of the earliest methods for reasoning about probabilistic programs is through *expectations*. Originally proposed by Kozen [26], 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* [29, 32].

Concretely, Morgan and McIver defined an operator on expectations 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 [16] to probabilistic programs. For verification purposes, the wpe operator has two key strengths. First, it enables reasoning about probabilities and expected values in the output of a probabilistic program. 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 allowing the effect of the program to be applied by syntactically transforming the expectation 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 can simplify reasoning about loops: if we can find an expectation I satisfying an *invariant* condition, then the wpe of a loop can be easily bounded. The invariant condition can be checked by analyzing just the body of the loop, rather than the entire loop. Thus, finding invariants is one of the primary obstacles towards automated reasoning about probabilistic programs.

Discovering Invariants. Two recent works have considered how to automatically infer invariant expectations for probabilistic loops. The first is Prinsys [21]. 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 reliance on a template of a particular form is limiting; furthermore, the user must manually solve a system of logical formulas to produce the invariant.

The second work, by Chen et al. [14], focuses on inferring polynomial invariants. By restricting to this class, their method can avoid templates and can apply the 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 [17, 19]. In these methods, the program is executed with a variety of inputs to produce a set of execution traces. This data is viewed as a training set, and a machine learning algorithm is used to find a classifier describing the invariant. Data-driven techniques reduce the reliance on templates, and can treat the program as a black box—the precise implementation of the program need not be known, as long as the learner can execute the program to gather input and output data. In the probabilistic setting, there are a few key 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 black-box access to 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.

Outline. After covering preliminaries (Section 2), we present our contributions.

- A general method called Exist for learning invariants for probabilistic programs (Section 3). 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 set of input states given counterexamples to the invariant conditions.
- Concrete instantiations of Exist tailored for handling two problems: learning *exact invariants* (Section 4), and learning *sub-invariants* (Section 5). Our method for exact invariants learns a *model tree* [33], 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* [39] 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 6). Our tool can learn invariants and sub-invariants for examples considered in prior work and new, more difficult versions that are beyond the reach of prior work.

We discuss related work in Section 7.

2 Preliminaries

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

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

The semantics is entirely standard and can be found in Appendix A; commands map memories to distributions over memories [25]. Since we will be interested in running programs on concrete inputs, *we will assume throughout that all loops are almost surely terminating*; this property can often be established by other methods (e.g., [12, 13, 30]).

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

Definition 1. *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., [8]). We will often view numeric expressions e 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, $[flip = 0] \cdot (x + 1)$ is an expectation, and $[flip = 0] \cdot (x+1) \leq x+1$. We sometimes write G for the guard of a command.

$$
\begin{aligned}
\mathsf{wpe}(\mathbf{skip}, E) &\coloneqq E \\
\mathsf{wpe}(x \leftarrow e, E) &\coloneqq E[e/x] \\
\mathsf{wpe}(x \xleftarrow{\$} d, E) &\coloneqq \lambda\sigma. \sum_{v \in \mathcal{V}} [\![d]\!]_\sigma(v) \cdot E[v/x] \\
\mathsf{wpe}(P\,;\,Q, E) &\coloneqq \mathsf{wpe}(P, \mathsf{wpe}(Q, E)) \\
\mathsf{wpe}(\mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q, E) &\coloneqq [e] \cdot \mathsf{wpe}(P, E) + [\neg e] \cdot \mathsf{wpe}(Q, E) \\
\mathsf{wpe}(\mathbf{while}\ e : P, E) &\coloneqq \mathsf{lfp}(\lambda X.\ [e] \cdot \mathsf{wpe}(P, X) + [\neg e] \cdot E)
\end{aligned}
$$

Fig. 1: Morgan and McIver's weakest pre-expectation operator

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. 1. The case for loops takes 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 monotone on the complete lattice $\mathcal{E}$, so the least fixed-point exists by the Knaster-Tarski 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 $[\![P]\!]_\sigma$.

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

As a consequence, we can analyze the expected value of E after running P on arbitrary input state σ by computing $\mathsf{wpe}(P, E)(\sigma)$. While the definition of wpe is straightforward, one problem is that for loops, the least fixed point definition of $\mathsf{wpe}(\textbf{while}\ e : P, E)$ is hard to compute.

3 Algorithm Overview

In this section, we introduce the two related problems we aim to solve, and a meta-algorithm to tackle both of them. We will see how to instantiate the meta-algorithm's subroutines in Section 4 and Section 5.

Problem statement. As we saw in Section 2, the weakest pre-expectation calculus provides a syntactic way to compute the expected value of an expression *postE* after running a program, except when the program is a loop. Computing or bounding the weakest pre-expectations of loops is a simple task when an *invariant* or *sub-invariant* expectation is known, but these expectations can be difficult to find. Thus, we aim to develop an algorithm to automatically synthesize invariants and sub-invariants of probabilistic loops. More technically, our algorithm tackles the following two problems:

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

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

   Such an expectation I is an *exact invariant* of the loop with respect to *postE*. Since $\mathsf{wpe}(\textbf{while}\ G : P, postE)$ is a fixed point of $\Phi^{\mathsf{wpe}}_{postE}$, $\mathsf{wpe}(\textbf{while}\ e : P, E)$ has to be an exact invariant of the loop. Furthermore, when $\textbf{while}\ G : P$ is almost surely terminating and *postE* is upper-bounded, the existence of an exact invariant I implies $I = \mathsf{wpe}(\textbf{while}\ e : P, E)$. (We defer the proof to Appendix B.)
2. **Finding sub-invariants:** Given a loop $\textbf{while}\ G : P$ and expectations $preE, postE$, we aim to learn an expectation I such that

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

$$pre \leq I. \tag{3}$$

   The first inequality says that I is a sub-invariant: on states that satisfy G, the expected 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*. Any sub-invariant lower-bounds the weakest pre-expectation of the loop, i.e., $I \leq \mathsf{wpe}(\textbf{while}\ G : P, E)$ [22]. Together with the second inequality $pre \leq I$, the existence of a sub-invariant I ensures that *pre* lower-bounds the weakest pre-expectation.

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 $pre \leq I$. 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.

```
EXIST(prog, pexp, nruns, nstates):
    feat ← getFeatures(prog, pexp)
    states ← sampleStates(feat, nstates)
    data ← sampleTraces(prog, pexp, feat, nruns, states)
    while not timed out:
        models ← learnInv(feat, data)
        candidates ← extractInv(models)
        for inv in candidates:
            verified, cex ← verifyInv(inv, prog)
            if verified:
                return inv
            else:
                states ← states ∪ cex
                states ← states ∪ sampleStates(feat, nstates)
        data ← data ∪ sampleTraces(prog, pexp, feat, nruns, states)
```

Fig. 2: Algorithm EXIST

Methods. We solve both problems with one algorithm, EXIST (short for EXpectation Invariant SynThesis). Our data-driven method resembles Counterexample Guided Inductive Synthesis (CEGIS), but differs in two ways. First, candidates are synthesized by fitting a machine learning model to program trace data starting from random input states. Our target programs are 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 counterexamples can be generated by using a computer algebra system to solve an optimization problem.

We present the pseudocode in Fig. 2. EXIST takes a probabilistic program *prog*, a post-expectation or a pair of pre/post-expectation *pexp*, and hyper-parameters *nruns* and *nstates*. EXIST starts by generating a list of features *feat*, which are numerical expressions formed by program variables used in *prog*. Next, EXIST samples *nstates* initialization *states* and runs *prog* from each of

those states for *nruns* 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 initial states, generates trace data from running *prog* on each of these states for *nruns* 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.

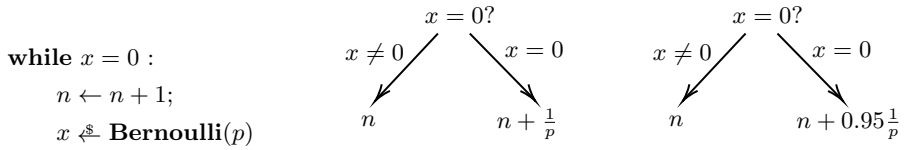
4 Learning Exact Invariants

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

At a high-level, we first sample a set of program states *states* using sampleStates. From each program state $s \in states$, sampleTraces executes *prog* and estimates $\mathsf{wpe}(prog, postE)(s)$. Next, learnInv trains regression models M to predict estimations of $\mathsf{wpe}(prog, postE)(s)$ 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 $\mathsf{wpe}(prog, postE)$, 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 $I = \Phi^{\mathsf{wpe}}_{postE}(I)$.

The reader may wonder why we took this complicated path, first estimating the weakest pre-expectation of the loop, and then computing the invariant. If we are able to learn an expression for $\mathsf{wpe}(prog, postE)$ directly, then why are we interested in the invariant I? The answer is that given the invariant I, we can also *verify* that our computed value of $\mathsf{wpe}(prog, postE)$ is correct by applying the invariant rule. 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. 3. The simple program geo repeatedly loops: whenever x becomes non-zero we exit the loop; otherwise we increase n by 1 and draw 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 $[x \neq 0] \cdot n + [x = 0] \cdot (n + \frac{1}{p})$.



(a) Program: geo (b) Model tree for $\mathsf{wpe}(\mathsf{geo}, n)$ (c) Another model tree

Fig. 3: Running example: Program and model tree

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 [33]. 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 *multiplication models* – products of constant powers of features. For example, assuming $n, \frac{1}{p}$ are both features, Fig. 3b and Fig. 3c are two model trees with linear leaf models, and Fig. 3b expresses the weakest pre-expectation $\mathsf{wpe}(\mathsf{geo}, n)$. Formally, the leaf model M is either

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

for constants $\{\alpha_i\}_i$. Note that multiplication models can also be viewed as linear models on logarithmic values because $\log M_m(x) = \sum_{i=1}^{|\mathcal{F}|} \alpha_i \cdot \log(x_i)$. We focus on linear models and multiplication models because of their simplicity and expressiveness; it is straightforward to adapt our method to other leaf models.

4.1 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., $n + 1$.

4.2 Sample Initial States (sampleStates)

Recall that Exist aims to learn an expectation I that is (approximately) equal to 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. 3, 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 wpe(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 choosing initializations that ensure perfect coverage seems like a difficult problem, our implementation uses a simpler strategy: sampleStates generates *nstates* 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 they 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 6), though other strategies are certainly possible (e.g., performing a grid search of initial states from some space).

4.3 Sample Training Data (sampleTraces)

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

$$(s_i, v_i) = \left(s_i, \frac{1}{N} \sum_{i=1}^{nruns} postE(s_i) \right).$$

Above, the value v_i is the empirical mean of $postE$ in the output state of running *prog* from initial state s_i; as *nruns* grows large, this average value approaches the true expected value $\mathsf{wpe}(prog, postE)(s)$.

4.4 Learning a Model Tree (learnInv)

Now that we have the training set $data = \{(s_1, v_1), \ldots, (s_K, v_K)\}$ (where $K = nstates$), 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 $\mathsf{wpe}(prog, postE)(s_i)$, 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

$$I = postE + [G] \cdot I' \tag{4}$$

because I being an invariant means

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

In many cases, the expectation $I' = \mathsf{wpe}(P, I) - postE$ 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.

Aiming to learn weakest pre-expectations I in the form of Eq. (4), 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

$$err_l(T_l, data) = \left(\sum_{i=1}^{K} \left(postE(s_i) + G(s_i) \cdot T_l(\mathcal{F}_l(s_i)) - v_i \right)^2 \right)^{1/2}, \tag{5}$$

where $postE(s_i)$ 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 $postE(s_i) + G(s_i) \cdot T_l(\mathcal{F}_l(s_i))$ 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 $postE(s_i) + G(s_i) \cdot T_l(\mathcal{F}_l(s_i)) = postE(s_i) = v_i$; thus, we only need to generate and collect trace data for initial states where 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.

4.5 Extracting Expectations from Models (extractInv)

Given the learned model trees T_l and T_m, we extract expectations that approximate $\mathsf{wpe}(prog, postE)$ 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. We apply several rounding schemes to generate a list of rounded model trees.
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 $postE + [G] \cdot I'$ approximates $\mathsf{wpe}(\textbf{while}\ G : P, postE)$, we construct each candidate invariant $inv \in invs$ by replacing I' in the pattern $postE + [G] \cdot I'$ by an expectation obtained in the second step.

4.6 Verify Extracted Expectations (verifyInv)

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

$$inv = [\neg G] \cdot postE + [G] \cdot \mathsf{wpe}(P, inv). \tag{6}$$

Since the learned model might not predict every data point exactly, we must verify whether inv satisfies this equality using verifyInv. If not, verifyInv looks for counterexamples that maximize 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. (6):

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

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.

5 Learning Subinvariants

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

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

Intuitively, $\Phi^{\mathsf{wpe}}_{postE}(I)$ 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 decrease each iteration, and $pre \leq I$.

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 $\Phi^{\mathsf{wpe}}_{postE}(I)(s)$. We then define a loss to sum up the violation of $I \leq \Phi^{\mathsf{wpe}}_{postE}(I)$ and $pre \leq I$ 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 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. [39]. We defer the technical details of neural model trees to Appendix C, but 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 support the sub-invariant loss function.

Outline. We will discuss changes in sampleTraces, learnInv and verifyInv for learning sub-invariants but omit descriptions of getFeatures, sampleStates, extractInv because Exist generates features, samples initial states and extracts expectations in the same way as in Section 4. To simplify the exposition, we will assume getFeatures generates the same set of features $\mathcal{F} = \mathcal{F}_l = \mathcal{F}_p$ for model trees with linear models and model trees with multiplication models.

5.1 Sample Training Data (sampleTraces)

Unlike when sampling data for learning exact invariants, here, sampleTraces runs only one iteration of the given program $prog = \textbf{while}\ G : P$ instead of running it until the loop exits. 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, sampleTraces runs the loop body P for $nruns$ times and records the set of output states reached from s_i. If executing P from s_i leads to output states $\{s_{i1}, \ldots, s_{inruns}\}$, then sampleTraces produces the training example:

$$(s_i, S_i) = (s_i, \{s_{i1}, \ldots, s_{inruns}\}),$$

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

5.2 Learning a Neural Model Tree (learnInv)

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

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

If $I = postE + [G] \cdot I'$, 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 $postE + [G] \cdot I'$. Thus, we want to train a model tree T such that T translates into an expectation I', and

$$pre \leq postE + [G] \cdot I' \tag{7}$$

$$[G] \cdot (postE + [G] \cdot I') \leq [G] \cdot \mathsf{wpe}(P, postE + [G] \cdot I') \tag{8}$$

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. (7) and $err_2(T, data)$ captures Eq. (8).

Defining err_1 is relatively simple: we sum up the one-sided difference between $pre(s)$ and $postE(s) + G(s) \cdot T(\mathcal{F}(s))$ 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,

$$err_1(T, data) := \sum_{i=1}^{K} \max\left(0, preE(s_i) - postE(s_i) - G(s_i) \cdot T(\mathcal{F}(s_i))\right). \tag{9}$$

Above, $preE(s_i)$, $postE(s_i)$, 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. (8) across state $s \in states$, but we do not have exact access to $\mathsf{wpe}(P, postE + [G] \cdot I')$ and need to approximate it 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 $[G] \cdot \mathsf{wpe}(P, postE + G \cdot I')(s_i)$ by

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

Therefore, we define

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

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, 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 discussion in Appendix C.)

5.3 Verify Extracted Expectations (verifyInv)

The verifier verifyInv is very similar to the one in Section 4 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

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

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 $pre(s) - inv(s)$ 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.

6 Evaluations

We implemented our prototype in Python, using Mathematica to verify and perform counterexample generation. We have evaluated our tool on a set of

18 benchmarks drawn from different sources in prior work [14, 21, 24]. 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 15/18 benchmarks within a timeout of 600 seconds. Our tool was able to generate these 15 invariants in reasonable time, taking between 3 to 299 seconds. The sampling phase dominates the time in most cases. We also compare Exist with a tool from prior literature, Mora [7]. 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 (§7).
- To evaluate sub-invariant learning, we created multiple problem instances for each benchmark by supplying different pre-expectations. On a total of 32 such problem instances, Exist was able to infer correct invariants in 25 cases, taking between 14 to 196 seconds.

The tables of experimental results can be found in the Appendix.

Implementation Details. For input parameters to Exist, we use $nruns = 500$ and $nstates = 500$. Besides input parameters listed in Fig. 2, we allow 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 described in Appendix D. Also, when incorporating counterexamples cex to the set of initializations $states$, we make multiple copies of each counterexample to give them more weights in the training. All experiments were conducted on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz machine with 32GB RAM running Ubuntu 18.04.4 LTS.

6.1 R1: Evaluation of the exact invariant method

Efficacy of invariant inference. Exist was able to infer provably correct invariants in 15/18 benchmarks. Out of 15 successful benchmarks, only 3 of them need user-supplied features ($\frac{1}{p}$ for GeoAr, $n \cdot p$ for Bin2 and Sum0). We provide the detailed results in Appendix E. The running time of Exist is dominated by the sampling time. However, this phase can be parallelized easily.

Failure analysis. Exist failed to generate invariants for 3/18 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.

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 [7]. Mora is designed to compute exact averages and moments of program expressions for loops that run forever, a problem that is related to, but exactly the same as, synthesizing exact invariants. While Mora is very fast, typically computing moments in just a few seconds, its restrictions on programs make it impossible to directly encode our benchmarks; for instance, it does not support conditional statements. We attempted to manually translate our benchmarks as Mora programs, e.g., by introducing additional indicator variables and converting conditional statements to arithmetic assignments (we show some examples of this translation in the Appendix). 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.

6.2 R2: Evaluation of the sub-invariant method

Efficacy of invariant inference. Exist is able to synthesize sub-invariants for 25/32 benchmarks. Tables 2 and 3 (in Appendix E) report the post-expectation ($postE$), pre-expectation ($preE$), the inferred invariant (Learned Invariant), and timings for different phases as before. Two out of 25 successful 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 $p \cdot n$. The benchmarks were all solved within 2-3 minutes. 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.

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.

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. For instance, for GeoAr with $preE = x + [z \neq 0] \cdot y \cdot (1-p)/p$, Exist could sometimes 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.

7 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 [21] 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. Unlike Prinsys, this approach doesn't need templates; 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. [18] 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.

There is also a line of work on abstract interpretation for analyzing probabilistic programs; Chakarov and Sankaranarayanan [11] search for linear expectation invariants using a "pre-expectation closed cone domain", while recent work by Wang et al. [38] employs a sophisticated algebraic program analysis approach.

Another line of work applies *martingales* to derive insights of probabilistic programs. Chakarov and Sankaranarayanan [10] showed several applications of martingales in program analysis, and Barthe et al. [5] 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. [6, 7] 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. [27], Wang et al. [37] 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. [1] proves probabilistic termination by learning ranking supermartingales from trace data. Our method for learning sub-invariants (Section 5) 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. [1] 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 [17]. ICE learning with decision trees [20] modifies the decision tree learning algorithm to capture *implication counterexample* to handle inductiveness. Hanoi [31] uses counterexample-based inductive synthesis (CEGIS) [36] to build a data-driven invariant inference engine that alternates between weakening and strengthening candidates for synthesis. Recently work uses neural networks to learn invariants [34]. These systems perform classification, while our work uses regression.

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 [2]. 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.

References

[1] Abate, A., Giacobbe, M., Roy, D.: Learning probabilistic termination proofs. In: Silva, A., Leino, K.R.M. (eds.) International Conference on Computer Aided Verification (CAV), Lecture Notes in Computer Science, vol. 12760, pp. 3–26, Springer (2021), https://doi.org/10.1007/978-3-030-81688-9_1, URL https://doi.org/10.1007/978-3-030-81688-9_1

[2] Aguirre, A., Barthe, G., Hsu, J., Kaminski, B.L., Katoen, J.P., Matheja, C.: A pre-expectation calculus for probabilistic sensitivity. Proceedings of the ACM on Programming Languages **5**(POPL) (Jan 2021), https://doi.org/10.1145/3434333, URL https://arxiv.org/abs/1901.06540, distinguished Paper Award.

[3] Albarghouthi, A., Hsu, J.: Synthesizing coupling proofs of differential privacy. Proceedings of the ACM on Programming Languages **2**(POPL) (Jan 2018), https://doi.org/10.1145/3158146, URL https://arxiv.org/abs/1709.05361

[4] Baier, C., Clarke, E.M., Hartonas-Garmhausen, V., Kwiatkowska, M.Z., Ryan, M.: Symbolic model checking for probabilistic processes. In: International Colloquium on Automata, Languages and Programming (ICALP), Bologna, Italy, Lecture Notes in Computer Science, vol. 1256, pp. 430–440, Springer (1997), https://doi.org/10.1007/3-540-63165-8_199, URL https://doi.org/10.1007/3-540-63165-8_199

[5] Barthe, G., Espitau, T., Ferrer Fioriti, L.M., Hsu, J.: Synthesizing probabilistic invariants via Doob's decomposition. In: International Conference on Computer Aided Verification (CAV), Toronto, Ontario, Lecture Notes in Computer Science, vol. 9779, pp. 43–61, Springer-Verlag (2016), https://doi.org/10.1007/978-3-319-41528-4_3, URL https://arxiv.org/abs/1605.02765

[6] Bartocci, E., Kovács, L., Stankovič, M.: Automatic generation of moment-based invariants for prob-solvable loops. In: International Symposium on Automated Technology for Verification and Analysis, pp. 255–276, Springer (2019)

[7] Bartocci, E., Kovács, L., Stankovič, M.: Mora-automatic generation of moment-based invariants. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 492–498, Springer (2020)

[8] Batz, K., Kaminski, B.L., Katoen, J., Matheja, C.: Relatively complete verification of probabilistic programs: an expressive language for expectation-based reasoning. Proceedings of the ACM on Programming Languages **5**(POPL), 1–30 (2021), https://doi.org/10.1145/3434320, URL https://doi.org/10.1145/3434320

[9] Carbin, M., Misailovic, S., Rinard, M.C.: Verifying quantitative reliability for programs that execute on unreliable hardware. In: ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Indianapolis, Indiana, pp. 33–52, ACM

(2013), https://doi.org/10.1145/2509136.2509546, URL https://doi.org/10.1145/2509136.2509546

[10] Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: International Conference on Computer Aided Verification (CAV), Saint Petersburg, Russia, pp. 511–526 (2013), URL https://www.cs.colorado.edu/~srirams/papers/cav2013-martingales.pdf

[11] Chakarov, A., Sankaranarayanan, S.: Expectation invariants for probabilistic program loops as fixed points. In: Müller-Olm, M., Seidl, H. (eds.) International Symposium on Static Analysis (SAS), Munich, Germany, Lecture Notes in Computer Science, vol. 8723, pp. 85–100, Springer (2014), https://doi.org/10.1007/978-3-319-10936-7_6, URL https://doi.org/10.1007/978-3-319-10936-7_6

[12] Chatterjee, K., Fu, H., Goharshady, A.K.: Termination analysis of probabilistic programs through Positivstellensatz's. In: International Conference on Computer Aided Verification (CAV), Toronto, Ontario, Lecture Notes in Computer Science, vol. 9779, pp. 3–22, Springer-Verlag (2016), ISBN 978-3-319-41528-4, https://doi.org/10.1007/978-3-319-41528-4_1, URL https://doi.org/10.1007/978-3-319-41528-4_1

[13] Chatterjee, K., Fu, H., Novotný, P., Hasheminezhad, R.: Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Saint Petersburg, Florida, pp. 327–342 (2016), ISBN 978-1-4503-3549-2, https://doi.org/10.1145/2837614.2837639, URL https://doi.acm.org/10.1145/2837614.2837639

[14] Chen, Y., Hong, C., Wang, B., Zhang, L.: Counterexample-guided polynomial loop invariant generation by Lagrange interpolation. In: International Conference on Computer Aided Verification (CAV), San Francisco, California, Lecture Notes in Computer Science, vol. 9206, pp. 658–674, Springer (2015), https://doi.org/10.1007/978-3-319-21690-4_44, URL https://doi.org/10.1007/978-3-319-21690-4_44

[15] Dehnert, C., Junges, S., Katoen, J., Volk, M.: A storm is coming: A modern probabilistic model checker. In: International Conference on Computer Aided Verification (CAV), Heidelberg, Germany, Lecture Notes in Computer Science, vol. 10427, Springer-Verlag (2017), URL http://arxiv.org/abs/1702.04311

[16] Dijkstra, E.W.: Guarded commands, non-determinancy and a calculus for the derivation of programs. In: Bauer, F.L., Samelson, K. (eds.) Language Hierarchies and Interfaces, International Summer School, Marktoberdorf, Germany, Lecture Notes in Computer Science, vol. 46, pp. 111–124, Springer (1975), https://doi.org/10.1007/3-540-07994-7_51, URL https://doi.org/10.1007/3-540-07994-7_51

[17] 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. **69**(1–3), 35–45 (2007), https://doi.org/10.1016/j.scico.2007.01.015, URL https://doi.org/10.1016/j.scico.2007.01.015

[18] Feng, Y., Zhang, L., Jansen, D.N., Zhan, N., Xia, B.: Finding polynomial loop invariants for probabilistic programs. In: International Symposium on Automated Technology for Verification and Analysis, pp. 400–416, Springer (2017)

[19] Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for esc/-java. In: Oliveira, J.N., Zave, P. (eds.) International Symposium of Formal Methods Europe (FME), Berlin, Germany, Lecture Notes in Computer Science, vol. 2021, pp. 500–517, Springer (2001), https://doi.org/10.1007/3-540-45251-6_29, URL https://doi.org/10.1007/3-540-45251-6_29

[20] Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. SIGPLAN Not. **51**(1), 499512 (Jan 2016), ISSN 0362-1340, https://doi.org/10.1145/2914770.2837664, URL https://doi.org/10.1145/2914770.2837664

[21] Gretz, F., Katoen, J., McIver, A.: Prinsys - on a quest for probabilistic loop invariants. In: International Confnerece on Quantitative Evaluation of Systems (QEST), Buenos Aires, Argentina, Lecture Notes in Computer Science, vol. 8054, pp. 193–208, Springer (2013), https://doi.org/10.1007/978-3-642-40196-1_17, URL https://doi.org/10.1007/978-3-642-40196-1_17

[22] Kaminski, B.L.: Advanced weakest precondition calculi for probabilistic programs. Ph.D. thesis, RWTH Aachen University, Germany (2019), URL http://publications.rwth-aachen.de/record/755408

[23] Kaminski, B.L., Katoen, J., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected run-times of probabilistic programs. In: European Symposium on Programming (ESOP), Eindhoven, The Netherlands, Lecture Notes in Computer Science, vol. 9632, pp. 364–389, Springer-Verlag (2016), https://doi.org/10.1007/978-3-662-49498-1_15, URL https://dx.doi.org/10.1007/978-3-662-49498-1_15

[24] Kaminski, B.L., Katoen, J.P.: A weakest pre-expectation semantics for mixed-sign expectations. In: 2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), pp. 1–12, IEEE (2017)

[25] Kozen, D.: Semantics of probabilistic programs. Journal of Computer and System Sciences **22**(3), 328–350 (1981), https://doi.org/10.1016/0022-0000(81)90036-2, URL https://www.sciencedirect.com/science/article/pii/0022000081900362

[26] Kozen, D.: A probabilistic PDL. Journal of Computer and System Sciences **30**(2) (1985), https://doi.org/10.1016/0022-0000(85)90012-1

[27] Kura, S., Urabe, N., Hasuo, I.: Tail probabilities for randomized program runtimes via martingales for higher moments. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 135–153, Springer (2019)

[28] Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: International Conference on Computer Aided Verification (CAV), Snowbird, Utah, Lecture Notes in Computer Science, vol. 6806, pp. 585–591, Springer-Verlag (2011)

[29] McIver, A., Morgan, C.: Abstraction, Refinement, and Proof for Probabilistic Systems. Monographs in Computer Science, Springer-Verlag (2005)

[30] McIver, A., Morgan, C., Kaminski, B.L., Katoen, J.: A new proof rule for almost-sure termination. Proceedings of the ACM on Programming Languages **2**(POPL), 33:1–33:28 (2018), https://doi.org/10.1145/3158121, URL http://doi.acm.org/10.1145/3158121

[31] Miltner, A., Padhi, S., Millstein, T., Walker, D.: Data-driven inference of representation invariants. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, p. 115, PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020), ISBN 9781450376136, https://doi.org/10.1145/3385412.3385967, URL https://doi.org/10.1145/3385412.3385967

[32] Morgan, C., McIver, A., Seidel, K.: Probabilistic predicate transformers. ACM Transactions on Programming Languages and Systems **18**(3), 325–353 (1996), https://doi.org/10.1145/229542.229547, URL dl.acm.org/ft_gateway.cfm?id=229547

[33] Quinlan, J.R.: Learning with continuous classes. In: Australian Joint Conference on Artificial Intelligence (AI), Hobart, Tasmania, vol. 92, pp. 343–348 (1992)

[34] Si, X., Dai, H., Raghothaman, M., Naik, M., Song, L.: Learning loop invariants for program verification. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems, p. 77627773, NIPS'18, Curran Associates Inc., Red Hook, NY, USA (2018)

[35] Smith, C., Hsu, J., Albarghouthi, A.: Trace abstraction modulo probability. Proceedings of the ACM on Programming Languages **3**(POPL) (Jan 2019), https://doi.org/10.1145/3290352, URL https://arxiv.org/abs/1810.12396

[36] Solar-Lezama, A.: Program sketching. Int. J. Softw. Tools Technol. Transf. **15**(56), 475495 (Oct 2013), ISSN 1433-2779, https://doi.org/10.1007/s10009-012-0249-7, URL https://doi.org/10.1007/s10009-012-0249-7

[37] Wang, D., Hoffmann, J., Reps, T.: Central moment analysis for cost accumulators in probabilistic programs (2021)

[38] Wang, D., Hoffmann, J., Reps, T.W.: PMAF: an algebraic framework for static analysis of probabilistic programs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Philadelphia, Pennsylvania, pp. 513–528 (2018), https://doi.org/10.1145/3192366.3192408, URL https://doi.org/10.1145/3192366.3192408

[39] Yang, Y., Morillo, I.G., Hospedales, T.M.: Deep neural decision trees. CoRR **abs/1806.06988** (2018), URL http://arxiv.org/abs/1806.06988

A Omitted 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); we let Σ denote the set of all program states. The semantics of probabilistic programs is defined in terms of distributions. To avoid measure-theoretic technicalities, we assume that $\mathcal{V}$ is countable.

Definition 2. *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 $\mathbf{Dist}(S)$.*

As is standard, programs P are interpreted as maps $[\![P]\!] : \Sigma \to \mathbf{Dist}(\Sigma)$. This definition requires two standard operations on distributions.

Definition 3. *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. 4; we comment on a few details here. First, given a state σ, we interpret expressions e and distribution expressions d as values $[\![e]\!]_\sigma \in \mathcal{V}$ and distributions over values $[\![d]\!]_\sigma \in \mathbf{Dist}(\mathcal{V})$, respectively; we implicitly assume that all expressions are well-typed. Second, 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 we will be interested in running programs on concrete inputs, *we will assume throughout that all loops are AST*; this property can often be established by other methods (e.g., [12, 13, 30]).

B Omitted proofs

Theorem 2. *If* **while** $e : P$ *is almost surely terminating, P is loop-free, and E is an expectation bounded by some constant k, denoted by $E \leq \mathcal{E}_{\leq k}$, we have:*

$$I = [e] \cdot \mathsf{wpe}(P, I) + [\neg e] \cdot E \iff I = \mathsf{wpe}(\textbf{while } G : P, E).$$

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

Fig. 4: Program semantics

Proof. The reverse direction of Theorem 2 follows from the definition of wpe:

$$\mathsf{wpe}(\mathbf{while}\ e : P, E) \coloneqq \mathsf{lfp}(\Phi_E^{\mathsf{wpe}}).$$

so $I = \mathsf{wpe}(\mathbf{while}\ e : P, E)$ must be a fixed point of Φ_E^{wpe}.

For the forward direction, we first show that it holds if E is bounded by 1, and then for E bounded by k, since E/k is bounded by 1, we have

$$I/k = [e] \cdot \mathsf{wpe}(P, I/k) + [\neg e] \cdot E/k \implies I/k = \mathsf{wpe}(\mathbf{while}\ G : P, E/k).$$

By linearity of wpe, $k \cdot \mathsf{wpe}(\mathbf{while}\ G : P, E/k) = \mathsf{wpe}(\mathbf{while}\ G : P, E)$, and $k \cdot \mathsf{wpe}(P, I/k) = \mathsf{wpe}(P, k \cdot I)$. Thus, we have

$$
\begin{aligned}
& I = [e] \cdot \mathsf{wpe}(P, I) + [\neg e] \cdot E \\
\implies & I/k = [e] \cdot \mathsf{wpe}(P, I/k) + [\neg e] \cdot E/k \\
\implies & I/k = \mathsf{wpe}(\mathbf{while}\ G : P, E/k) \\
\implies & I = \mathsf{wpe}(\mathbf{while}\ G : P, E).
\end{aligned}
$$

To prove that the forward direction holds for $E \leq \mathcal{E}_{\leq 1}$, we first show that the characteristic function has a unique fixed point. We first introduce the *weakest liberal pre-expectation* transformer wlpe, a dual notion to the wpe transformer. Formally, wlpe is defined the same way as wpe, except that all wpe are replaced by wlpe, and

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

See, e.g., Kaminski [22, Table 4.2] for the full definition. By Kaminski [22, Theorem 4.11], for any **pWhile** program Q, $E' \in \mathcal{E}_{\leq 1}$,

$$\mathsf{wlpe}(Q, E') = \lambda\sigma. \sum_{\sigma' \in \Sigma} E'(\sigma') \cdot [\![Q]\!]_\sigma(\sigma') + (1 - \sum_{\sigma' \in \Sigma} [\![Q]\!]_\sigma(\sigma')).$$

Note that $\sum_{\sigma' \in \Sigma} [\![\textbf{while } e : P]\!]_\sigma(\sigma') = 1$ because the loop is almost surely terminating, and $E' \in \mathcal{E}_{\leq 1}$, so

$$\begin{aligned}
&\mathsf{wlpe}(\textbf{while } e : P, E') \\
&= \lambda\sigma. \sum_{\sigma' \in \Sigma} E'(\sigma') \cdot [\![\textbf{while } e : P]\!]_\sigma(\sigma') + (1 - \sum_{\sigma' \in \Sigma} [\![\textbf{while } e : P]\!]_\sigma(\sigma')) \\
&= \lambda\sigma. \sum_{\sigma' \in \Sigma} E'(\sigma') \cdot [\![\textbf{while } e : P]\!]_\sigma(\sigma') \\
&= \mathsf{wpe}(\textbf{while } e : P, E'). \qquad \text{(By Theorem 1)}
\end{aligned}$$

Unfolding the definition of wlpe and wpe, we get that for any $E' \in \mathcal{E}_{\leq 1}$,

$$\mathsf{lfp}(\Phi^{\mathsf{wpe}}_{E'}) = \mathsf{gfp}(\Phi^{\mathsf{wlpe}}_{E'}). \tag{10}$$

The loop is almost surely terminating, so P is also almost surely terminating, and using the same reasoning, $\mathsf{wlpe}(P, F) = \mathsf{wpe}(P, F)$ for any $F \in \mathcal{E}$. Thus, for any $F \in \mathcal{E}$,

$$[e] \cdot \mathsf{wlpe}(P, F) + [\neg e] \cdot E = [e] \cdot \mathsf{wpe}(P, F) + [\neg e] \cdot E.$$

Therefore, $\Phi^{\mathsf{wlpe}}_E = \Phi^{\mathsf{wpe}}_E$, and specifically,

$$\mathsf{gfp}(\Phi^{\mathsf{wlpe}}_E) = \mathsf{gfp}(\Phi^{\mathsf{wpe}}_E). \tag{11}$$

Since $E \in \mathcal{E}_{\leq 1}$, combining Eq. (10) and Eq. (11), we get $\mathsf{lfp}(\Phi^{\mathsf{wpe}}_E) = \mathsf{gfp}(\Phi^{\mathsf{wpe}}_E)$. Thus, Φ^{wpe}_E must have a unique fixed point.

Returning to the proof, the left hand side, $I = \Phi^{\mathsf{wpe}}_E(I)$, implies that I is a fixed point of Φ^{wpe}_E. Since Φ^{wpe}_E has a unique fixed point, I must also be the least fixed point of Φ^{wpe}_E, and the right hand side follows. □

C Neural Model Trees.

Difficulty of training with standard algorithms. When calculating the error contributed by one training example ((s_i, v_i) or (s_i, S_i)) in err_l, err_m or err_1, the training algorithm only needs to evaluate the model tree in the training (T_l, T_m, or T) on the feature vector of *one* state; thus, either the leaf model on T's root is used , or exactly *one* children subtree of T is used. The training algorithm is thus able to dispatch each training example to one subtree of T and independently optimize each subtree of T using training examples passed to each subtree as data. However, with err_2, the error of model tree T on one training example (s_i, S_i) depends on T's predications on different states: $T(\mathcal{F}(s_i))$ and $T(\mathcal{F}(s_{ij}))$ for $s_{ij} \in S_i$. Evaluating $T(\mathcal{F}(s_i))$ and all $T(\mathcal{F}(s_{ij}))$ can use both children subtrees of T. Thus, we cannot dispatch each training example to one subtree of T and optimize each subtree independently. Furthermore, because of the use of $\max(0, -)$ function in err_2, it seems impossible to solve this problem by rearranging the terms.

Constructions of Neural Model Trees The essence of neural networks is that they are functions with easily differentiable parameters. When we apply a model tree M to *data*, the two important steps are

1. Dispatching training examples in *data* to leaves according to predicates on internal nodes;
2. Applying leaf models to training examples passed to each leaf.

Adapting the neural decision tree developed by Yang et al. [39], we have a differentiable model for the first step: given a set of features $\mathcal{F}$ we can split on, we have a neural decision tree nndt : trainable parameters $\to$ (data $\to \{0,1\}^{|2^{\mathcal{F}}|}$), with a trainable cut point c_i for each feature $e_i \in \mathcal{F}$, such that $\mathsf{nndt}(\{c_i\})$ maps a data example d to a one-hot vector of length $2^{\mathcal{F}}$; the hot bit of the one-hot vector represents the leaf d gets sent to according to d's satisfaction of each predicate $e_i \leq c_i$. Thus, we can construct a neural model tree nnmt by applying the leaf model to the results of nndt. Specifically, nnmt : trainable parameters $\to$ (data $\to \mathbb{R}$) has $2^{\mathcal{F}}$ leaf models, represented as a $2^{\mathcal{F}}$ vector, and all the cut points c_i as trainable parameters, and given data d,

$$\begin{aligned}&\mathsf{nnmt}(\text{leaf model vector}, \{c_i\}))(d)\\ &= \langle \mathsf{nnmt}(\text{leaf model vector}, \{c_i\}))(d), \text{leaf model vector}\rangle(d),\end{aligned}$$

where the inner product $\langle \mathsf{nnmt}(\text{leaf model vector}, \{c_i\}))(d), \text{leaf model vector}\rangle$ selects the leaf model on the leaf the data d gets mapped to, and we then apply the selected leaf model back to d. When the leaf models are linear, the application of a nnmt model can be implemented through matrix multiplication, and then nnmt is differentiable with respect to its trainable parameters. Also, 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.

D Hyperparameters and Implementation Details

D.1 Feature Generation

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). 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*, $\mathsf{getFeatures}$ generates

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

D.2 Rounding Schemes

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. 3c to integers gives us the model tree in Fig. 3b. 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 $round(c, digit) \cdot \prod_{i=1}^{|\mathcal{F}|} x_i^{\text{int}(a_i)}$.

E Experimental result

Results for exact invariant synthesis can be found in Table 1, and results for sub-invariant synthesis can be found in Tables 2 and 3.

Table 1 summarizes the results obtained when synthesizing exact invariants. Table 2 and Table 3 summarizes the results obtained when synthesizing sub invariants. In these tables, we list the post-expectation ($postE$), pre-expectation ($preE$), the inferred invariant (Learned Invariant), time for sampling data (ST), time for model learning (LT), the time for verification checks (VT) and the total time (TT).

F Encoding our benchmarks in Mora's syntax

?? shows how we encoded two of our benchmarks into Mora [7].

G Benchmark programs

For benchmarks that have multiple similar variations, we omit some simpler ones.

```
int z, bool flip, float p1
while (flip == 0):
    d = bernoulli.rvs(size=1, p=p1)[0]
    if d:
        flip = 1
    else:
        z = z + 1
```

Fig. 5: Geo0: Adapted from Listing 1 from Prinsys paper

```
int z, x, bool flip, float p1
while (flip == 0):
    d = bernoulli.rvs(size=1, p=p1)[0]
    if d:
        flip = 1
    else:
        x = x * 2
        z = z + 1
```

Fig. 6: Geo1: Adapted from Fig. 5

```
int count, bool c1, c2 float p1, p2
while not (c1 or c2):
    c1 = bernoulli.rvs(size=1, p=p1)[0]
    if c1:
        count = count + 1
    c2 = bernoulli.rvs(size=1, p=p2)[0]
    if c2:
        count = count + 1
```

Fig. 7: Fair: Two coins in one loop

```
int c, b, rounds, float p
while b > 0:
    d = bernoulli.rvs(size=1, p=p)
    if d:
        c = c+b
        b = 0
    else:
        c = c-b
        b = 2*b
    rounds += 1
```

Fig. 8: Mart: Martingale, Listing 4 from Prinsys paper

```
int x, y, z, float p
while 0 < x and x < y:
    d = bernoulli.rvs(size=1, p=p)[0]
    if d:
        x = x + 1
    else:
        x = x - 1
    z = z + 1
    rounds += 1
```

Fig. 9: Gambler: Gambler's ruin

```
int x, y, z, float p
while not (z == 0):
    y = y + 1
    d = bernoulli.rvs(size=1, p=p)[0]
    if(d):
        z = 0
    else:
        x = x + y
```

Fig. 10: GeoAr: Geometric distribution mixed with Arithmetic progression

```
int x, y, n, float p
while(n > 0):
    d = bernoulli.rvs(size=1, p=p)[0]
    if(d):
        x = x + y
    n = n-1
```

Fig. 11: Bin0: A plain binomial distribution

```
int x, n, M, float p
while n - M < 0:
    d = bernoulli.rvs(size=1, p=p)[0]
    if d:
        x = x + 1
    n = n + 1
```

Fig. 12: Bin1: A binomial distribution with an unchanged variable

```
int x, y, n, float p
while(n > 0):
    d = bernoulli.rvs(size=1, p=p)[0]
    if(d):
        x = x + n
    else:
        x = x + y
    n = n-1
```

Fig. 13: Bin2: Binomial distribution mixed with arithmetic progression sums

```
int n, count
while(n > 0):
    x1 = bernoulli.rvs(size=1, p=0.5)[0]
    x2 = bernoulli.rvs(size=1, p=0.5)[0]
    x3 = bernoulli.rvs(size=1, p=0.5)[0]
    n = n - 1
    c1 = x1 or x2 or x3
    c2 = (not x1) or x2 or x3
    c3 = x1 or (not x2) or x3
    count = count + c1 + c2 + c3
```

Fig. 14: LinExp: Mix of binomial distribution and linearity of expectation

```
int x, n, float p
while(n > 0):
    d = bernoulli.rvs(size=1, p=p)[0]
    if(d):
        x = x + n
    n = n - 1
```

Fig. 15: Sum0: Probabilistic sum of arithmetic series

```
int x, y, n, float p
while(n > 0):
    d = bernoulli.rvs(size=1, p=p)[0]
    if(d):
        x = x + 1
    else:
        y = y + 1
    n = n - 1
```

Fig. 16: DepRV: Product of dependent random variables

```
int x, float p1, p2
while(x == 0):
    d1 = bernoulli.rvs(size=1, p=p1)[0]
    if(d1):
        x = 0
    else:
        d2 = bernoulli.rvs(size=1, p=p2)[0]
        if(d2):
            x = -1
        else:
            x = 1
```

Fig. 17: Prinsys: Listing 2 from Prinsys paper

```
bool c, t, float  p1, p2
while(c == 1):
   if t:
       d1 = bernoulli.rvs(size=1, p=p1)[0]
       if d1:
           c = 0
       else:
           t = not t
   else:
       d2 = bernoulli.rvs(size=1, p=p2)[0]
       if d2:
           c = 0
       else:
           t = not t
```

Fig. 18: Duel: Duelling cowboys

```
int x, count
while(x <= 10):
    x = x + 1
        count = count + 1
```

Fig. 19: Detm: Deterministic loop

```
int x, z, float p
while(x-1 >= 0):
    d = bernoulli.rvs(size=1, p=p)[0]
    if(d):
        x = x - 1
    z = z + 1
```

Fig. 20: RevBin: A "reversed" binomial distribution

Table 1: Exact Invariants generated by Exist

Name	postE	Learned Invariant	(sec)	(sec)	(sec)	(sec)
BiasDir	x	$[x == y] \cdot (-0.5 \cdot x$ $-0.5 \cdot y + 0.5) + x$	27.75	0.37	2.65	30.76
Bin0	x	$[n > 0] \cdot ([y == 0] \cdot 0 +$ $[y! = 0] \cdot p \cdot n \cdot y) + x$	88.77	4.93	3.37	97.07
Bin1	n	$[n < M] \cdot$ $(-n + M) + n$	37.17	11.81	2.60	51.58
Bin2	x	$[n > 0] \cdot (p \cdot n \cdot n$ $-p \cdot n \cdot y + n \cdot y) + x$	84.64	26.54	0.48	111.66
DepRV	$x \cdot y$	-	-	-	-	-
Detm	$count$	$[x <= 10] \cdot$ $(-x + 11) + count$	0.14	0.48	11.82	12.44
Fair	$count$	$[c1 + c2 == 0] \cdot ((p1 + p2)$ $/((p1 + p2) - p1 \cdot p2))$ $+count$	6.86	2.45	0.30	9.61
Duel	t	-	-	-	-	-
Gambler	z	$[x > 0 \& y > x] \cdot$ $(x \cdot (y - x)) + z$	149.69	1.62	11.04	162.35
Geo0	z	$[flip == 0] \cdot ([z <= 0.127] \cdot$ $(1 - p1)/p1 + [z > 0.127] \cdot$ $(1 - p1)/p1) + z$	90.19	3.93	5.53	99.65
Geo1	z	$[flip == 0] \cdot$ $((1 - p1)/p1) + z$	14.12	2.07	0.02	16.21
Geo2	z	$[flip == 0] \cdot$ $((1 - p1)/p1) + z$	16.09	2.08	0.02	18.19
GeoAr	x	$[z \neq 0] \cdot ([z <= 0.127] \cdot$ $(1 - p1)/p1) + [z > 0.127] \cdot$ $(1 - p1)/p1) + x$	90.19	3.93	5.53	99.65
LinExp	z	-	-	-	-	-
Mart	$rounds$	$[b > 0] \cdot (1/p) + rounds$	30.62	4.21	0.04	34.87
Prinsys	$[x == 1]$	$[x == 0] \cdot (-p2 + 1) +$ $([x == 1])$	2.61	0.37	0.06	3.04
RevBin	z	$[x > 0] \cdot (x/p) + z$	297.19	2.26	0.05	299.5
Sum0	x	$[n > 0] \cdot (0.5 \cdot p \cdot n \cdot n +$ $0.5 \cdot p \cdot n) + x$	88.07	4.26	5.05	97.38

Table 2: Sub-Invariants generated by EXIST

Name	postE	preE	Learned Invariant	(s)	(s)	(s)	(s)
BiasDir	x	$[x \neq y] \cdot x$	$[x == y] \cdot (0.1 \cdot p - 0.5 \cdot x - 0.5 \cdot y + 0.1) + x$	14.93	48.62	3.24	66.79
		$[x == y] \cdot 1/2$	$[x == y] \cdot (0.1 \cdot p - 0.5 \cdot x - 0.5 \cdot y + 0.5) + x$	15.39	69.79	9.52	94.7
Bin0	x	$x + [n > 0] \cdot (p \cdot n \cdot y)$	-	-	-	-	-
		x	$[n > 0] \cdot 0 + x$	19.64	34.33	0.54	54.51
Bin1	n	$n + [n < M] \cdot (-p \cdot n + p \cdot M)$	-	-	-	-	-
		n	$[n < M] \cdot 0 + n$	9.02	28.94	18.41	56.37
Bin2	x	$x + [n > 0] \cdot (1 - p) \cdot n \cdot y$	-	-	-	-	-
		x	$[n > 0] \cdot 0 + x$	19.10	32.71	4.70	56.51
DepRV	$x \cdot y$	$x \cdot y + [n > 0] \cdot (1/4 \cdot n \cdot n)$	-	-	-	-	-
		$x \cdot y$	$[n > 0] \cdot 0 + x \cdot y$	18.99	41.97	0.23	61.20
Detm	$count$	$count + [x <= 10] \cdot 1$	$[x <= 10] \cdot 1 + count$	4.10	42.19	0.11	46.40
Duel	t	$1 + c \cdot (-p2 / (p1 + p2 - p1 \cdot p2))$	-	-	-	-	-
Fair	$count$	$count + [c1 + c2 == 0] \cdot (p1 + p2)$	$[c1 + c2 == 0] \cdot (p1 + p2) + count$	11.92	123.66	2.12	137.68
		$count$	$[c1 + c2 == 0] \cdot 0 + count$	8.33	21.19	0.55	30.07
Gambler	z	z	$[x > 0 \& y > x] \cdot 0 + z$	10.33	69.88	2.54	82.75
		$x \cdot (y - x)$	$[x > 0 \& y > x] \cdot x \cdot (y - x) + z$	8.62	22.28	2.92	33.82
Geo0	z	$z + [flip == 0] \cdot (1 - p1)$	$[flip == 0] \cdot (-p1 + 1) + z$	10.33	69.88	2.54	82.75
		z	$[flip == 0] \cdot 0 + z$	9.65	21.26	2.39	33.3
		$[flip == 0] \cdot (1 - p1)$	$[flip == 0] \cdot (-p1 + 1) + z$	10.70	67.82	2.47	80.99
Geo1	z	z	$[flip == 0] \cdot 0 + z$	11.18	26.93	3.28	41.39
Geo2	z	z	$[flip == 0] \cdot 0 + z$	11.64	27.16	0.07	38.87

Table 3: Table 2 Continued

Name	postE	preE	Learned Invariant (sec)	(sec)	(sec)	(sec)	(sec)
	x	$x + [z! = 0]\cdot$ $y \cdot (1-p)/p$	-	-	-	-	-
		x	$[z! = 0] \cdot 0 + x$	10.69	31.66	0.41	42.76
LinExp	z	$z + [n > 0] \cdot 2$	$[n > 0]\cdot$ $(n+1) + z$	37.05	69.20	0.09	106.34
		$z + [n > 0] \cdot 2 \cdot n$	$[n > 0] \cdot 2 \cdot n + z$	37.29	69.34	0.71	107.34
Mart	$rounds$	$rounds+$ $[b > 0] \cdot 1$	$[b > 0] \cdot 1+$ $rounds$	23.13	67.27	0.14	90.54
		$rounds$	$[b > 0] \cdot 0 + rounds$	19.82	32.17	0.12	52.13
Prinsys	$[x == 1] \cdot 1$	$[x == 1] \cdot 1$	$[x == 0] \cdot (0)+$ $[x == 1]$	1.40	12.70	0.47	14.57
RevBin	z	$[x > 0] \cdot (x) + z$	$[x > 0] \cdot x + z$	25.92	168.13	2.20	196.25
		z	$[x > 0] \cdot 0 + z$	18.87	29.14	0.15	48.18
Sum0	x	$x + [n > 0]\cdot$ $(p \cdot n \cdot n/2)$	-	-	-	-	-
		$x + [n > 0]\cdot$ $(p \cdot n/2)$	$[n > 0]\cdot$ $(p \cdot n) + x$	20.38	71.23	1.40	93.01

Table 4: Encoding of GeoAr and Mart in Mora Syntax

Program	Encoding