Repairing Decision-Making Programs Under Uncertainty

Aws Albarghouthi, Loris D'Antoni, and Samuel Drews(✉)

University of Wisconsin-Madison, Madison, USA
sedrews@wisc.edu

Abstract. The world is uncertain. Programs can be wrong. We address the problem of *repairing a program under uncertainty*, where program inputs are drawn from a probability distribution. The goal of the repair is to construct a new program that satisfies a probabilistic Boolean expression. Our work focuses on loop-free decision-making programs, e.g., classifiers, that return a Boolean- or finite-valued result. Specifically, we propose *distribution-guided inductive synthesis*, a novel program repair technique that iteratively (*i*) *samples* a finite set of inputs from a probability distribution defining the precondition, (*ii*) synthesizes a *minimal repair* to the program over the sampled inputs using an SMT-based encoding, and (*iii*) *verifies* that the resulting program is correct and is *semantically close* to the original program. We formalize our algorithm and prove its correctness by rooting it in computational learning theory. For evaluation, we focus on repairing machine learning classifiers with the goal of making them *unbiased* (fair). Our implementation and evaluation demonstrate our approach's ability to repair a range of programs.

1 Introduction

Program repair is the problem of modifying a program P to produce a new program P' that satisfies some desirable property. A majority of the investigations in automatic program repair target deterministic programs and Boolean properties, e.g., assertion violations [10,17,20,22,27]. The world, however, is uncertain, and program correctness is not always a Boolean, black-or-white property.

In this paper, we address the problem of automating program repair in the presence of uncertainty. By uncertainty, we mean that the inputs to the program are drawn from some probability distribution D. Thus, we have a *probabilistic precondition*—for instance, the input x to a program $P(x)$ may follow a Laplacian distribution. The correctness property of interest is a *probabilistic postcondition*, which we define as an expression over probabilities of program outcomes. For instance, we might be interested in ensuring that $P(r > 0) > 0.9$—the program returns a positive value at least 90% of the time—or that $P(r_1 > 0) > P(r_2 > 0)$—it is more likely that the return value r_1 is positive than that r_2 is positive. We restrict our attention to loop-free programs that return Boolean-valued (or finite-valued) results, e.g., machine learning classifiers that map inputs to a finite

R. Majumdar and V. Kunčak (Eds.): CAV 2017, Part I, LNCS 10426, pp. 181–200, 2017.
DOI: 10.1007/978-3-319-63387-9_9

set of classes, and repairs that consist of altering real-valued constants in the program.

Technique: Distribution-Guided Inductive Synthesis. To address the program repair problem in the presence of uncertainty, we propose a novel program synthesis technique that we call *distribution-guided inductive synthesis* (DIGITS). The overall flow of DIGITS is illustrated in Fig. 1. Suppose we have a program P such that $\{pre\}P\{post\}$ does not hold. The goal of DIGITS is to construct a new program P' that is correct with respect to *pre* and *post* and that is *semantically close* to P. To do so, DIGITS tightly integrates three phases:

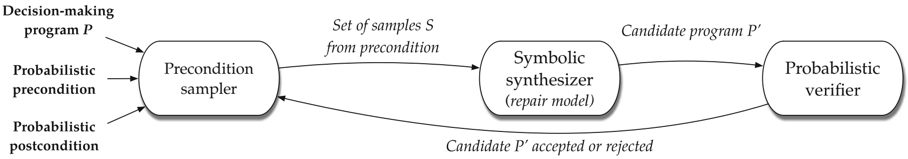
Decision-making program P
Probabilistic precondition
Probabilistic postcondition
Precondition sampler
Set of samples S from precondition
Symbolic synthesizer (repair model)
Candidate program P′
Probabilistic verifier
Candidate P′ accepted or rejected

Fig. 1. Abstract, high-level view of distribution-guided inductive synthesis

Sampling: Since the precondition *pre* is a probability distribution, DIGITS begins by sampling a finite set S of program inputs from *pre*—we call S the set of *samples*. The set S is used to sidestep having to deal with arbitrary distributions directly in the synthesis process.

Synthesis: The second step is a synthesis phase, where DIGITS searches for a set of candidate programs $\{P'_1, \ldots, P'_n\}$—following a given *repair model*—where each P'_i classifies the set of samples S differently. Given that there are exponentially many ways to partition the set S, DIGITS employs a novel trie-like data structure with *conflict-driven pruning* to avoid considering redundant partitions.

Quantitative verification: Every generated candidate program P' is checked for correctness and for close *semantic distance* with P. Specifically, DIGITS employs an automated probabilistic inference technique.

Theory: Computational Learning. We formalize DIGITS by posing it as a learning algorithm, and rooting it in computational learning theory [18]. Using the concept of the *Vapnik–Chervonenkis (VC) dimension* [7] of the repair model, we show that the algorithm converges to the optimal program with a high probability when operating over postconditions that satisfy a benign property and over repair models with a finite VC dimension, which holds for many repair scenarios, e.g., *sketching*-like approaches [26].

Application: Repairing Biased Programs. Our primary motivation for this work is repairing bias in decision-making programs, e.g., programs that decide whether to hire a person, to give them a loan, or other sensitive or potentially impactful decisions like prison sentencing [5]. These programs can be generated automatically as classifiers using machine learning or can be written by hand using expert insight. The problem of algorithmic bias has received considerable attention recently, due to the increasing deployment of automated decision-making in sensitive domains [12–14,16,19,23].

Existing notions of bias in the literature neatly correspond to probabilistic postconditions. For instance, *group fairness* [11–14] stipulates that the probability that a minority job applicant is hired is almost the same as that of a majority applicant being hired. We view the underlying population of applicants as a probabilistic precondition and pose the problem of repairing biased programs within our framework: the problem is to find a new, semantically close program that is unbiased. We implemented our approach, DIGITS, and applied it to *unbias* a range of classification programs that were generated automatically. Our results demonstrate (*i*) our technique's ability to repair a range of programs and (*ii*) the importance of our algorithmic contributions.

Contributions. We summarize our contributions as follows:

- We formalize the probabilistic program repair problem as an optimization problem whose solution is a repaired program that satisfies some probabilistic pre-/post-conditions.
- We present *distribution-guided inductive synthesis*, DIGITS, a novel synthesis methodology for automatically repairing loop-free programs under uncertain inputs and probabilistic postconditions.
- We formalize correctness of our algorithm and prove its convergence using the concept of VC dimension that is standard in computational learning theory and machine learning.
- We present an implementation of our technique, DIGITS. We apply DIGITS to the increasingly important problem of ensuring that decision-making programs are not biased, for a given particular notion of bias. Our thorough evaluation demonstrates the utility of our approach and the importance of our design decisions.

2 Illustrative Example

In this section, we illustrate the operation of DIGITS on a very simple example inspired by *algorithmic bias* problems [4].

Example Program. Consider the following program, hire:

```
fun hire(min, urank)
  dec = 1 <= urank <= 10
  return dec
```

hire is an extremely simplified automatic hiring program: it takes an applicant's information and decides whether to hire them or not, as indicated by the Boolean return variable dec. Specifically, hire takes as input the Boolean variable min, which indicates whether an applicant belongs to an underrepresented minority, and urank, which is a real-valued number indicating the rank of the university they attended. hire only hires applicants who attended top-10 universities.

Probabilistic Precondition. Let us now consider a probabilistic precondition for the program, which is a joint probability distribution over the variables min and urank. Intuitively, the precondition paints a picture of the relation between minority status and the university rank in the population of potential applicants. Consider the following precondition *pre*:

$$\begin{aligned} \mathsf{min} &\sim \mathsf{Bernoulli}(0.1) \\ \mathsf{urank} &\sim \mathsf{Gaussian}(10, 10) + 5 * \mathbb{1}(\mathsf{min}) \end{aligned}$$

Intuitively, 10% of the possible applicants are minorities, and the university rank of an applicant is drawn from a Gaussian distribution centered at 10 (with std. 10), if the applicant is not a minority. Otherwise, if the applicant is a minority, their university rank is a Gaussian centered around 15—as shown using the indicator function, $\mathbb{1}(\mathsf{min})$, which returns 1 when min is true and 0 otherwise.

Probabilistic Postcondition. The following postcondition formula asserts that *the probability of hiring minority applicants is at least 0.8 of the probability of hiring majority applicants*:

$$post \triangleq \frac{\mathbb{P}(\mathsf{dec} \mid \mathsf{min})}{\mathbb{P}(\mathsf{dec} \mid \neg\mathsf{min})} > 0.8$$

This is one of the many properties proposed to formalize notions of *fairness* in automated decision-making. This property is known as *group fairness* [13], and it is inspired by employment guidelines in the United States [3].

The postcondition does not hold for hire: Even though hire does not access the variable min, it only accepts applicants from top-10 universities, and, as per *pre*, minority applicants are less likely to attend top-10 universities; in fact, the value of the left hand side of *post* is ∼ 0.6.

Repair Model. We would like to automatically *repair* hire in order to make it satisfy the postcondition. Additionally, we would like to avoid *obvious* repairs that result in undesirable programs. For instance, the program that hires everyone (return true) obviously satisfies the postcondition. To avoid such programs, we look for a repair that minimizes the *semantic distance* between the new program and the original program.

For our example, we will restrict the space of possible repairs as a *sketch* of the original program—we call this the *repair model*:

```
fun hireRep(min, urank)
  dec = ●1 <= urank <= ●2
  return dec
```

The repair model is a parametric program with two *holes* to fill, $\bullet_1$ and $\bullet_2$, which we can replace with constants to produce a program that satisfies the postcondition. Note that our approach is more general and not only restricted to filling holes with constants.

Distribution-Guided Inductive Synthesis. Now that we have set up the problem, we are ready to illustrate our approach. We are looking for a new function hireRep that satisfies *post* and that minimizes the *semantic distance* $\mathbb{P}(\mathsf{hire} \neq \mathsf{hireRep})$, which denotes the probability that hire and hireRep return different outputs for the same input, which is distributed according to *pre*.

To find such a repair, we present *distribution-guided inductive synthesis* (DIGITS). DIGITS begins by sampling a finite set of inputs $S = \{s_1, \ldots, s_n\}$ from the precondition *pre*. The set of samples are used to *guide* the synthesis process with concrete examples from the distribution. DIGITS considers every possible partition of the samples into positive and negative samples, (S^+, S^-). For every such partition, it attempts to find a repair that returns true for all inputs in S^+ and false for all inputs in S^-. To perform the synthesis, we encode the search problem as a quantifier-free first-order formula and ask an SMT solver to find a solution that corresponds to a filling of the holes. For each synthesized program, DIGITS uses probabilistic program verification techniques to check if the program satisfies the postcondition and to quantify the semantic distance from hire.

There are two obvious issues here: (i) The number of partitions of a given set S is exponential in $|S|$. (ii) The search finds an arbitrary repair at every step; how do we ensure that we eventually find a repair that satisfies the postcondition?

First, while the number of partitions of S is exponential, DIGITS employs an efficient binary trie data structure to guide and prune the search space. For instance, if there is no repair for a partition (S^+, S^-), DIGITS utilizes UNSAT cores to remember an unsatisfiable subset of the samples and ensure that similar partitions are not considered, thus pruning away a large family of possible partitions.

Second, we theoretically demonstrate elegant properties of DIGITS that ensure it converges to an optimal solution. The formalization is rooted in classic ideas from computational learning theory, namely, VC dimension of our repair model.

Finding Repairs. DIGITS iteratively increases the size of the sample set S by drawing more samples from *pre*. The more samples it considers, the more likely it synthesizes programs that are close to an optimal solution. Let us consider a possible trajectory of DIGITS. Figure 2 shows three programs that can be produced by DIGITS in the course of sampling and synthesis. While all programs satisfy the postcondition, the *best* repair is hireRep3, as it has the smallest distance from the original program hire. Specifically, hireRep hires all applicants from the top-15 universities, making it the semantically closest one to hire.

3 The Probabilistic Repair Problem

In this section, we formally define the probabilistic repair problem.

```
fun hireRep1(min,urank)
  dec = 10 <= urank <= 15
  return dec
```

(a) distance ≈ 0.5

```
fun hireRep2(min,urank)
  dec = 5 <= urank <= 17
  return dec
```

(b) distance ≈ 0.4

```
fun hireRep3(min,urank)
  dec = 0 <= urank <= 15
  return dec
```

(c) distance ≈ 0.2

Fig. 2. Repairs synthesized by DIGITS

Program Model. We consider a simple program model where a program is written in a loop-free language whose syntax is defined below:

$$P := \; V \leftarrow E \mid \mathsf{if}\ B\ \mathsf{then}\ P\ \mathsf{else}\ P \mid P\ P \mid \mathsf{return}\ V$$

P is a program, V is the set of variables used in P, E is the set of linear arithmetic expressions over V, and B is the set of Boolean expressions over V. $V \leftarrow E$ denotes assigning an expression to a variable. We assume that there is a vector of variables $\boldsymbol{v}_I$ in V that are inputs to the program and never appear on the left-hand side of an assignment. We also assume there is a single Boolean variable $v_r \in V$ that is returned by the program. All variables are real-valued or Boolean. We always assume that programs are well-typed. Given a vector of constant values $\boldsymbol{c}$, where $|\boldsymbol{c}| = |\boldsymbol{v}_I|$, we use $P(\boldsymbol{c})$ to denote the result of executing P on the input $\boldsymbol{c}$.

Probabilistic Preconditions. Given a program P with variables V, we define a probabilistic precondition *pre* as a *joint probability distribution* over the variables $\boldsymbol{v}_I$. That is, we assume that the values of the inputs are initially drawn from the probability distribution *pre*.

Formally, we think of the distribution *pre* as a *probability space* $(\Omega, \mathcal{F}, \mathbb{P})$: Ω is the set of possible assignments to $\boldsymbol{v}_I$, $\mathcal{F} \subseteq 2^{\Omega}$ is a set of *events*, and $\mathbb{P} : \mathcal{F} \to [0, 1]$ denotes the probability of an event.

We will be interested in two kinds of events:

1. Given a Boolean expression B over $\boldsymbol{v}_I$ and v_r, overloading notation, a probability expression $\mathbb{P}(B)$ denotes

$$\mathbb{P}(\{\boldsymbol{c} \in \Omega \mid \exists r.\, P(\boldsymbol{c}) = r \wedge B[\boldsymbol{v}_I/\boldsymbol{c}, v_r/r] = \mathit{true}\})$$

   where the notation $B[\boldsymbol{x}/\boldsymbol{y}]$ denotes B with all occurrences of $\boldsymbol{x}$ replaced by $\boldsymbol{y}$. That is, $\mathbb{P}(B)$ is the probability of drawing a sample $\boldsymbol{c}$ from the precondition such that the program P returns a result satisfying B.
2. Suppose we are given two programs P and P' such that $\boldsymbol{v}_I$ and $\boldsymbol{v}'_I$ are of the same length and type. We will use $\mathbb{P}(P \neq P')$ to denote:

$$\mathbb{P}(\{\boldsymbol{c} \in \Omega \mid P(\boldsymbol{c}) \neq P'(\boldsymbol{c})\})$$

   That is, $\mathbb{P}(P \neq P')$, which we call the *semantic distance*, is the probability that the two programs return different results on the same input.

```
Procedure DIGITS($P, pre, post, R, n$)
    Input : Repair problem s.t. $\{pre\}P\{post\}$ does not hold, and a number $n$
    Output: Program $P' \in R$ such that $\{pre\}P\{post\}$ holds or $\bot$
    $S \leftarrow \emptyset$
    for $n$ times do
        $s \sim pre$
        $S \leftarrow S \cup \{s\}$
    $repairs \leftarrow \emptyset$
    foreach sets $S^+, S^-$ that partition $S$ do
        $P' \leftarrow$ REPAIR($S^+, S^-$)
        if $P' \neq \bot$ and $\{pre\}P'\{post\}$ then
            $repairs \leftarrow repairs \cup \{P'\}$
    if $repairs \neq \emptyset$ then
        return $P' \in repairs$ with minimal $\mathbb{P}(P \neq P')$
    else
        return $\bot$
```

Algorithm 1. Distribution-Guided Inductive Synthesis

Probabilistic Postconditions. Given a program P and a precondition *pre*, we would like to refer to the probability of the program to return a specific set of values. To that end, we define a *probabilistic postcondition*, *post*, as an inequality over terms of the form $\mathbb{P}(B)$, where B is a Boolean expression over $\boldsymbol{v}_I$ and v_r. Specifically, a probabilistic postcondition is of the form $e > c$, where $c \in \mathbb{R}$ and e is an arithmetic expression over terms of the form $\mathbb{P}(B)$, e.g., $\mathbb{P}(B_1)/\mathbb{P}(B_2) > 0.75$.

Program Correctness. Given a triple $(P, pre, post)$, we say that P is *correct* with respect to *pre* and *post*, denoted $\{pre\}P\{post\}$, *iff* *post* is true.

Repair Problem. The *probabilistic repair* problem is a tuple $(P, pre, post, R)$, where $(P, pre, post)$ are as defined above, and R is a set of programs called the *repair model*, i.e., the set of possible repairs. A *solution to a repair problem* is a program $P' \in R$ such that $\{pre\}P'\{post\}$ holds, and the semantic distance $\mathbb{P}(P \neq P')$ is minimal.

The semantic distance condition is present to try to preserve as much of the original program behavior as possible.

4 Distribution-Guided Inductive Synthesis

In this section, we describe the distribution-guided inductive synthesis algorithm (DIGITS) for finding approximate solutions to the probabilistic repair problem.

DIGITS, shown in Algorithm 1, takes as input a repair problem and a number n that bounds the search depth. DIGITS first builds a set S of n samples from *pre*, and then, for every possible way to split S into positive and negative examples S^+ and S^-, it finds a candidate repair $P' \in R$ consistent with S^+ and S^-. DIGITS finally outputs the candidate repair semantically closest to P. Intuitively, DIGITS tries to inductively learn the correct repair from a finite set of samples.

Example 1. Recall the example from Sect. 2. Suppose we are given two samples $s_1 = 12$ and $s_2 = 17$, where we only consider the variable urank in the sample, as min is not used by the program. Suppose $S^+ = \{12, 17\}$ and $S^- = \emptyset$. Then, the program hireRep2 in Fig. 2 correctly classifies S^+ and S^-. Alternatively, suppose we consider the sets $S^+ = \{12\}$ and $S^- = \{17\}$. Then, a potential repair is hireRep3. If we were to add a new sample $s_3 = 15$, there would not be a repair for the sets $S^+ = \{12, 17\}$ and $S^- = \{15\}$.

To implement DIGITS, one needs to provide two components: (a) the procedure REPAIR that produces programs consistent with labeled examples and (b) a (sound) probabilistic inference algorithm to (i) check whether the synthesized program satisfies *post*, and (ii) compute the probability $\mathbb{P}(P \neq P')$. In the following we assume that such components are given. The DIGITS algorithm is relatively simple, but we show that it enjoys interesting convergence properties.

4.1 Convergence of Digits

In this section, we use classic concepts from computational learning theory to show that, under certain assumptions, the DIGITS algorithm quickly converges to good repaired programs when increasing the size n of the sample set.

Throughout this section we assume we are given a program P, a repair model R, a precondition *pre*, and a postcondition *post*, such that there exists an optimal solution $P^* \in R$ to the corresponding probabilistic program repair problem. The relationship between P, R, the programs which satisfy *post*, and P^* is visualized in Fig. 3(a). Given two programs P' and P'', we write $Er(P', P'') = \mathbb{P}(P' \neq P'')$ to denote the distance (error) between the two programs; we introduce this additional notation to make the connections to computational learning theory more explicit.

To state our main theorem, we need to recall the concept of Vapnik–Chervonenkis (VC) dimension from computational learning theory [18]. Intuitively, the VC dimension captures the expressiveness of a certain concept class; in our setting, the concept class is the repair model R. Given a set of examples S, we say that the repair model R *shatters* S iff, for every two sets S^+ and S^- that partition S, there exists a program $P_1 \in R$ such that (i) for every $s \in S^+$, $P_1(s) = true$, and (ii) for every $s \in S^-$, $P_1(s) = false$. The *VC dimension* of the repair model R is the largest integer k such that there exists a set of examples S with cardinality k that is shattered by R.

Example 2. Consider the class of linear separators in $\mathbb{R}^2$. For any collection and classification of three non-colinear points in $\mathbb{R}^2$, it is possible to construct a linear separator that is consistent with that classification; therefore, linear separators *shatter* any set of size 3. However, no linear separator can shatter *any* set of four points—for example, the points $\{(0,0), (1,1)\}$ cannot be separated from $\{(1,0), (0,1)\}$; Thus, the VC dimension of linear separators is 3.

We define the function $\text{VC}\textsc{cost}(\varepsilon, \delta, k) = \frac{1}{\varepsilon}(4 \log_2(\frac{2}{\delta}) + 8k \log_2(\frac{13}{\varepsilon}))$ [7], which we will use in the following theorems.

Lemma 1 (Error Bound of digits). *If the repair model R has finite VC dimension k, then, for every program $P' \in R$, function* REPAIR*, bounds $\varepsilon > 0$ and $\delta > 0$, and set of samples S drawn from pre of size $n \geq \text{VC}\textsc{cost}(\varepsilon, \delta, k)$, there exist sets S^+ and S^- that partition S such that, with probability $\geq 1 - \delta$, we have that $Er(P', \textsc{repair}(S^+, S^-)) \leq \varepsilon$.*

Lemma 1 extends the classic notion of learnability of concept classes with finite VC dimension [7] to probabilistic program repair. Intuitively, if a repair model R has finite VC dimension, any function that correctly synthesizes from finitely many samples in *pre* will get arbitrarily close to a target solution—including P^*—with polynomially many samples. Lemma 1, however, does *not* guarantee that the synthesis algorithm will find a program consistent with the postcondition.

Intuitively, we need to ensure that there are *enough* programs close to P^* that satisfy *post*; to do so, we reason about how the error on the repair problem propagates to the error on the postcondition. Specifically, for a program $\hat{P}$ and $\alpha > 0$, we define the set $B_\alpha(\hat{P}) = \{P' \in R \mid Er(\hat{P}, P') \leq \alpha\}$; in other words, $B_\alpha(\hat{P})$ exactly characterizes a *ball* of programs in the repair model that are *close* to $\hat{P}$. Now, we define a notion of *robustness* of the postcondition with respect to a program $\hat{P}$: we say that the pair $(\hat{P}, \mathit{post})$ (or just $\hat{P}$ when *post* is clear from context) is α-*robust* iff

$$\forall P' \in B_\alpha(\hat{P}).\, \{\mathit{pre}\}P'\{\mathit{post}\}$$

Figure 3(b) visualizes how the convergence of DIGITS follows from α-robustness: if $\hat{P}$ is α-robust, then DIGITS invoked on a sufficiently large set of samples S will, with high probability, encounter a split S^+, S^- where every program consistent with that split is contained in $B_\alpha(\hat{P})$. Thus if P' is the result of $\textsc{repair}(S^+, S^-)$, then $Er(\hat{P}, P') \leq \alpha$ and P' satisfies *post*. We can now give our main theorem, which formalizes this property.

Theorem 1 (Convergence of digits). *Assume that there exist an $\alpha > 0$ and program $\hat{P}$ such that $(\hat{P}, \text{post})$ is α-robust. Let k be the VC dimension of repair model R. For all bounds $0 < \varepsilon \leq \alpha$ and $\delta > 0$, for every function* REPAIR *and $n \geq \text{VC}\textsc{cost}(\varepsilon, \delta, k)$, with probability $\geq 1 - \delta$ we have that* DIGITS *enumerates a program P' with $Er(\hat{P}, P') \leq \varepsilon$ and $\{\text{pre}\}P'\{\text{post}\}$.*

Corollary 1 (Convergence to P^*). *In particular, if P^* is α-robust, and ε, δ, and n are constrained as above, and $\textsc{digits}(P, \text{pre}, \text{post}, R, n) = P'$, then with probability $\geq 1 - \delta$ we have that $P' \neq \bot$, $Er(P^*, P') \leq \varepsilon$, and $\{\text{pre}\}P'\{\text{post}\}$.*

Theorem 1 and Corollary 1 represent the heart of the convergence result. However, there are two major technicalities.

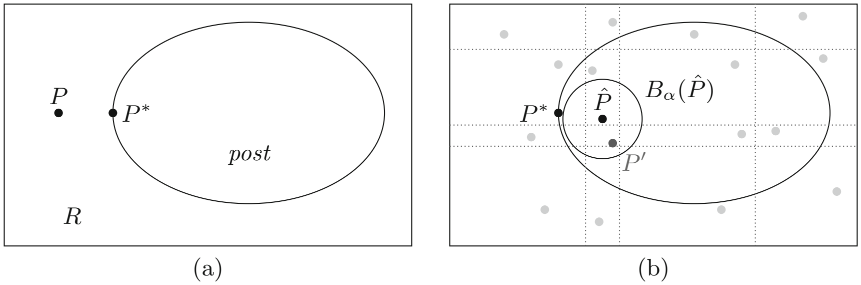

Fig. 3. Visualization of aspects of DIGITS: (a) Programs that satisfy *post* are a subset of R. (b) Samples split R into 16 regions, each with a candidate program. If $\hat{P}$ is α-robust, with high probability DIGITS finds P' close to $\hat{P}$; if $\hat{P}$ is close to P^*, so is P'.

First, P^* usually is not α-robust; in particular, if there exists $P' \in B_\alpha(P^*)$ with $Er(P, P') = Er(P, P^*) - \alpha$, then P^* is not actually optimal. In other words, We can expect P^* to lie on the boundary of the set of correct programs, as in Fig. 3(a). However, Theorem 1 still guarantees that with high probability, DIGITS will find a solution arbitrarily close to *any* α-robust program $\hat{P} \in R$; if there exist α-robust programs that are close to the optimal solution, DIGITS still converges to the optimal solution. We refine this notion in the following Corollary.

Corollary 2 (Weak convergence to P^*). *For $\alpha > 0$, let $A \subseteq R$ be the set of programs $\hat{P}$ where $(\hat{P}, \mathrm{post})$ is α-robust. Let $\Delta = \min_{\hat{P} \in A}\{Er(P^*, \hat{P})\}$. If ε, δ, and n are constrained as above, and* DIGITS$(P, \mathrm{pre}, \mathrm{post}, R, n) = P'$*, then with probability $\geq 1 - \delta$ we have that $P' \neq \bot$, $Er(P^*, P') \leq \Delta + \varepsilon$, and $\{\mathrm{pre}\}P'\{\mathrm{post}\}$.*

Extensions of Corollary 2 still provide strong results on the convergence of DIGITS: for example, if P^* is not α-robust, but there exists an α-robust $\hat{P}$ with $P^* \in B_\alpha(\hat{P})$, then one can show $\lim_{\alpha \to \infty} \Delta = 0$; in this case, running DIGITS for sufficiently large n preserves the desired convergence result from Corollary 1.

Second, an optimal P^* that satisfies *post* may not actually exist. Suppose, for example, that for some event E, *post* is the expression $\mathbb{P}(E) > 0.5$, and when evaluated on the input program P, $\mathbb{P}_P(E) = 0.4$. Then for every repaired program P' with $\mathbb{P}_{P'}(E) = 0.5 + \epsilon$, there may exist P'' that satisfies *post* with $\mathbb{P}_{P''}(E) = 0.5 + \frac{\epsilon}{2}$ and $Er(P, P'') = Er(P, P') - \frac{\epsilon}{2}$; the limit of this process could give us P^* with $\mathbb{P}_{P^*}(E) = 0.5$, but now P^* no longer satisfies *post*. To resolve this, we take P^* to be the infimum with respect to $Er(P, \cdot)$ of the set $\{P' \in R \mid \{pre\}P'\{post\}\}$. Since this P^* does not satisfy *post*, it is trivially not α-robust, and we rely on the result of Corollary 2.

The convergence of DIGITS relies on the existence of α-robust programs. Theorem 1—which follows directly from α-robustness—gives us a way to check, with high probability, whether any α-robust programs exist: we can run the algorithm for the number of iterations given by Theorem 1 for arbitrarily small δ and just see whether any solution for the program repair problem is found. If not, we can infer that with probability $1 - \delta$ no α-robust programs exist. The success of DIGITS in our evaluation (Sect. 6) suggests, as we might expect, that this would be a pathological case.

4.2 Efficient Search Strategy and Data Structure

The DIGITS algorithm is fairly abstract and opens many doors to optimizations. In this section, we present a concrete data structure for implementing DIGITS and show how it can be used to run the REPAIR algorithm on smaller inputs than with a naïve implementation.

We propose to use a binary trie of height n to describe all the possible ways to partition the set of samples $S = \{s_1, \ldots, s_n\}$ into two sets S^+ and S^-. In the trie, each node at depth i corresponds to splitting on the sample s_i; a 0-labeled edge (resp. 1-labeled edge) from depth i to depth $i+1$ denotes that, in this path, $s_i \in S^-$ (resp. $s_i \in S^+$).

We use $\{0,1\}^{\leq n}$ to denote the set of strings of length at most n over the alphabet $\{0,1\}$. A binary trie for a set of samples $S = \{s_1, \ldots, s_n\}$ is a function $f : \{0,1\}^{\leq n} \mapsto R \cup \{\bot\}$ that maps strings to repaired programs. Given a string $\boldsymbol{b} = b_1 \ldots b_k \in \{0,1\}^{\leq n}$, let $S^+_{\boldsymbol{b}}$ (resp. $S^-_{\boldsymbol{b}}$) be the set of all $s_i \in S$ such that $b_i = \mathit{true}$ (resp. $b_i = \mathit{false}$). We define $f(b_1 \ldots b_k)$ as $\textsc{repair}(S^+_{\boldsymbol{b}}, S^-_{\boldsymbol{b}})$.

One of the many advantages of using this trie representation is that it allows us to dynamically increase the sample set size n without restarting the algorithm: whenever all strings of length at most n have been exhausted, simply sample an additional point and compute f for all strings of length $n+1$. Thus, instead of fixing the sample size a priori, the algorithm can run continuously, adding more samples as needed, until meeting some stopping criteria.

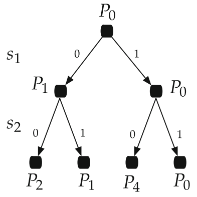

Fig. 4. Trie ex.

Example 3. Figure 4 shows a trie of height 2 for Example 1. Here only consider the samples s_1 and s_2. Each layer in the trie corresponds to a sample, and each node is assigned a candidate program consistent with the samples. For example, $f(0,1) = P_1$ is a program consistent with $S^+ = \{17\}$ and $S^- = \{12\}$. Note that P_1 is also consistent with $S^+ = \{\}$ and $S^- = \{12\}$, thus $f(0)$ can also be P_1.

Solution Propagation. Our first optimization builds on the idea illustrated at the end of Example 3 and propagates solutions down the trie, therefore reducing the number of times we call the function REPAIR and the average number of samples on which the function is called.

In the following we assume that the function REPAIR has the following property (this assumption simplifies our presentation, but does not affect convergence of our algorithm): given two sets of samples S^+, S^- and a new sample s, (i) if $\textsc{repair}(S^+, S^-) = P'$ and $P'(s) = \mathit{true}$, then $\textsc{repair}(S^+ \cup \{s\}, S^-) = P'$, and (ii) if $\textsc{repair}(S^+, S^-) = P'$ and $P'(s) = \mathit{false}$, then $\textsc{repair}(S^+, S^- \cup \{s\}) = P'$.

To compute the binary trie of height n, we also need to compute repairs for nodes $b_1 \dots b_k$ such that $k < n$, and therefore it would seem we have to call the function REPAIR $2^{n+1} - 1$ times instead of the 2^n required by the DIGITS algorithm. The following theorem allows us to avoid this problem.

Theorem 2 (Solution propagation). *Given a string $b_1 \dots b_k \in \{0,1\}^{\leq n}$ of length $k < n$, if $f(b_1 \dots b_k) = P'$ and $P'(s_{k+1}) = b$, then $f(b_1 \dots b_k b) = P'$.*

Informally, the above theorem states that the program corresponding to a certain node has to be already consistent with one of the two labeling of the following examples. Therefore, even though there are $2^{n+1} - 1$ nodes in the binary trie, we only need to call the function REPAIR for half of the 2^i nodes at depth i, or 2^n nodes total.

Additionally, the calls to REPAIR have fewer constraints: while the DIGITS algorithm as presented in Algorithm 1 always calls the function REPAIR on exactly n samples, when using the trie structure with solution propagation, REPAIR is called only on i samples at depth i. One can show that the average number of samples used by REPAIR asymptotically approaches $n - 1$.

Conflict-Driven Pruning. While solution propagation is a good strategy for reusing successful solutions, we can also learn from the instances in which REPAIR returns $\bot$. In particular, let's say that for some string $\boldsymbol{b} = b_1 \dots b_k \in \{0,1\}^{\leq n}$, we have that $f(b_1 \dots b_k) = \bot$. Trivially, we know that for every $b_1 \dots b_k b_{k+1} \dots b_{k+j} \in \{0,1\}^{\leq n}$ $f(b_1 \dots b_k b_{k+1} \dots b_{k+j}) = \bot$. Using this idea, we can prune the search and avoid calling the function REPAIR on partitions that are trivially going to fail. More generally, when a failure occurs, we can identify a subset of the labelings that caused the failure and use it to reduce the set of explored nodes.

Theorem 3 (Conflict-driven pruning). *Let $\boldsymbol{b} = b_1, \dots, b_k$. Let $\boldsymbol{b}'$ be a subsequence of $\boldsymbol{b}$, e.g., $b_2 b_{10} b_{11} b_{20}$. If* REPAIR$(S^+_{\boldsymbol{b}'}, S^-_{\boldsymbol{b}'}) = \bot$*, then $f(b_1 \dots b_k) = \bot$.*

While detecting what subsets of the samples caused the failure can be hard, this theorem can be used to vastly reduce the number of times the function REPAIR is called. In our implementation, we will use the *unsatisfiable cores* produced by the SMT solver to compute the subsets of the samples that induce failures.

5 Implementation

We implemented an instantiation of the DIGITS algorithm in Python. The DIGITS algorithm is abstract and modular. Therefore, to implement it we need to provide a number of components: a repair model R, the procedure REPAIR that produces programs consistent with labeled examples, and a probabilistic inference algorithm to (i) check whether the synthesized program respects the postcondition, (ii) compute the semantic difference between the synthesized program P' and the original program P. In this section, we describe the concrete choices of these components for our implementation.

Repair Model. Since we are mostly interested in repairing machine learning classifiers, a natural repair model R is only allowing modifications to real-valued constants appearing in the program. These constants are essentially the *weights* of the classifier.

Formally, let P be the program we are trying to repair, and let $c_1, \ldots, c_n$ be all of the constants appearing in P. For simplicity, assume all constants are different. Given constants $d_1, \ldots, d_k$, we write $P[c_1/d_1, \ldots, c_n/d_n]$ to denote the program in which each constant c_i has been replaced with the constant d_i. Finally, the set of allowed repairs is defined as

$$R = \{P[c_1/d_1, \ldots, c_n/d_n] \mid d_1, \ldots, d_n \in \mathbb{R}\}.$$

We only consider programs containing linear real arithmetic expressions. As such, our repair model can be viewed as a set of unions of polytopes with a bounded number of faces (bounded by the size of the program). It can be shown such polytopes have finite VC dimension [25], and therefore, so does our repair model.

Repair *Implementation.* The implementation of $\textsc{repair}(S^+, S^-)$ follows a sketch-like approach [26], where we encode the program and the samples as a formula whose solution is a *filling* of the *holes* defined by the repair model R.

Let P be the program we are trying to repair, and let $c_1, \ldots, c_n$ be all of the constants appearing in P, as discussed above. We will first create a new program $P_R = P[c_1/h_1, \ldots, c_n/h_n]$, where $h_1, \ldots, h_n$ are fresh variables that do not appear in P. We call h_i *holes*. We now encode the program P_R as a formula as follows, using the function $\textsc{enc}$. To simplify the encoding, and without loss of generality, we assume that P_R is in *static single assignment* ($\textsc{ssa}$) form.

$$\textsc{enc}(v \leftarrow E) \triangleq v = [\![E]\!] \qquad \textsc{enc}(P_1\ P_2) \triangleq \textsc{enc}(P_1) \wedge \textsc{enc}(P_2)$$
$$\textsc{enc}(\mathsf{if}\ B\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2) \triangleq ([\![B]\!] \Rightarrow \textsc{enc}(P_1)) \wedge (\neg[\![B]\!] \Rightarrow \textsc{enc}(P_2))$$

where $[\![B]\!]$ is the denotation of an expression, which, in our setting, is a direct translation to a logical statement. For example, $[\![\mathtt{x} + \mathtt{y} > \mathtt{0}]\!] \triangleq \mathtt{x} + \mathtt{y} > \mathtt{0}$.

Once we have encoded the program P_R as a formula φ, for each sample $s_i \in S^+$, we will construct the formula

$$\varphi_i \triangleq \exists V.\, \varphi[\boldsymbol{v}_I/s_i] \wedge v_r = \mathit{true}$$

where V is the set of variables of P, which do not include the introduced holes $h_1, \ldots, h_n$. Similarly, for each sample $s_i \in S^-$, we will construct the formula

$$\varphi_i \triangleq \exists V.\, \varphi[\boldsymbol{v}_I/s_i] \wedge v_r = \mathit{false}$$

Finally, a model to the formula $\bigwedge_i \varphi_i$ is an assignment to the holes $h_1, \ldots, h_n$ that corresponds to a program in the repair model R that correctly labels the positive and negative examples. Specifically, $\textsc{repair}(S^+, S^-)$ finds a model $m \models \bigwedge_i \varphi_i$ and returns the program $P_R[h_1/m(h_1), \ldots, h_n/m(h_n)]$, where $m(h_i)$ is the value of h_i in the model m. If $\bigwedge_i \varphi_i$ is unsatisfiable, then $\textsc{repair}$ returns $\bot$.

Theorem 4 (Soundness and completeness of repair). *Suppose we are given a program P and the repair model R defined above, along with two sets of samples S^+ and S^-. Then, if* REPAIR(S^+, S^-) *returns a program P', P' must appear in R and correctly classifies S^+ and S^-. Otherwise, there is no program $P' \in R$ that correctly classifies S^+ and S^-.*

Probabilistic Inference. In our implementation, this component can be instantiated with any probabilistic inference tool—e.g., PSI [15]. We use the tool FairSquare [4], which is also written in Python and has already been used to verify fairness properties of decision-making programs. Moreover, unlike several other tools, the inference algorithm used in FairSquare is sound and complete and therefore meets the criteria of the DIGITS algorithm.

To speed up the search, we use sampling to approximate the probabilistic inference and quickly process obvious queries. At the end of the algorithm we use FairSquare to verify the output of DIGITS.

6 Evaluation

In this section, we evaluate the effectiveness of our algorithm on benchmarks obtained by training machine learning models on an online dataset [1]. First, we show that our algorithm can produce good repairs on many of the benchmarks. Second, we illustrate the efficacy of the optimizations discussed in Sect. 4.2.

Benchmarks. We used an online dataset [1] comprised of 14 demographic features for over 30,000 individuals to generate a number of classifiers and a probabilistic precondition. The precondition uses a graph structure represented as a probabilistic program: at each node, there is an inferred Gaussian distribution for a variable, and the edges of the graph induce correlations between variables.

We generated *support vector machines* with *linear kernels* (SVMs) and *decision trees* (DTs) to classify high- versus low-income individuals using the WEKA data mining software [2] until we obtained 3 SVMs and 3 DTs that did *not* satisfy a probabilistic postcondition describing group fairness. In particular, we used the following postcondition:

$$\frac{\mathbb{P}(\text{high income} \mid \text{female})}{\mathbb{P}(\text{high income} \mid \text{male})} \geq 0.85$$

The learned models are small and employ at most three features. Most of the generated models violated the postcondition because they were strongly influenced by a particular feature, *capital gain*, which was highly correlated with gender in the dataset.

The combined size of the precondition and decision-making program ranges from 20 to 100 lines of code. Though this is a much smaller scale than industrial applications of machine learning, the repair problems are highly non-trivial.

Effectiveness of Algorithm. Table 1 details the performance of DIGITS on our suite of benchmarks that were given 600 s to perform repair. For example, on the DT labeled DT_{16}, the table shows that in 584 s, DIGITS was able to enumerate all possible labelings for a set of 50 samples (the depth of the trie), and found a solution satisfying the postcondition that differs from the original program with probability 0.098. Despite the fact that there are $2^{50} \approx 10^{15}$ such labelings, DIGITS needed only to check 53,255 of these possibilities (nodes in the trie): among these possibilities, DIGITS calls REPAIR for just 1,903 of the labelings that have a consistent solution, avoiding 26,627 (93%) potential calls to REPAIR using solution propagation—each of these also avoids a call to the verification oracle. Additionally, DIGITS calls REPAIR for just 1,064 of the labelings that are inconsistent and return $\perp$, avoiding 23,661 (95%) potential calls to REPAIR that would also return $\perp$ using conflict-driven pruning.

It is visibly apparent that solution propagation and conflict-driven pruning save many synthesis and verification queries. However, savings from conflict-driven pruning are only possible once the depth of the search (the number of constraints) is large enough that many labelings are inconsistent—when the number of constraints exceeds the VC dimension of the repair model. Therefore, we expect the instances with a more expressive repair model to perform worse.

Accordingly, Table 1 includes multiple results for each of the SVMs, where the number of holes is varied: the SVMs compare an expression $c_0 + c_1x_1 + c_2x_2 + \ldots$ to 0, where each c_i is replaced with a hole. The variants with fewer holes are

Table 1. Results of running DIGITS on benchmarks with a 600 s best-effort period. Solution propagation, conflict-driven pruning, and cost minimization are used for all.

Name	Holes	Result of DIGITS			Trie details				
		Samples	Time (s)	Sem. diff.	Nodes	Consistent	Soln. prop.	Incon.	Unsat pruned
DT_{16}	5	50	584	0.098	53,255	1,903	26,627	1,064	23,661
DT_{44}	3	91	553	0.03	68,051	1,741	34,025	418	31,867
DT_4	2	79	594	0.14	94,733	1,746	47,366	707	44,914
SVM_3	4	16	561	0.067	7,015	1,000	3,507	338	2,170
	3	23	587	0.0615	17,977	1,600	8,988	388	7,001
	2	91	582	0.0455	123,935	1,980	61,967	247	59,741
	1	661	599	0.0595	437,583	662	218,791	1,305	216,825
SVM_4	5	10	470	0.197	1,523	508	761	100	154
	4	13	507	0.204	4,599	1,018	2,299	205	1,077
	3	22	559	0.195	14,885	1,424	7,442	406	5,613
	2	83	598	0.04	92,495	1,674	46,247	217	44,357
	1	628	600	0.044	395,013	629	197,506	1,228	195,650
SVM_5	6	8	410	0.1305	507	240	253	11	3
	5	10	568	0.122	1,693	632	846	64	151
	4	13	559	0.1025	4,587	1,018	2,293	194	1,082
	3	23	575	0.096	15,361	1,348	7,680	332	6,001
	2	88	583	0.056	103,649	1,736	51,824	224	49,865
	1	598	599	0.067	358,203	599	179,101	1,176	177,327

generated by removing the holes for coefficients of x_i in increasing order of the mutual information between x_i and gender, as per the precondition. The last remaining hole allows only for the constant offset c_0 to be changed. The table illustrates the trade-off between expressivity and performance: though the instances with more holes have a strictly larger repair model and thus have the potential to contain solutions with better semantic difference, the search is slow, and the trie cannot enumerate as large a set of samples: the synthesis queries are over more variables and are more complex; more solutions are satisfiable, so conflict-driven pruning does not provide the same advantages. In general, as the number of holes decreases, the best solution has improving semantic difference because the trie is explored deeper. This trend continues until the only hole that remains is the constant offset, when the repair model is no longer expressive enough to capture solutions with such a minimal difference.

Optimizations. Running DIGITS as a trie allows solution propagation to avoid a synthesis query for half of all explored nodes; additionally, when the queries are performed, they are over smaller sets of constraints. Figure 5 (Left) illustrates for the benchmark DT$_{44}$ the time saved by using the trie structure instead of explicitly enumerating the 2^n possible labelings for a sample set of size n: each point on the red line (labeled *without solution propagation*) indicates the amount of time necessary to explicitly compute the 2^n possibilities, while the blue line (labeled *with solution propagation*) denotes the total time of exploring the trie up to depth n. The plot illustrates that a run of DIGITS using the trie structure's solution propagation provides exponential savings in the size of the sample sets.

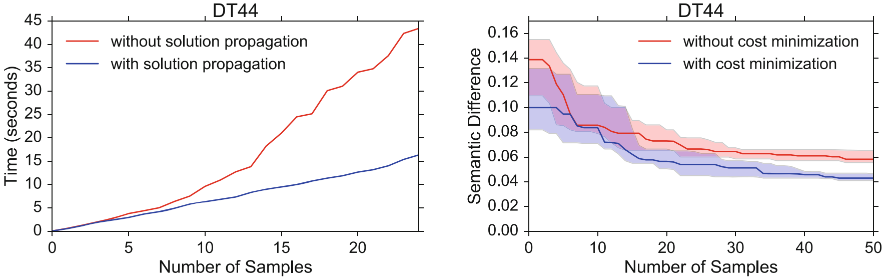

Fig. 5. (Left) efficiency of solution propagation; (Right) efficacy of cost heuristic and convergence of variance.

Syntactic Cost Minimization. Recall that our repair model consists of modifying the value of any real-valued constants in the program. As a heuristic to quickly guide the search to better solutions, we introduce a notion of the *syntactic cost* of a candidate repaired program. Specifically, if the original values of holes $h_1, \ldots, h_n$ are the values $c_1, \ldots, c_n$, then we compute the cost as $\sum_i \left|\frac{h_i - c_i}{c_i}\right|$; whenever we submit an SMT query for a set of constraints, we require that it approximately minimizes this cost function. The intuition is that since the behaviors of these programs are entirely determined by their constants' values, the amount that these values are changed is correlated with the semantic difference. Syntactic cost minimization is utilized in all our prior results.

Figure 5 (Right) contains the results of running DIGITS on DT_{44} for 20 different random seeds using cost minimization, and the same 20 different random seeds without using cost minimization. The solid lines denote the median value for the best semantic difference across the 20 runs as a function of the depth reached by the trie structure; the transparent region denotes the 90% confidence interval of the best semantic difference across all runs. It illustrates two concepts: first, that the use of the cost minimization heuristic allows for DIGITS to converge to better solutions faster. Second, it shows that while the variance between the best solutions across the different runs is high for a small number of samples, this variance decreases as the number of samples increases, suggesting DIGITS is robust with respect to the exact values of the sampled points.

7 Related Work

Program Repair and Synthesis. Automated repair has been studied in the non-probabilistic setting [10,17,20,22,27]. Closest to our work is the tool Qlose, which attempts to repair a program to match a set of test cases while attempting to minimize a mixture of syntactic and semantic distances between the original and repaired versions [10]. The approach in Qlose itself cannot be directly lifted to probabilistic programs and postconditions because it relies on a finite set of input-output examples—it finds candidates for repairs by making calls to an SMT solver with the hard constraint that the examples should be classified correctly. In our setting, the output of the optimal repair on the samples is not known a priori and our goal is to ultimately find a program that satisfies a probabilistic postcondition over an infinite set of inputs. Several of Qlose's general principles do carry over: namely, using a sketch-based approach [26] to fix portions of the code and minimizing semantic changes.

In probabilistic model checking, a number of works have addressed the model repair problem, e.g., [6,9]. In this work, the idea is to modify transition probabilities in finite-state Markov Decision Processes to satisfy a probabilistic temporal property. Our setting is quite different, in that we are modifying a program manipulating real-valued variables to satisfy a probabilistic postcondition.

Our problem of repairing probabilistic programs is closely related to the synthesis of probabilistic programs. The technique of *smoothed proof search* [8] approximates a combination of functional correctness and maximization of an expected value as a smooth, continuous function. It then uses numerical methods to find a local optimum of this function, which translates to a synthesized program that is likely to be correct and locally maximal. Unlike our approach, smoothed proof search lacks formal convergence guarantees.

Stochastic Satisfiability. Our problem is closely related to, and subsumes, the problem of E-MAJSAT [21], a special case of *stochastic satisfiability*

(SSAT) [24] and a means for formalizing probabilistic planning problems. E-MAJSAT is of NP^{PP} complexity. In E-MAJSAT, a formula has two sets of propositional variables, a deterministic and a probabilistic set. The goal is to find an assignment of deterministic variables such that the probability that the formula is satisfied is above a given threshold. Our setting is similar, but we operate over formulas in linear real arithmetic and have an additional optimization objective stipulating semantic closeness. The deterministic variables in our setting are the holes defining the repair; the probabilistic variables are program inputs.

Algorithmic Fairness. Concerns of algorithmic fairness are recent, and there are many competing fairness definitions [11–14, 16]. Approaches to enforcing fairness in machine-learned classifiers include altering the data to remove correlations with protected attributes [13] and imposing a fairness definition as a requirement of the learning algorithm [16]. However, the general problem presented in this paper of modifying an existing program (be it learned or manually constructed) to meet a quantitative probabilistic property is novel.

Acknowledgements. This material is based upon work supported by the National Science Foundation under Grant numbers 1566015 and 1652140.

A Proofs

Proof of Lemma 1. We cite the seminal result from [7] that if a *well-behaved*[1] concept class C has VC dimension k, then for any $0 < \varepsilon, \delta < 1$ and sample size at least $\max\{\frac{4}{\varepsilon}\log_2\frac{2}{\delta}, \frac{8k}{\varepsilon}\log_2\frac{13}{\varepsilon}\}$ drawn from probability distribution $\mathcal{D}$ and labeled by their classification by the target concept $c^* \in C$, any concept $c \in C$ consistent with those samples has $\mathit{error}_{\mathcal{D}}(c) \leq \varepsilon$ with probability at least $1-\delta$. Here, $\mathit{error}_{\mathcal{D}}(c)$ is the probability that a sample drawn from $\mathcal{D}$ is classified differently by c^* versus c.

Our program model satisfies the benign measure-theoretic restriction of *well-behavior* since it is equivalent to arbitrary collections of polytopes; therefore, for any $P' \in R$, some labeling of the $\geq \frac{4}{\varepsilon}\log_2\frac{2}{\delta} + \frac{8k}{\varepsilon}\log_2\frac{13}{\varepsilon}$ samples is consistent with P', and therefore the theorem from [7] applies.

Proof of Theorem 1. By Lemma 1, we know that with probability $\geq 1-\delta$, one of the results $P' = \text{REPAIR}(S^+, S^-)$ will have $Er(\hat{P}, P') \leq \varepsilon$. Since $(\hat{P}, \mathit{post})$ is α-robust and $\varepsilon \leq \alpha$, then $P' \in B_\alpha(\hat{P})$, and so $\{\mathit{pre}\}P'\{\mathit{post}\}$ holds.

Proof of Corollary 1. This follows immediately from Theorem 1 by letting $\hat{P} = P^*$.

[1] See [7] Appendix 1 for a discussion of well-behaved concept classes.

Proof of Corollary 2. Observe that the Er function respects the triangle inequality, i.e. $Er(P_1, P_2) \leq Er(P_1, P_3) + Er(P_3, P_2)$.

$$\begin{aligned} Er(P_1, P_2) &= \mathbb{P}(P_1 \neq P_2) \\ &= \mathbb{P}(P_1 \neq P_2 \wedge P_1 \neq P_3) + \mathbb{P}(P_1 \neq P_2 \wedge P_3 = P_1) \\ &= \mathbb{P}(P_1 \neq P_2 \wedge P_1 \neq P_3) + \mathbb{P}(P_1 \neq P_2 \wedge P_3 \neq P_2) \\ &\leq \mathbb{P}(P_1 \neq P_3) + \mathbb{P}(P_3 \neq P_2) \\ &= Er(P_1, P_3) + Er(P_3, P_2) \end{aligned}$$

Thus if $\hat{P}$ is α-robust and $Er(P^*, \hat{P}) = \varDelta$, we know P' has $Er(\hat{P}, P') \leq \varepsilon$ by Theorem 1, and the triangle inequality gives us that $Er(P^*, P') \leq \varDelta + \varepsilon$.

Proof of Theorem 2. P' is consistent with the first $k+1$ labeled samples if and only if it is consistent with the $k+1$-th sample as well as the first k samples.

Proof of Theorem 3. Assume towards a contradiction that $f(b_1 \dots b_k) \neq \bot$, but a subsequence ***b'*** has $\textsc{repair}(S^+_{\boldsymbol{b}'}, S^-_{\boldsymbol{b}'}) = \bot$. Then *adding* constraints, which reduces the set of solutions, *introduced* a new solution.

References

1. UCI machine learning repository: Census income. https://archive.ics.uci.edu/ml/datasets/Adult/
2. Weka. http://www.cs.waikato.ac.nz/ml/weka/
3. Code of federal regulations, July 2014. https://www.gpo.gov/fdsys/pkg/CFR-2014-title29-vol4/xml/CFR-2014-title29-vol4-part1607.xml. Accessed 18 June 2016
4. Albarghouthi, A., D'Antoni, L., Drews, S., Nori, A.: Fairness as a program property. In: FATML, November 2016. http://pages.cs.wisc.edu/sdrews/papers/fatml16.pdf
5. Angwin, J., Larson, J., Mattu, S., Kirchner, L.: Machine bias: there's software used across the country to predict future criminals. And it's biased against blacks, May 2016. https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing. Accessed 18 June 2016
6. Bartocci, E., Grosu, R., Katsaros, P., Ramakrishnan, C.R., Smolka, S.A.: Model repair for probabilistic systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 326–340. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19835-9_30
7. Blumer, A., Ehrenfeucht, A., Haussler, D., Warmuth, M.K.: Learnability and the vapnik-chervonenkis dimension. J. ACM (JACM) **36**(4), 929–965 (1989)
8. Chaudhuri, S., Clochard, M., Solar-Lezama, A.: Bridging boolean and quantitative synthesis using smoothed proof search. In: POPL, vol. 49, pp. 207–220. ACM (2014)
9. Chen, T., Hahn, E.M., Han, T., Kwiatkowska, M., Qu, H., Zhang, L.: Model repair for markov decision processes. In: 2013 International Symposium on Theoretical Aspects of Software Engineering (TASE), pp. 85–92. IEEE (2013)
10. D'Antoni, L., Samanta, R., Singh, R.: Qlose: program repair with quantitative objectives. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 383–401. Springer, Cham (2016). doi:10.1007/978-3-319-41540-6_21

11. Datta, A., Sen, S., Zick, Y.: Algorithmic transparency via quantitative input influence. In: Proceedings of 37th IEEE Symposium on Security and Privacy (2016)
12. Dwork, C., Hardt, M., Pitassi, T., Reingold, O., Zemel, R.S.: Fairness through awareness. In: Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, 8–10 January 2012, pp. 214–226 (2012)
13. Feldman, M., Friedler, S.A., Moeller, J., Scheidegger, C., Venkatasubramanian, S.: Certifying and removing disparate impact. In: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, 10–13 August 2015, pp. 259–268 (2015). http://doi.acm.org/10.1145/2783258.2783311
14. Friedler, S.A., Scheidegger, C., Venkatasubramanian, S.: On the (im)possibility of fairness. CoRR abs/1609.07236 (2016). http://arxiv.org/abs/1609.07236
15. Gehr, T., Misailovic, S., Vechev, M.: PSI: exact symbolic inference for probabilistic programs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 62–83. Springer, Cham (2016). doi:10.1007/978-3-319-41528-4_4
16. Hardt, M., Price, E., Srebro, N.: Equality of opportunity in supervised learning. CoRR abs/1610.02413 (2016). http://arxiv.org/abs/1610.02413
17. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005). doi:10.1007/11513988_23
18. Kearns, M.J., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT press, Cambridge (1994)
19. Kobie, N.: Who do you blame when an algorithm gets you fired?, January 2016. http://www.wired.co.uk/article/make-algorithms-accountable. Accessed 18 June 2016
20. Könighofer, R., Bloem, R.: Automated error localization and correction for imperative programs. In: Formal Methods in Computer-Aided Design (FMCAD 2011), pp. 91–100. IEEE (2011)
21. Littman, M.L., Goldsmith, J., Mundhenk, M.: The computational complexity of probabilistic planning. J. Artif. Intell. Res. **9**(1), 1–36 (1998)
22. Mechtaev, S., Yi, J., Roychoudhury, A.: Directfix: looking for simple program repairs. In: Proceedings of the 37th International Conference on Software Engineering, vol. 1, pp. 448–458. IEEE Press (2015)
23. Miller, C.C.: When algorithms discriminate, July 2015. http://www.nytimes.com/2015/07/10/upshot/when-algorithms-discriminate.html?_r=0. Accessed 18 June 2016
24. Papadimitriou, C.H.: Games against nature. J. Comput. Syst. Sci. **31**(2), 288–301 (1985)
25. Sharma, R., Nori, A.V., Aiken, A.: Bias-variance tradeoffs in program analysis. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, NY, USA, pp. 127–137 (2014). http://doi.acm.org/10.1145/2535838.2535853
26. Solar-Lezama, A.: Program synthesis by sketching. Ph.D. thesis, University of California, Berkeley (2008)
27. Von Essen, C., Jobstmann, B.: Program repair without regret. Formal Methods Syst. Des. **47**(1), 26–50 (2015)